mathematical properties which make it unlikely that it was chosen at random. This does not by any means prove that it is pre-broken, but it does make people nervous. • Performance—DSS is about a hundred times slower for signature verification than RSA with e = 3, and in many applications, signature verification happens frequently and is performance-sensitive. In terms of the other operations, DSS is much faster for key generation, though for most applications the performance of key generation is not an issue since it is not done frequently. RSA and DSS are similar in performance for generating signatures, though DSS has the advantage that some of the signature computation can be precomputed before seeing the message. The NIST reasoning is that for the application of smart cards, this ability to precompute for signatures makes DSS superior, since a human will not need to wait as long when logging into a system. 170 PUBLIC KEY ALGORITHMS 6.5.5 • DSS requires choosing a unique secret number for each message. There are several ways of doing this, but they all have problems (see next section). • Patents—One of the advantages of DSS was that it was not owned by RSADSI or PKP (which own exclusive rights to most other public key techniques) and could be used royalty-free. PKP subsequently acquired a patent by Schnorr, which it claims covers DSS. NIST has never conceded that Schnorr covers DSS, and at this time the issue remains unresolved. 6.5.5 Per-Message Secret Number Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. —John Von Neumann (1951) Both DSS and ElGamal require that the signer generate a unique secret number for each message. If the same secret number were used for two different messages, it would expose the signer's private key. Likewise, if a secret number were predictable or guessable, the signer's private key would be exposed. How is the private key exposed if the secret number for a message is known? In DSS, the signature is Xm = Sm −1(dm + STm) mod q. Remember that Sm is the secret number, dm is the message digest, Tm is gSm mod p mod q, and S is the signer's private key. So if Sm is known, then we can compute (XmSm − dm)Tm −1 mod q = S mod q This is all we need to forge DSS signatures. How is the private key exposed when two messages share the same secret number? In DSS, if m and m' are signed using the same secret number Sm, then we can compute (Xm − Xm') −1(dm − dm') mod q = Sm mod q This is enough to compute S mod q as we did above, allowing us to forge signatures. Similar arguments exist for ElGamal signatures. See Homework Problem 10. There are several ways of generating a unique secret number for each message. Keep in mind that signatures might be done with a device with minimal computational ability. • Use truly random numbers. The problem with this is that it requires special hardware. It's difficult enough to make hardware predictable, but it's even harder to make it predictably unpredictable. 6.6 HOW SECURE ARE RSA AND DIFFIE-HELLMAN? 171 • Use a cryptographic pseudo-random number generator. The problem with this is that it requires nonvolatile storage in order to store its state. • Use a cryptographic hash of a combination of the message and the signer's private key. The problem with this is that it can't be computed until the message is known, eliminating a claimed advantage of DSS and ElGamal over RSA. 6.6 HOW SECURE ARE RSA AND DIFFIE-HELLMAN? A brute force attack (trying all possible keys) requires an exponential amount of overhead. The security of RSA is based on the difficulty of factoring. The security of Diffie-Hellman is based on the difficulty of solving the discrete log problem. These problems have been proven to be equivalently difficult. The best known algorithms for solving them are subexponential (less than exponential), but superpolynomial, (more than any fixed degree polynomial). Because the difficulty is subexponential, the required size of the keys in these public key algorithms is much larger (say 1024 bits) than a corresponding secret key (say 80 bits). At the RSA patent expiration party, Eric Hughes created and performed the following, sung to the tune of Supercalifragilisticexpialidocious. Superpolynomial subexponential runtimes. Even though in practice it would take you several lifetimes, If you ran it long enough you'd always find those two

primes. Superpolynomial subexponential runtimes E to the root-log root-log-log [4×] When I was but a naive lad first coding two's and three's I thought the only "orders of" were trivialities. But when I saw this function something opened up to me The elegance of computational complexity. [Chorus] I was at a meeting when up came a man in black Who told me that his agency had mounted an attack. Convincing him was fruitless that his budget would collapse All I know his trumpeter will soon be playing Taps. [Chorus] In virtual environments has grown up a debate Of whether strong cryptography can overthrow the state. But several such technologies including public key Shall herald in the coming age of crypto-anarchy. 172 PUBLIC KEY ALGORITHMS 6.7 6.7 ELLIPTIC CURVE CRYPTOGRAPHY (ECC) As we said in §6.6 How Secure Are RSA and Diffie-Hellman?, there are known subexponential (but superpolynomial) algorithms for breaking RSA and Diffie-Hellman based on modular arithmetic. Elliptic curve cryptography (ECC) is important because the mathematicians do not (yet?) have subexponential algorithms for breaking it. Therefore, it is believed to be secure with much smaller key sizes, which is important for performance. ECC is a candidate replacement for public key cryptographic schemes like RSA, Diffie-Hellman, ElGamal, DSS, etc. For some of these crytographic schemes, one can replace modular multiplication by elliptic curve multiplication directly, resulting in algorithms referred to as ECC Diffie-Hellman, ECC ElGamal, etc. An elliptic curve is a set of points on the coordinate plane satisfying an equation of the form y2 + axy + by = x3 + cx2 + dx + e. In order to use elliptic curves for, say, Diffie-Hellman, there needs to be some mathematical operation on two points in the set that will always produce a point also in the set. Let's call that operation multiplication, although in ECC it will not look like the multiplication you are used to. And the operation has to be associative, so that you can use the repeated squaring trick to raise a number to a large power in time linear with the length of the exponent. In other words, to "exponentiate" a point by 128, you should be able to "multiply" the point by itself (you've now raised it to the power 2), then multiply the result by itself (you've now raised it to the power 4), multiply the result by itself (to have raised it to the power 8), etc. Since "multiplication" is associative, it will be true that (gx)y = gxy = (gy)x . And it is also important that doing discrete logs is hard (knowing g and gx , it is disproportionately difficult to compute x). ECC can be done with at least two types of arithmetic, each of which gives different definitions of multiplication. When you do regular Diffie-Hellman, you have to specify p and g. When you do ECC-Diffie-Hellman, you have to specify the constants in the elliptic curve equation and the type of arithmetic you are using (to take the place of p), and a point on the elliptic curve (to take the place of g). The two types of arithmetic are • Zp arithmetic (modular arithmetic with a large prime p as the modulus) • GF(2n) arithmetic, which can be done with shifts and ⊕s. This can be thought of as modular arithmetic of polynomials with coefficients mod 2. While more complex to understand, the code to implement ECC is no more complex than one that efficiently does modular arithmetic with big integers. And it is faster, at least for private key operations, since until someone comes up with a subexponential algorithm for breaking ECC, the keys can be smaller. For public key operations, such as signature verification, RSA is likely to be faster, even with larger keys, because it can use a small public exponent. See Chapter 24 Math with AES and Elliptic Curves or ROSI99 for more information on ECC. 6.8 ZERO KNOWLEDGE PROOF SYSTEMS 173 6.8 ZERO KNOWLEDGE PROOF SYSTEMS A zero knowledge proof system only does authentication. It allows you to prove that you know a secret (something associated with your public key) without actually revealing the secret. RSA is a zero knowledge proof system, in the sense that you can prove you know the secret associated with your public key without revealing your private key. However, there are zero knowledge proof systems with much higher performance than RSA, although they do not have the ability to do signatures or encryption. The classic example of a zero knowledge authentication scheme is

based on graphs. A graph is a bunch of vertices connected by a bunch of edges. Typically, we name the vertices and specify the edges as pairs of vertices. We consider two graphs isomorphic if we can rename the vertices of one to get a graph identical to the other. Nobody knows how to efficiently determine whether two arbitrary graphs are isomorphic. The assumption that this is hard forms the basis of the authentication scheme. Alice specifies a large graph (say 500 vertices). She renames the vertices to produce an isomorphic graph. Call the two graphs Graph A and Graph B. Alice knows the mapping that will transform Graph A into Graph B. Nobody else can compute it (in reasonable time). Her public key is the specification of the two graphs. Her private key is the mapping between the two graphs. To prove to Bob that she is Alice, she renames the vertices to find a new set of graphs, say G1, G2,...Gk, which she sends to Bob. Then Bob asks, for each i, for Alice to show him the mapping between Gi and one of Graph A or Graph B. Bob can choose which one, but he can't ask Alice to show both mappings for any i (or else Bob could piece the two mappings together to get a mapping from Graph A to Graph B). If Fred tries to impersonate Alice, he can make some graphs that are mapped from Graph A and some graphs that are mapped from Graph B, but he won't be able to find any graph for which he could show a map for both. So for each graph he sends to Bob, he will have only a 50% chance of successfully showing the requested mapping. For 30 graphs, the odds of Fred successfully impersonating Alice are only 1 in 230, or one in ten billion. Why is this zero knowledge? After Alice proves herself to Bob, Bob knows some graphs with mappings to Graph A, and some with mappings to Graph B. He could have generated these himself, so Alice can't have given him any actual information. The graph-based authentication scheme is unfortunately too inefficient for practical use. The following authentication protocol, while not quite zero knowledge, is extremely efficient. It is a variant of Fiat-Shamir [FEIG87]. Alice establishes a public key consisting of ⟨n,v⟩, where n is the product of two large primes (just like the n in RSA), and v is a number for which only Alice knows the square root mod n. Finding such an n is done just like in RSA. Finding v is really easy. Alice merely selects any random number s and squares it mod n to obtain v. After doing so, Alice can forget n's factors and only remember s as her secret, and divulge ⟨n,v⟩ as her public key. 174 PUBLIC KEY ALGORITHMS 6.8 To prove to Bob that she is Alice, she does the following: 1. Alice chooses k random numbers, r1, r2,...rk. For each ri, she sends ri 2 mod n to Bob. 2. Bob chooses a random subset of the ri 2 and tells Alice which subset he has selected to be known as subset 1. The others will be known as subset 2. 3. Alice sends sri mod n for each ri 2 of subset 1, and sends ri mod n for each ri 2 of subset 2. 4. Bob squares Alice's replies mod n. For those ri 2 in subset 1 he checks that the square of the reply is vri 2 mod n. For those ri 2 in subset 2 he checks that the square of the reply is ri 2 mod n. Why does this work? • Finding square roots mod n is at least as hard as factoring n. This means that if you knew an easy way to find square roots mod n, you'd be able to factor n. And we all hope that factoring is difficult. • Suppose Fred wants to impersonate Alice. Anyone (including Fred) can compute squares mod n. Fred cannot take square roots mod n, but if he starts with a random r, he can compute r2 . Fred can give the correct answers for subset 2. But he cannot give the correct answers for subset 1, since he does not know s. So, what is the purpose of subset 2? Why isn't the protocol simply that Alice sends pairs ⟨ri 2, sri⟩? The problem with the simpler protocol is that once Alice sends a list of values to Bob, Bob can send the same values to Carol and successfully impersonate Alice. With the protocol as How to factor n if you can compute square roots mod n We'll assume n is odd and not the power of a prime. (If n is even, you can factor out all the factors of 2. If n is a power of a prime, you can try computing its kth root using ordinary arithmetic, and you'll only need to try for $k \leq \log_p n$ where p is the smallest prime you're not willing to try dividing into n directly.) Assume you have a method to compute square roots mod n. You choose a random x and compute s = x2 mod n.

Then you use your method of computing square roots mod n to compute the square root of s mod n, say y. This gives you two numbers, x and y, with the same square mod n. So (x + y)(x − y) = x2 − y2 = 0 mod n. If n has k distinct prime factors, then x2 has 2k square roots mod n (see §23.5 Chinese Remainder Theorem). So if n has at least 2 distinct prime factors, there is at least a 50% chance that y isn't x or −x mod n. In that case, neither x + y mod n nor x − y mod n is 0 mod n. And so the gcd of either of them with n must be a nontrivial factor of n. 6.8.1 ZERO KNOWLEDGE PROOF SYSTEMS 175 specified, the only information Bob can get is some numbers zi for which Alice tells him the square root of zi (those in subset 2), and some numbers zi for which Alice tells him the square root of vzi. He doesn't need Alice in order to find numbers for which he knows the square root—he can get such numbers himself by taking random numbers and squaring them. But he does need Alice for finding pairs ⟨ri 2 , sri⟩. However, for any ⟨ri 2 , sri⟩ he obtains from Alice rather than starting with ri and squaring it himself, he will not know ri —he will only know sri. So, assuming Fred has overheard Alice proving her identity to some people (maybe even Fred), Fred may have collected some values of ⟨ri 2, sri⟩. When Fred attempts to impersonate Alice he has a choice for each number of taking one of the values he has overheard from Alice, and for those he will be able to know the answer if they are selected to be in subset 1, or he can choose a random r and square it, and for those he will know the answer if they are selected to be in subset 2. But there will be no number for which he'll know both answers. That means there is a 50% probability, for each i, that Fred will be unlucky, and Carol will ask for the answer Fred does not know. If the protocol demands that Fred sends enough values (say 30), then the probability is overwhelming that his impersonation will be discovered. This scheme is much less work than RSA. Work for Alice is 45 modular multiplies (30 squarings plus an average 15 multiplies by s). Work for Bob is the same. By contrast, using RSA Alice must do a modular exponentiation with an average 768 modular multiplies, while Bob would get off easier with three (assuming a public exponent of 3). 6.8.1 Zero Knowledge Signatures Any zero knowledge system can be transformed into a public key signature scheme, though the performance in terms of bandwidth and CPU power usually makes the resulting scheme unattractive. Let's assume a typical sort of zero knowledge system. Alice has some sort of secret that enables her to transmit something and compute any answer to a question Bob might pose about that something, whereas an impostor can answer only one specific question Bob might pose. For example, in the case of the graph isomorphism scheme, Alice's secret is the mapping between graphs G1 and G2. The "something" that Alice transmits is a new graph, Gi. Bob's challenges are binary values. Only Alice can answer both 0 (show me the mapping from Gi to G1) and 1 (show me the mapping from Gi to G2). An impostor, say Trudy, can only answer one of the values (if she derived Gi from G1, she'll be lucky if Bob asks 0 but be unable to respond if Bob asks 1). In some other zero knowledge proof schemes Bob's challenge is a larger number (say 16 bits). Alice can answer any value Bob supplies, whereas impostor Trudy can only know a single value. So Trudy in that case would have only a 1 in 216 chance (per challenge) of being lucky enough for Bob to supply the question she can answer. 176 PUBLIC KEY ALGORITHMS 6.9 A signature scheme is not interactive. Bob cannot supply a challenge, or set of challenges. Instead, Alice has a message m that she wishes to sign. We're going to use a message digest function as a Bob surrogate. The message digest function will create a set of challenges that Alice cannot predict, and the fact that she can answer all the queries will reassure someone that Alice did produce the signature. Let's use Fiat-Shamir as an example. If Alice were proving her identity to Bob using Fiat-Shamir, her public key would be ⟨n,v⟩ and she'd transmit ri 2 mod n (for randomly chosen r1,...rk). Then Bob would send her k binary challenges. Now let's transform it into a signature scheme. Alice (and someone verifying Alice's signature) will take the message m, concatenated with the k values ri 2 mod n. The result is message

digested, and the resulting message digest is used as the surrogate Bob. For a 128-bit message digest, if k is 128, then each bit in the message digest corresponds to the challenge for the corresponding ri. The signature on m consists of the k values ri 2 mod n and the k responses to the calculated challenges. Why can't impostor Trudy forge a signature? She can choose any k values of ri, but until she chooses them, she cannot predict the generated challenge for each one. She can spend a lot of off-line searching for a set of ri that will happen to generate the challenges she can answer, so k probably needs to be a bit larger than it would be in an interactive zero knowledge proof. For instance, it might be acceptable for an impostor to have only a one in a billion chance of fooling an interactive Bob, but an impostor that was constructing a signature might be able to test a billion signatures for one that wound up asking the right questions. If there's a 1 in 264 chance, then a signature verifier can be reasonably certain that an impostor could not have been lucky enough, even with an off-line search, to find an appropriate set of ri. 6.9 HOMEWORK 1. In mod n arithmetic why does x have a multiplicative inverse if and only if x is relatively prime to n? 2. In section §6.4.2 Defenses Against Man-in-the-Middle Attack, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key? 3. In RSA, is it possible for more than one d to work with a given e, p, and q? 4. In RSA, given that the primes p and q are approximately the same size, approximately how big is $\phi(n)$ compared to n? 6.9 HOMEWORK 177 5. In DSS, other than saving users the trouble of calculating their own p, q, and g, why is there an efficiency gain if the value of p, q, and g are constant, determined in the specification? 6. What is the probability that a randomly chosen number would not be relatively prime to some particular RSA modulus n? What threat would finding such a number pose? 7. How would you modify ElGamal to operate with a smaller exponent like DSS? 8. Suppose Fred sees your RSA signature on m1 and on m2 (i.e. he sees m1 d mod n and m2 d mod n). How does he compute the signature on each of m1 j mod n (for positive integer j), m1 −1 mod n, m1·m2 mod n, and in general m1 j ·m2 k mod n (for arbitrary integers j and k)? 9. Suppose we have the encoding that enables Carol to mount the cube root attack (see §6.3.5.2 The Cube Root Problem). If Carol sends a message to Bob, supposedly signed by you, will there be anything suspicious and noticeable about the signed message, so that with very little additional computation Bob can detect the forgery? Is there anything Carol can do to make her messages less suspicious? 10. In ElGamal, how does knowing the secret number used for a signature reveal the signer's private key? How do two signatures using the same secret number reveal the signer's private key? [Hint: p−1 is twice a prime. Even though not all numbers have inverses mod p−1, division can still be performed if one is willing to accept two possible answers. (We're neglecting the case where the divisor is (p−1)/2, since it is extremely unlikely.)] 11. Transform the graph isomorphism scheme described in §6.8 Zero Knowledge Proof Systems into a signature scheme. Make an estimate of how large a signature would need to be. PART 2 AUTHENTICATION 179 7 OVERVIEW OF AUTHENTICATION SYSTEMS Authentication is the process of reliably verifying the identity of someone (or something). There are lots of examples of authentication in human interaction. People who know you can recognize you based on your appearance or voice. A guard might authenticate you by comparing you with the picture on your badge. A mail order company might accept as authentication the fact that you know the expiration date on your credit card. This chapter gives an overview of authentication systems, and subsequent chapters deal with the issues in more detail. There are two interesting cases. One case is when a computer is authenticating another computer, such as when a print spooler wants to authenticate a printer. The other case occurs when a person is using a public workstation, for instance a workstation installed in a public area (like a roomful of workstations set up for the convenience of attendees

at a convention). Such a workstation can perform sophisticated operations, but it will not store secrets for every possible user. Instead, the user's secret must be remembered by the user. A person is most likely to remember a secret if it is a text string (a password) and the user is allowed to choose it. Many users will not choose passwords wisely, and therefore the secret is a low-quality secret (one vulnerable to guessing). In this chapter we'll discuss various forms of authentication: password-based, address-based, and cryptographic. In subsequent chapters we cover authentication system concepts and standards in more depth. 7.1 PASSWORD-BASED AUTHENTICATION It's not who you know. It's what you know. Everyone knows what a password is, right? We thought so, too, until we tried to define it. When we use the phrase password-based authentication we are referring to a secret quantity (the password) that you state to prove you know it. The big problem with password-based authentication is eavesdropping. 180 OVERVIEW OF AUTHENTICATION SYSTEMS 7.1 There are certain types of systems that might be thought of as password-based that we are excluding from our definition. For instance, in some systems a user has a secret word or phrase that the user thinks of as a password, but the string known by the user is converted into a cryptographic key. We consider such a system a cryptographic rather than a password-based authentication system. Why does anyone use password-based authentication when cryptographic authentication is more secure? When dealing with people unaided by a workstation (e.g. the declining fraction of people at "dumb terminals"), it's difficult to avoid basing protocols on passwords, though as we will see (§8.8 Authentication Tokens), there are some clever devices being manufactured today that make cryptographic solutions compatible with human beings. Unfortunately, even computer-computer authentication is often based on passwords. Sometimes cryptography is not used because the protocol started out as a human-computer protocol and was not redesigned when its use got expanded to computer-computer communication. Sometimes it isn't used because the protocol designers assumed (perhaps correctly) that cryptography would be overly expensive in implementation time or processing resources. And sometimes cryptography is avoided because of legal issues. There are some cases in which it is really annoying that the designers opted for a simple password-based scheme. For instance, some older cellular phones transmit the telephone number of the phone and a password when making a call, and if the password corresponds to the telephone number, the phone company lets the call go through and bills that telephone number. The problem is, anyone can eavesdrop on cellular phone transmissions and clone such a phone, meaning they can make a phone that uses the overheard ⟨telephone number, password⟩ pair. Indeed, this is a problem—criminals do clone phones for stealing phone service and/or making untraceable calls. It would have been technologically easy to implement a simple cryptographic challenge-response protocol in the phone, and indeed the newer cell phones do this. There are other issues involved in using passwords. If you are using a workstation, accessing lots of resources across the network, it would be inconvenient for you to have to do a remote login, typing a name and password, every time you accessed a resource such as a file server. For a distributed system to be as convenient to use as a centralized one, you should only need a single password and should only need to enter it once per session. Your workstation could remember your name and password and transmit them on your behalf every time you accessed a remote resource, but this assumes you have the same password on every remote system. How could it be possible to have the same password on many systems? You could individually set your password to the same value on all the systems, but how would you manage to change your password and have the stored password information change simultaneously on all the systems? Alice I'm Alice, the password is fiddlesticks Bob 7.1.1 PASSWORD-BASED AUTHENTICATION 181 7.1.1 Off- vs. On-Line Password Guessing One way of guessing passwords is simply to type passwords at the system

that is going to verify the password. To thwart such an on-line attack, the system can make it impossible to guess too many passwords in this manner. For instance, ATM machines eat your card if you type three incorrect passwords. Alternatively, the system can be designed to be slow, so as not to allow very many guesses per unit time. Also, with an on-line attack, the system can become suspicious that someone might be attempting to break in, based on noticing an unusually large number of incorrect passwords. The system might then dispatch a human to investigate. In contrast, in an off-line attack, an intruder can capture a quantity X that is derived from a password in a known way. Then the intruder can, in complete privacy, use an arbitrary amount of compute power to guess passwords, convert them in the known way, and see if X is produced. Because a source of good password guesses is a dictionary, an off-line password guessing attack is sometimes referred to as a dictionary attack. If it is possible for an intruder to do off-line guessing, the secret must be chosen from a much larger space (see §8.3 Off-Line Password Guessing). 7.1.2 Storing User Passwords How does a server know Alice's password? There are several possibilities. 1. Alice's authentication information is individually configured into every server Alice will use. 2. One location, which we'll call an authentication storage node, stores Alice's information, and servers retrieve that information when they want to authenticate Alice. 3. One location, which we'll call an authentication facilitator node, stores Alice's information, and a server that wants to authenticate Alice sends the information received from Alice to the authentication facilitator node, which does the authentication and tells the server yes or no. In cases 2 and 3, it's important for the server to authenticate the authentication storage or facilitator node, since if the server were fooled into thinking a bad guy's node was the authentication storage or facilitator node, the server could be tricked into believing the wrong authentication information, and therefore let bad guys impersonate valid users. Regardless of where authentication information is stored, it is undesirable to have the database consist of unencrypted passwords because anyone who captured the database could impersonate all the users. Someone could capture the database by breaking into the node with the database, or by stealing a backup tape. In the first case above (authentication information individually configured into every server), capturing a server's database (of unencrypted passwords) would enable 182 OVERVIEW OF AUTHENTICATION SYSTEMS 7.1.2 someone to impersonate all the users of that server. Also, if a user had the same password on multiple servers, that user could then be impersonated at the other servers as well. In the second and third cases, many servers would use the one location, and capturing its database would enable someone to impersonate the users of all those servers. There's a trade-off, though. It might be difficult to physically protect every server, whereas if all the security information is in one location, it is only necessary to protect that one location. Put all your eggs in one basket, and then watch that basket very carefully. —Anonymous An alternative to storing unencrypted passwords is to store hashes of passwords, as is done in UNIX and VMS (see §8.3 Off-Line Password Guessing). Then if anyone were to read the password database they could do off-line password-guessing attacks, but would not be able to obtain passwords of users who chose passwords carefully. Alternatively, we could have the node that stores the password information encrypt the stored passwords (so that the server could decrypt a given password when needed). With hashed passwords an intruder who can read the hashed password database can do a password-guessing attack because the intruder will likely know the hash function (the function itself would not be secret). But with encrypted passwords the intruder can't get the passwords without knowledge of the node's key. Since the node would be a computer, the node's key would be a high-quality key, not derived from a password a human might be able to remember, and therefore invulnerable to guessing. Seemingly, then, encrypting (rather than hashing) the password database would be more secure, since an intruder would

not only have to be able to read the node's database of encrypted passwords, but also acquire the node's key. Encryption of the password database is indeed fairly secure for storage of the database on backup media, but it really isn't much protection against an intruder compromising the node itself, since the node's key is probably easily obtainable once inside the system. Most likely the node's key would be stored in some easily accessible place, readable by anyone with privileges (intruders always seem to obtain privileges). Worse yet, if the key isn't stored in nonvolatile memory somewhere, it likely would have to be typed in by the system manager every time the node was rebooted, in which case the key would most likely be handwritten and pasted to the keyboard. It is possible to combine both techniques by encrypting the database of hashed passwords. Then you get the security benefits of both. Sun Microsystems' NIS (Network Information Service), formerly known as YP (Yellow Pages), uses the directory service as an authentication storage node and stores the user's hashed password there. Like the original UNIX password file, the directory service information is world-readable, which means that anything can claim to be a server needing to see some user's authentication information. As generally deployed, the NIS directory service does not prove its identity to the server requesting the user's authentication information, so impersonation of the 7.2 ADDRESS-BASED AUTHENTICATION 183 directory service is somewhat of a security hole. NIS does make authentication across a network more convenient, but we suspect its availability will not convince NSA that it is safe to hook all its computers to the Internet. 7.2 ADDRESS-BASED AUTHENTICATION It's not what you know. It's where you are. Address-based authentication does not rely on sending passwords around the network, but rather assumes that the identity of the source can be inferred based on the network address from which packets arrive. It was adopted early in the evolution of computer networks by both UNIX and VMS. The basic idea is that each computer stores information which specifies accounts on other computers that should have access to its resources. For instance, suppose account name Smith on the machine residing at network address N is allowed access to computer C. Requests for resources are commands like copy a specified file, log in, or execute the following command at the specified remote machine. If a request arrives from address N claiming to be sent on behalf of user Smith, then C will honor the request. On UNIX, the Berkeley rtools support such access; on VMS, similar functionality is called PROXY. The general idea can be implemented in various ways. • Machine B might have a list of network addresses of "equivalent" machines. If machine A is listed, then any account name on A is equivalent to the same account name on B. If a request from A arrives with the name JohnSmith, then the request will be honored if it is for anything that the account JohnSmith on B was allowed to do. This has the problem that the user has to have the identical account name on all systems. • Machine B might instead have a list of ⟨address, remote account name, local account name⟩. If a request arrives from address A with the name Jekyll, then the database is scanned for the matching entry, say ⟨A, Jekyll, Hyde⟩. Then the request is honored provided the local account Hyde is authorized to do the request. UNIX implements two account mapping schemes: • /etc/hosts.equiv file. There is a global file (named /etc/hosts.equiv) which implements the first scheme above. The file /etc/hosts.equiv on machine A contains a list of computers that have identical user account assignments. Suppose a computer B is listed in /etc/hosts.equiv. Suppose A receives a request with account name Smith, and B's address in the source address field of the network header. If an account with the name Smith exists on machine A, the 184 OVERVIEW OF AUTHENTICATION SYSTEMS 7.2 request will be given the same privileges as the local user Smith. (Actually, an exception is made for the privileged account root; it will not be given access by virtue of an entry in /etc/ hosts.equiv). The /etc/hosts.equiv file is useful for managing corresponding accounts in bulk on machines with common accounts and common management. • Per-user .rhosts files. In each

UNIX user's home directory, there can be a file named .rhosts, which contains a list of ⟨computer, account⟩ pairs that are allowed access to the user's account. Any user Bob can permit remote access to his account by creating a .rhosts file in his home directory. The account names need not be the same on the remote machine as they are on this one, so Bob can handle the case where he has different account names on different systems. Because of the way the information is organized, any request that is not for an account named the same as the source account must include the name of the account that should process the request. For instance, if the local account name on system A is Bob and the request is from computer B, account name Smith, the request has to specify that it would like to be treated with the privileges given account name Bob. (See Homework Problem 1.) On VMS, individual users are not permitted to establish their own proxy access files. (This is considered a security feature to prevent users from giving access to their friends). Instead, there is a centrally managed proxy database that says for each remote ⟨computer, account⟩ pair what account(s) that pair may access, usually with one of them marked as the default. For example, there might be an entry specifying that account Smith from address B should have access to local accounts Bob and Alice, where the account Bob might be marked as the default. The VMS scheme makes access somewhat more user-friendly than the UNIX scheme in the case where a user has different account names on different systems. Generally (in VMS) the user need not specify the target account in that case. In the rare case where a user is authorized to access multiple accounts on the remote computer, a target account can be specified in the request to access an account other than the default. (In UNIX, it is always necessary to specify the target account in the request if the account name is different.) Address-based authentication is safe from eavesdropping, but is subject to two other threats: • If someone, say Trudy, gains privilege on a node FOO, she can access all users' resources of FOO—there's nothing authentication can do about that. But in addition, she can access the network resources of any user with an account on FOO by getting FOO to claim the request comes from that user. Theoretically, it is not obvious how Trudy, once she gains access to FOO, would know which other machines she could invade, since the ⟨machine, account⟩ pairs reachable from ⟨FOO, Smith⟩ are not listed at FOO, but are instead listed at the remote machine. However, in a lot of cases, if ⟨FOO, Smith⟩ allows proxy access from ⟨BAR, JohnS⟩, then it is likely that node BAR specifically allows account Smith at node FOO access to account JohnS at BAR. So Trudy, once she has privileges on FOO, can scan the proxy database (in the case of VMS) or the /etc/hosts.equiv file and the .rhosts files of each 7.2.1 ADDRESS-BASED AUTHENTICATION 185 of the users (in the case of UNIX), and make a guess that any ⟨node, account name⟩ pairs she finds are likely to let her in from node FOO with the specified account name. • If someone on the network, say Trudy, can impersonate network addresses of nodes, she can access all the network resources of all users who have accounts on any of those nodes. It is often relatively easy to impersonate a network address. For instance, on broadcast LANs (like Ethernet and Token Ring), it is easy not only to send traffic with the address of a different node on that LAN, but also to receive traffic destined for that node. In many other cases, although it is easy to transmit a packet that has a false source address, the returning traffic would be delivered to the real node rather than the impersonating node. Depending on the environment, therefore, address-based authentication may be more or less secure than sending passwords in the clear. It is unquestionably more convenient and is the authentication mechanism of choice in many distributed systems deployed today. 7.2.1 Network Address Impersonation The wire protocol guys don't worry about security because that's really a network protocol problem. The network protocol guys don't worry about it because, really, it's an application problem. The application guys don't worry about it because, after all, they can just use the IP address and trust the network. —Marcus

Ranum How can Trudy impersonate Alice's network address? Generally it is easy to transmit a packet claiming any address as the source address, either at the network layer or the data link layer. Sometimes it is more difficult. For instance, due to the design of token rings, if Trudy and Alice were on the same ring and Trudy were to transmit a packet using Alice's data link address, then Alice might remove Trudy's packet from the ring or raise a duplicate address error. Another example is star-topology LANs, where each node is connected via a point-to-point link to a central hub. A hub might learn (or be configured with) the data link address of the node on each link, and refuse to forward a packet if the data link source address is not correct for that line, though a hub (being a layer 2 device) is unlikely to check the network layer (layer 3) source address. A router could also have features to make it difficult for someone to claim to be a different network address. If a router has a point-to-point link to an endnode, it could (like the hub) refuse to accept a packet if the network layer source address is not correct according to the router's configuration. A router could also be configured, on a per-link basis, with a set of addresses that it should expect to appear on each link, and refuse packets with unexpected source addresses. Perhaps if the routers were all trusted, and authentication were added to routing messages to prevent someone 186 OVERVIEW OF AUTHENTICATION SYSTEMS 7.3 from injecting bad routing messages, then a router would not need to be configured on a per-link basis with a set of expected source addresses, but instead could derive the per-link expected source addresses from the routing database. It is often more difficult for Trudy to receive messages addressed to Alice's network address than for Trudy to claim Alice's address as the source address. If Trudy is on the same LAN as Alice, it is trivial—Trudy just needs to listen to packets addressed to Alice's data link address. If Trudy is not on the same LAN as Alice, but is on one of the LANs in the path between the source and Alice, it's a little harder but still easy. The reason it's a little harder is that Trudy would need to listen to all packets address to the data link address of the router that forwards packets towards Alice from the LAN on which Trudy resides. This is harder than listening to Alice's data link address because a router will probably be receiving packets for many destinations, and Trudy will have to be fast enough to ignore the messages that aren't specifically for Alice. If Trudy is not on the same LAN as Alice, and not on the path between the source and Alice, then it is more difficult for Trudy to receive packets addressed to Alice. Trudy could potentially inject routing messages that would trick the routers into sending traffic for Alice's address to Trudy. It is possible to add a cryptographic authentication mechanism to router messages which would prevent Trudy from injecting routing information. Even with cryptographic authentication of routers, it may be possible for Trudy to be able to both transmit from and receive packets for Alice's network address. For instance, in IP, there is a feature known as source routing, in which Trudy can inject an IP packet containing not just a source address (Alice's network address) and destination address D, but also a source route that gives intermediate destinations. IP will route to each intermediate destination in turn. Assume Trudy uses the source route ⟨Alice, Trudy, D⟩. The applications that run on IP that do address-based authentication will use the field Alice for the address check. When D receives a packet from Alice via source route ⟨Alice, Trudy, D⟩, then D, following the IP host requirements document (RFC 1122), will reply with source route ⟨D, Trudy, Alice⟩. The return traffic will go to Trudy. 7.3 CRYPTOGRAPHIC AUTHENTICATION PROTOCOLS It's not what you know or where you are, it's %zPy#bRw Lq(ePAoa&N5nPk9W7Q2EfjaP!yDB$S Cryptographic authentication protocols can be much more secure than either password-based or address-based authentication. The basic idea is that Alice proves her identity to Bob by performing a cryptographic operation on a quantity Bob supplies. The cryptographic operation performed by Alice is based on Alice's secret. We talked in Chapter 2 Introduction to Cryptography about how 7.4 WHO IS BEING AUTHENTICATED? 187 hashes, secret key

cryptography, and public key cryptography could all be used for authentication. The remainder of this chapter discusses some of the more subtle aspects of making these protocols work. 7.4 WHO IS BEING AUTHENTICATED? Suppose user Bob is at a workstation and wants to access his files at a file server. The purpose of the authentication exchange between the workstation and the file server is to prove Bob's identity. The file server generally does not care which workstation Bob is using. There are other times when a machine is acting autonomously. For instance, directory service replicas might coordinate updates among themselves and need to prove their identity to each other. Sometimes it might be important to authenticate both the user and the machine from which the user is communicating. For example, a bank teller might be authorized to do some transactions, but only from the terminal at the bank where that teller is employed. What's the difference between authenticating a human and authenticating a computer? A computer can store a high-quality secret, such as a long random-looking number, and it can do cryptographic operations. A workstation can do cryptographic operations on behalf of the user, but the system has to be designed so that all the person has to remember is a password. The password can be used to acquire a cryptographic key in various ways: • directly, as in doing a hash of the password • using the password to decrypt a higher-quality key, such as an RSA private key, that is stored in some place like a directory service 7.5 PASSWORDS AS CRYPTOGRAPHIC KEYS Cryptographic keys, especially public key based cryptographic keys, are specially chosen very large numbers. A normal person would not be able to remember such a quantity. But a person can remember a password. It is possible to convert a text string memorizable by a human into a cryptographic key. For instance, to form a DES secret key, a possible transformation of the user's password is to do a cryptographic hash of the password and take 56 bits of the result. It is much more tricky (and computationally expensive) to convert a password into something like an RSA private key, since an RSA key has to be a carefully constructed number. 188 OVERVIEW OF AUTHENTICATION SYSTEMS 7.6 Jeff Schiller proposed a method of converting a password into an RSA key pair that has plausible efficiency. The basic idea is to convert the user's password into a seed for a random number generator. Choosing an RSA key pair involves testing a lot of randomly chosen numbers for primality to obtain two primes. And the primality tests involve choosing random numbers with which to test the randomly chosen primes. If the RSA key pair generator is run twice, and if for each run the sequence of the random numbers selected by the random number generator is the same, the RSA key pair generator will find the same RSA key pair. So if the user's password is the seed for the random number generator, the RSA key pair generator will always generate the same RSA key pair for the user. Running the RSA key pair generator from the beginning, seeded with the user's password, is likely to take an unacceptable amount of computation, since typically the RSA key generator will find many non-primes before it finally finds two primes. If it had to be done every time the user logged in, it would be prohibitively expensive. But after the workstation successfully finds the RSA key pair derived from the user's password, it can give the user something to remember that would enable it to complete its work more quickly. For example, if (starting with the user's password as seed) the first prime was found after 857 guesses and the second after 533 guesses, then the user might be told to remember ⟨857, 533⟩. It is an interesting problem to come up with a scheme that minimizes the amount of information the user needs to remember while still significantly lowering the work necessary for the computer. Note that since the ⟨857, 533⟩ quantity is not particularly security-sensitive, it could be posted on the user's workstation without significant loss of security. Schemes based on this idea (direct conversion of the user's password to a public key pair) are not used because they perform poorly and because knowledge of the public key alone gives enough information for an off-line password-guessing attack. So the usual practice when using public keys with humans is

to encrypt the user's private key with the user's password and store it somewhere (e.g., the directory service) from which it can be retrieved by the user's workstation and decrypted. In order for Alice's workstation, on behalf of Alice, to prove Alice's identity, it must first retrieve her encrypted private key from the directory service. Then it takes the password as typed by Alice, converts it into a secret key, and uses that secret key to decrypt the encrypted private key. There are clever schemes for doing this described in §10.4 Strong Password Credentials Download Protocols. 7.6 EAVESDROPPING AND SERVER DATABASE READING Public key technology makes it easy to do authentication in a way that is both secure from eavesdropping and from an intruder reading the server database. Alice knows her own private key. Bob stores Alice's public key. An intruder who reads Bob's database (and therefore obtains Alice's public key) will not be able to use the information to impersonate Alice. Authentication is done by 7.6 EAVESDROPPING AND SERVER DATABASE READING 189 Alice using her private key to perform a cryptographic operation on a value Bob supplies, and then transmitting the result to Bob. Bob checks the result using Alice's public key. Without public key cryptography, it's difficult to protect against both eavesdropping and server database reading with a single protocol, although it is easy to do one or the other. For instance, let's try a protocol such as is used when a person logs into a system such as UNIX. Bob (the computer authenticating the user Alice) stores a hash of Alice's password. Let's say Alice's password is fiddlesticks. In this protocol someone, say Trudy, who accessed Bob's database would not be able to impersonate Alice. She could do an off-line password-guessing attack if she were to obtain Alice's hashed password from Bob's database, but that would not help her if Alice had chosen a good password. But if Trudy were to eavesdrop when Alice was proving her identity to Bob, Trudy would obtain Alice's password. So this protocol is secure against server database disclosure but not against eavesdropping. Now consider another protocol (see below). Let's assume that Bob stores Alice's actual secret. In this case Trudy, if she were to eavesdrop, would not be able to obtain information to allow her to impersonate Alice (except by an off-line password-guessing attack). But if she were to read Bob's database, she'd obtain Alice's secret. We'd almost claim it was impossible to simultaneously protect against eavesdropping and server database disclosure without using public key cryptography. However, an elegant protocol invented by Leslie Lamport can claim to accomplish this feat. Lamport's scheme does have a serious drawback, though. A user can only be authenticated a small finite number of times (such as 1000) before it is necessary to reinstall the user's information at the Alice I'm Alice Bob knows Alice's public key picks random R R R signed with Alice's private key checks result using Alice's public key Alice Alice, fiddlesticks Bob knows hash of Alice's password computes hash of fiddlesticks compares it with stored value 190 OVERVIEW OF AUTHENTICATION SYSTEMS 7.7 server; that is, someone with privileges at the server and knowledge of the user's password must reconfigure the server with the user's security information. Phil Karn has implemented Lamport's scheme in his S/Key software, and it is being used in the Internet. Lamport's scheme is described in §10.2 Lamport's Hash. 7.7 TRUSTED INTERMEDIARIES Assume that network security is based on secret key technology. If the network is fairly large, say n nodes, and each computer might need to authenticate each other computer, then each computer would need to know n−1 keys, one for each other system on the network. If a new node were added to the network, then n keys would need to be generated, enough for that new node to have a shared secret with each of the other nodes. Somehow the keys would have to be securely distributed to all the other nodes in the network. This is clearly unworkable in any but very small networks. Alice I'm Alice Bob knows Alice's secret picks random R R computes X = cryptographic function of her secret and R X computes same function and compares it to X α γ ε ζ η θ δ β 7.7.1 TRUSTED INTERMEDIARIES 191 7.7.1 KDCs One way to make things manageable is to use a

trusted node known as a Key Distribution Center (KDC). The KDC knows keys for all the nodes. If a new node is installed in the network, only that new node and the KDC need to be configured with a key for that node. If node α wants to talk to node β, α talks to the KDC (securely, since α and the KDC share a key), and asks for a key with which to talk to β. The KDC authenticates α, chooses a random number Rαβ to be used as a key to be shared by α and β for their conversation, encrypts Rαβ with the key the KDC shares with α and gives that to α. The KDC also encrypts Rαβ with the key the KDC shares with β and gives that to β, with the instruction that it is to be used for conversing with α. (Usually, the KDC will not bother to actually transmit the encrypted Rαβ to β but rather will give it to α to forward to β.) The encrypted message to β that the KDC gives to α to forward is often referred to as a ticket. Besides containing Rαβ, the ticket generally contains other information such as an expiration time and α's name. We'll discuss protocols involving KDCs in §9.4 Mediated Authentication (with KDC). KDCs make key distribution much more convenient. When a new user is being installed into the network, or when a user's key is suspected of having been compromised, there's a single location (the KDC) that needs to be configured. The alternative to using a KDC is installing the user's information at every server to which the user might need access. There are some disadvantages to KDCs, though: • The KDC has enough information to impersonate anyone to anyone. If it is compromised, all the network resources are vulnerable. • The KDC is a single point of failure. If it goes down, nobody can use anything on the network (or rather, nobody can start using something on the network—keys previously distributed can continue to be used). It is possible to have multiple KDCs which share the same database of keys, but that means added complexity and cost for extra machines and replication protocols, and added vulnerability, since there are now more targets that need to be protected. • The KDC might be a performance bottleneck, since everyone will need to frequently communicate with it. Having multiple KDCs can alleviate this problem. KDC α β γ δ ε ζ η θ ι 192 OVERVIEW OF AUTHENTICATION SYSTEMS 7.7.2 7.7.2 Certification Authorities (CAs) Key distribution is easier with public key cryptography. Each node is responsible for knowing its own private key, and all the public keys can be accessible in one place. But there are problems with public keys as well. If, for instance, all the public keys are published in The New York Times, or stored in the directory service, how can you be sure that the information is correct? An intruder, Trudy, might have overwritten the information in the directory service or taken out her own ad in The New York Times. If Trudy can trick you into mistaking her public key for Alice's, she can impersonate Alice to you. The typical solution for this is to have a trusted node known as a Certification Authority (CA) that generates certificates, which are signed messages specifying a name (Alice) and the corresponding public key. All nodes will need to be preconfigured with the CA's public key so that they can verify its signature on certificates, but that is the only public key they'll need to know a priori. Certificates can be stored in any convenient location, such as the directory service, or each node can store its own certificate and furnish it as part of the authentication exchange. CAs are the public key equivalent of KDCs. A CA or a KDC is the single trusted entity whose compromise can destroy the integrity of the entire network. The advantages of CAs over KDCs are: • The CA does not need to be on-line. It might be in a locked room protected by a scary-looking guard. Perhaps only a single very trusted individual has access to the CA. That person types the relevant information at the CA, and the CA writes a floppy disk with the new user's certificate, and the floppy disk can be hand-carried to a machine that's on the network. If the CA is not on-line it cannot be probed by curious intruders. • Since the CA does not have to be on-line or perform network protocols, it can be a vastly simpler device, and therefore it might be more secure. • If the CA were to crash, the network would not be disabled (as would be the case with a KDC). The only operation that would be impacted is installing new users (until things start expiring, such

as certificates or Certificate Revocation Lists—see §7.7.3 Certificate Revocation). So it's not as essential to have multiple CAs. • Certificates are not security-sensitive. If they are stored in a convenient, but potentially insecure, location like the directory service, a saboteur might delete certificates, which might prevent network access by the owners of those certificates, but the saboteur cannot write bogus certificates or modify certificates in any way, since only the CA can generate signatures. • A compromised CA cannot decrypt conversations, whereas a compromised KDC that sees a conversation between any two parties it serves can decrypt the conversation. A compromised CA can fool Alice into accepting an incorrect public key for Bob, and then the CA can impersonate Bob to Alice, but it will not be able to decrypt a conversation between the real Alice 7.7.3 TRUSTED INTERMEDIARIES 193 and the real Bob. (It's still really bad for a CA to be compromised, but we're just saying it's not quite as bad as compromise of a KDC.) Why do you security people always speak of compromise as if it's a bad thing? Good engineering is all about compromise. —overheard at a project review 7.7.3 Certificate Revocation There is a potential disadvantage with CAs. Suppose Fred is given a certificate with an expiration time a year in the future, and then Fred is fired. Since Fred is now a disgruntled ex-employee, it would be nice to alert the network not to allow him access. With KDCs it's easy—merely delete his key from the KDC. With CAs, though, it's not as straightforward to deny access to someone once he is given a certificate. It is common practice to put an expiration date in a certificate. The certificate becomes invalid after that date. The typical validity interval is about a year. A disgruntled ex-employee can do a lot of damage in a year, even without a machine gun. But you wouldn't want validity intervals much smaller than that, because renewing certificates is a nuisance. The solution is similar to what was done for credit cards. When the bank issues a credit card, it prints an expiration date, perhaps a year in the future. But sometimes a card is reported stolen, or the bank might for some other reason like to revoke it. The credit card company publishes a book of credit card numbers that stores should refuse to honor. These days, most stores are hooked to computer networks where they check the validity of the card, but in ancient times merchants needed to rely on the book of bad credit card numbers, which was presumably published frequently. X.509 [ISO97] has a defined format for a certificate, as well as of a Certificate Revocation List (CRL). We document the formats in §13.6 PKIX and X.509. A CRL lists serial numbers of certificates that should not be honored. A new CRL is posted periodically, and lists all revoked and unexpired certificates. A certificate is valid if it has a valid CA signature, has not expired, and is not listed in the CA's most recent CRL. The CRL has an issue time. If an application wants to ensure that none of the certificates it honors has been revoked in the last hour, say, then the application will need to see a CRL issued within the last hour before it will honor any certificates. This means, in this case, that a new CRL must be posted at least once an hour. An intruder might delete the latest CRL, in which case applications that want to see a CRL posted within the last hour will refuse to honor any certificates, but the intruder cannot impersonate a valid user by destroying the CRL or overwriting it with a CRL issued longer ago than the application's tolerance. A certificate includes the user's name, the user's public key, an expiration time, a serial number, and the issuing CA's signature on the entire contents of the certificate. An X.509 CRL includes a list of serial numbers of unexpired revoked certificates and an issue time for the CRL. 194 OVERVIEW OF AUTHENTICATION SYSTEMS 7.7.4 Suppose Bob is an application that needs to authenticate the user Alice. Bob needs Alice's certificate and a recent CRL. Bob can obtain them from the directory service, or perhaps Alice transmits them to Bob. Armed with the certificate and CRL, Bob decides what public key is associated with the name Alice. If the certificate is validly signed by the CA and has not expired, and the CRL is validly signed by the CA and is sufficiently recent and does not list Alice's certificate, then Bob will assume that Alice's public key is as stated in

the certificate. Then Bob will go through an authentication handshake whereby Alice proves she knows the private key that corresponds to the public key listed in her certificate. Revocation schemes are discussed in more detail in §13.4 Revocation. 7.7.4 Multiple Trusted Intermediaries A problem with both KDCs and CAs as described so far is that they require that there be a single administration trusted by all principals in the system. Anyone who compromises the KDC or the CA can impersonate anyone to anyone. As you try to scale authentication schemes up to international or intercorporate scale, you discover that there is no one who everyone trusts (and if there were, they would be too busy with more important tasks to operate and manage a KDC or CA). The solution is to break the world into domains. Each domain has one trusted administration (one logical KDC or one CA—there might be replicas for availability, but they are all are functionally equivalent). If Alice and Boris are in the same domain, they authenticate as described above. If they are in different domains, then authentication is still possible, but a little more complicated. 7.7.4.1 Multiple KDC Domains How does key distribution work with multiple KDC domains? Let's say Alice is in the CIA and Boris is in the KGB. The CIA will manage a KDC, and the CIA's KDC will know Alice's key. Boris's key is known to the KGB's KDC. Given that the two organizations want to be able to exchange secure electronic mail (perhaps so they can express outrage that the other organization is CIA's KDC KGB's KDC Alice KKGB-CIA Boris 7.7.4.1 TRUSTED INTERMEDIARIES 195 engaged in, gasp, spying), they can make this possible by having a key that the two KDCs share. So the CIA's KDC, in addition to having keys for all its users, will also have configured a key which it shares with the KGB's KDC. Let's call that shared key KKGB-CIA. KKGB-CIA is used when a user in the CIA wants to have a secure communication with a user in the KGB. Alice, knowing Boris is in the KGB, tells her KDC that she wants to talk to the KGB's KDC. Her KDC facilitates this the same way it would facilitate communication between two CIA users. It generates a new random key, Knew, and encrypts a message containing Knew using Alice's key. This message is to inform Alice of Knew. It separately encrypts a message containing Knew using the key KKGB-CIA. That message, which also contains Alice's name, will be sent to the KGB's KDC. The fact that the message is encrypted with KKGB-CIA lets the KGB's KDC know that Alice is from the CIA's domain. (Note that the KGB's KDC will probably have many KDCs with which it shares keys. For performance reasons, a cleartext clue that the message is coming from the CIA's KDC will have to be sent along with the encrypted message so that the KGB's KDC will know to try the key KKGB-CIA.) After verifying that the message is encrypted with KKGB-CIA, the KGB's KDC will generate a key KAlice-Boris and send that to Alice, encrypted with Knew. It will also generate a message for Boris (which it gives to Alice to deliver) that will be encrypted with Boris's key. The message will contain KAlice-Boris and the information that Alice is from the CIA domain. There would likely be thousands of domains in an international/intercorporate/interdenominational internetwork. The conceptually simplest method of allowing users from one domain to talk to users in another domain securely is to have every pair of KDCs configured with a shared key. So the CIA KDC would have a shared key, not only with the KGB's KDC, but with Greenpeace's KDC, and MIT's KDC, and IBM's KDC. Perhaps someone (either in Greenpeace or the CIA) might decide that there was no reason for traffic between the two domains, in which case those KDCs would not need to share a key. But most likely there would be so many domains that it would Alice let me talk to KGB's KDC CIA's KDC KGB's KDC Boris CIA's KDC generates Knew KAlice{use key Knew} KCIA-KGB{Alice from my domain wants to talk to you; use Knew} let me talk to Boris KDC generates KAlice-Boris Knew{use KAlice-Boris} KBoris{talk to Alice from CIA; use KAlice-Boris} 196 OVERVIEW OF AUTHENTICATION SYSTEMS 7.7.4.2 be unworkable to have every pair of domains configured with a shared key. So there would likely be either a tree of KDCs, or some perhaps less structured logical interconnection of KDCs. Users can securely

authenticate even if their KDCs are not directly linked, if a chain of KDCs can be found. It isn't obvious how one would find an appropriate chain of KDCs. For instance, as we will discuss in the Kerberos chapters, Kerberos V4 does not allow chains of KDCs; to have interdomain communication between two KDCs, they have to have a shared key. Kerberos V5 allows arbitrary connectivity, but assumes there is a default hierarchy, with perhaps additional links (shared keys) between pairs of KDCs that are not directly connected in the default hierarchy. After Alice has negotiated a chain of KDCs to get to Boris's KDC, the encrypted message Boris receives should list the chain of KDCs that helped establish the path. The reason for this is that Boris might trust some chains more than others. For instance, the KGB's KDC might have a shared key with MIT's KDC. That KDC, if overtaken by playful undergraduates, could send a message to the KGB's KDC that Alice, from the CIA, wants to talk to the KGB's KDC, through the path CIA–MIT. It is likely that the KGB's KDC would not trust anything more than one hop through MIT; that is, it will believe MIT's KDC when it tells it there are users from MIT who wish to communicate, but it won't believe it if MIT claims a different organization is attempting communication with the KGB through MIT's KDC. 7.7.4.2 Multiple CA Domains The situation is similar with CAs. Each CA services a set of users, and issues certificates for those users. This is functionally similar to a KDC having a shared key with a user. The users of a particular CA can verify each other's certificates, since all users of a particular CA know its public key. How can Alice be sure she knows Boris's public key if Alice's CA is different from Boris's CA? This is accomplished by having the two CAs issue certificates for each other. Alice obtains a KDC1 KDC2 KDC3 KDC4 KDC5 KDC6 KDC KDC KDC KDC KDC KDC KDC KDC KDC 7.8 SESSION KEY ESTABLISHMENT 197 certificate, signed by her own CA, stating the public key of Boris's CA. Then, using that certificate plus Boris's certificate (which has been signed by Boris's CA), she can verify Boris's public key. • Alice obtains Boris's CA's certificate stating that its public key is P1, signed by her own CA. • Alice obtains Boris's certificate stating that his public key is P2, signed with key P1. Because she has both of those certificates, she now can verify Boris's public key. As with chains of KDCs, it is possible that some CAs will not have certificates for each other, but there will be a chain of CAs which works. In Chapter 13 PKI (Public Key Infrastructure), we discuss various strategies for finding appropriate chains of certificates. It is really no different from the problem of navigating through logically interconnected KDCs, and there is really no universally accepted wonderful answer, though there are several workable schemes. 7.8 SESSION KEY ESTABLISHMENT If all that was done about computer network security was to replace all the cleartext password exchanges with cryptographic authentication, computer networks would be a lot more secure than they are today. But there are security vulnerabilities that occur after authentication. An eavesdropper might steal information by seeing your conversation. Something along the path (e.g., a router) might intercept messages in transit and modify them or replay them. Or someone on the path might "hijack" the entire session by impersonating the network address of one of the parties after the initial authentication is complete. We can protect against eavesdroppers, session hijackers, and message manglers by using cryptography throughout the conversation. But cryptography, of course, involves keys. What should we use for a key? One possibility is that our authentication exchange is based on public keys. In theory, in a conversation between Alice and Bob, Alice could encrypt all of the data she sends to Bob with Bob's public key (for privacy), and sign each message with her own private key (for integrity). Bob could similarly encrypt all the data he sends to Alice with Alice's public key, and sign the data with his private key. This isn't generally done because public key operations are computationally expensive. It is far more practical to have Alice and Bob somehow agree on a secret key, and protect the remainder of the conversation with the secret key. It would be nice if the authentication protocol helped

them agree on a secret key, in addition to having Alice and Bob authenticate each other. When Alice and Bob are using a shared secret key to authenticate each other they could continue to use that key to protect their conversation. However, it is a good idea to generate a separate session key: 198 OVERVIEW OF AUTHENTICATION SYSTEMS 7.9 • Keys sort of “wear out” if used a lot. If you’re an intruder trying to find a key, the more encrypted data you have the more likely you’ll succeed. Since establishing a shared authentication key is usually an expensive operation involving out-of-band mechanisms, it is desirable that the authentication key be used only for the initial exchange. A secret per-session key can be generated at the time of authentication and used for integrity protection and encryption for the remainder of the session. • If Alice and Bob were to use the same secret key for authentication as for integrity-protecting their data, it might be possible for an intruder to record messages from a previous Alice-Bob conversation and inject those packets into a current Alice-Bob conversation, tricking them into thinking the messages were part of the current session. Within a session there is generally a sequence number to prevent an intruder from adding, deleting, or reordering messages. But there might be nothing in the packet to distinguish packets transmitted a month ago from packets transmitted today. For instance, the transport protocol might always start with sequence number 1. If the authentication protocol agreed upon a new session key for each conversation, then replayed messages from previous conversations would not be accepted as valid. • If the long-term shared secret key is compromised, it would be nice to prevent an old recorded conversation from being decryptable. If every conversation is encrypted with a different per-session key, then a suitable authentication protocol can prevent old conversations from being decrypted. • You may want to establish a session and give a relatively untrusted piece of software the session key, which is good only for that conversation, rather than giving it your long-term secret key, which the untrusted software could store away for future use. For these reasons, authentication protocols usually establish a session key, in addition to providing authentication. We’ll describe the subtleties of authentication protocols in Chapter 9 Security Handshake Pitfalls. 7.9 DELEGATION It’s not who you are. It’s who you’re working for. Sometimes it is necessary to have things act on your behalf. For instance, you might be logged into system Bob, and then need to access remote files from Bob. In that case, Bob will need to retrieve the files on your behalf, and will need to have the same privileges for those files as you have. Simi- 7.9 DELEGATION 199 larly you might tell a printserver to print a remote file that you are authorized to read. You could allow the required access by explicitly being available to log in to each resource that is ever needed on your behalf. That would be inconvenient at best, and sometimes impossible, for instance when you run a batch job when you aren’t around. One possible means of allowing access is to give your password to everything that might need to act on your behalf (printservers, foreign systems). This might be reasonable if you changed your password immediately after the work was completed. But most users would not change their password that frequently. Another possibility is that you could explicitly add every system to every ACL for every resource that they need to access on your behalf, and then, if you are conscientious, delete them from the ACL when the operation completes. These mechanisms are far too inconvenient, and most users would opt for no security at all if that was the alternative. Therefore it is advisable to have some more convenient mechanism for giving something permission to act on your behalf. This permission is known as delegation or authentication forwarding. The best mechanism for delegation in a computer network is to generate a special message, signed by you, specifying to whom you are delegating rights, which rights are being delegated, and for how long. Once the duration specified in the message has expired, the message no longer grants any permissions. Using public key cryptography, the usual approach is for you to sign the delegation message with your own

private key. Using KDCs, the usual approach is for you to ask the KDC to give Fred certain permissions. The KDC constructs a message listing the permissions you'd like to grant Fred, encrypts the message with a key known only to the KDC, and Fred is given the encrypted message. Fred cannot decrypt the message, but he can present it to the KDC when asking for access; the KDC can decrypt it and then see the rights that have been delegated to Fred. Why is it necessary for the delegation message to expire? If you are delegating rights to a printserver and it can be trusted for the hour it takes to finish printing all the documents ahead of yours, and then print your document, can't it always be trusted? If it were untrustworthy, surely it could have retrieved all your documents during the one hour in which you gave it access rights to your files. In theory this argument is true. You can't necessarily know that the printserver hasn't had some Trojan horse software installed which will steal all your files during the limited time you give it access, but limiting the delegation in time limits the window of vulnerability. You can further limit the vulnerability by limiting the scope of the delegation. Instead of specifying that the printserver has rights to see all your files, you might specify only the single file that you'd like printed. The need for delegation in some form is generally recognized, and various systems implement crude forms of it. However, designing a user interface and corresponding data structure where a user specifies what rights are delegated and to whom is an open research problem. 200 OVERVIEW OF AUTHENTICATION SYSTEMS 7.10 7.10 HOMEWORK 1. As stated in §7.2 Address-Based Authentication, UNIX requires a request from system A for a resource at B to explicitly state the desired account name at B if it is not identical to the account name at A. Why do you suppose it makes that requirement? How would one implement this feature without the requirement? 2. In §7.6 Eavesdropping and Server Database Reading we asserted that it is extremely difficult, without public key cryptography, to have an authentication scheme which protects against both eavesdropping and server database disclosure. Consider the following authentication protocol (which is based on Novell version 3 security). Alice knows a password. Bob, a server that will authenticate Alice, stores a hash of Alice's password. Alice types her password (say fiddlesticks) to her workstation. The following exchange takes place: Is this an example of an authentication scheme that isn't based on public key cryptography and yet guards against both eavesdropping and server database disclosure? 3. Extend the scenario in §7.7.4.1 Multiple KDC Domains to a chain of three KDCs. In other words, assume that Alice wants to talk to Boris through a chain of three KDCs (Alice's KDC, a KDC that has shared keys with both Alice's KDC and Boris's KDC, and finally, Boris's KDC). Give the sequence of events necessary to establish communication. Alice Alice,fiddlesticks Workstation Alice Bob knows Z = hash of Alice's password computes Y = hash of fiddlesticks R picks random R hash(Y,R) compares hash(Z,R) with received value 201 8 AUTHENTICATION OF PEOPLE In the previous chapter we discussed in general how a computer authenticates a computer across a network. This chapter deals with the special issues involved when a computer is authenticating a human. We use the terms user and human interchangeably. This chapter deals with password-related issues like how to force users to choose unguessable passwords, how to store password information securely at the system being logged into, and how to avoid divulging information to eavesdroppers. Authentication is done somewhat differently depending on the capabilities of the thing being authenticated. The two most important capabilities are the ability to store a high-quality cryptographic key and the ability to perform cryptographic operations. (A high-quality key is a secret chosen from a very large space so that it is computationally infeasible to guess the secret by exhaustive search.) A computer has both of these capabilities; a person has neither of them. Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations. (They are also large, expensive to

maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed. But they are sufficiently pervasive that we must design our protocols around their limitations.) User authentication consists of a computer verifying that you are who you claim to be. There are three main techniques: • what you know • what you have • what you are Passwords are one method of reassuring someone that you are who you claim to be, and fit into the what you know category. Physical keys or ATM cards fit into the what you have category. Biometric devices, such as voice recognition systems or fingerprint analyzers, fit into the what you are category. 202 AUTHENTICATION OF PEOPLE 8.1 8.1 PASSWORDS Passwords predate computers. The concept is simple: if Alice needs to prove to Bob that she is Alice, and Bob does not know Alice by sight, they can prearrange a special greeting (a password), and Bob can assume that someone who says the magic word is Alice. As with many aspects of security, the classic uses of passwords are in the military. All members of a group are given the password of the day, and when they return to the fort after dark they state the password to prove they are not the enemy. Most readers have undoubtedly logged into a system by typing their user name and password so often that they no longer think about it. There are a lot of problems with using passwords for authentication: • An eavesdropper might see the password when Alice is using it to log in. • An intruder might read the file where the computer stores password information. • Alice's password might be easy to guess by someone making direct login attempts to the computer. • Alice's password may be crackable by an off-line computer search, given information such as a recognizable quantity encrypted with the password. (The difference between this off-line attack and the on-line attack above is in the number of password guesses that can be practically tried.) • In attempting to force users to choose unguessable passwords, the system might become so inconvenient that it becomes unusable, or users might resort to writing passwords down. This chapter describes the mechanisms in use to prevent unauthorized people from gaining access to passwords. The main lines of defense are keeping transmission of the password from being overheard, limiting the number of incorrect guesses, and making passwords difficult to guess. However, the intruder isn't the only bad guy. The authorized users will become the enemy, too, if the security mechanism becomes too inconvenient to deal with. 8.2 ON-LINE PASSWORD GUESSING I can impersonate you if I can guess your password. And that might not be hard. I2 once heard someone remark, about a faculty member at a major university, that "although in all other respects she appears to be a sentient human being, she insists on using her first name as her password." On 8.2 ON-LINE PASSWORD GUESSING 203 some systems, passwords are administratively set to a fixed attribute of a person, such as their birthday or their badge number. This makes passwords easy to remember and distribute, but it also makes them easy to guess. I1 once worked on a system where students' passwords were administratively set to their first and last initials. Faculty accounts were considered more sensitive, so they were protected with three initials. An astonishing number of banks allow Internet access based on your account number and a password, and they set initial passwords to the last four digits of your Social Security Number. In many states in the U.S., it is the local custom to use your Social Security Number as your driver's license number, and to use your driver's license as identification for cashing checks. That means cashiers can easily get your account number (from the front of the check) and your social security number (from the driver's license). It is surprising there has been so little fraud based on this poor design. Even if passwords are chosen so they are not obvious, they may be guessable if the impostor gets enough guesses. In fact, given enough guesses, any password, no matter how carefully chosen, can be guessed (by enumerating all finite character sequences until you hit on the correct password). Whether this is feasible depends on how many guesses it takes and how rapidly passwords can be tested. In

the military use of passwords, guessing is not a problem. You show up at the door. You utter a word. If it's the right word, they let you in; if it's the wrong word, they shoot you. Even if you know the password is the month in which the general was born, guessing is not an attractive pursuit. Most terminals do not have an "execute user" function controllable from the remote end (as useful as that might be), so this mechanism for limiting password guessing is not available. Given that people's fingers slip or they forget that they changed their password, it's important that they get more than one chance anyway. There are ways to limit the number or rate of guesses. The first is to design the system so that guesses have to actually be typed by a human. Computers are much faster and more patient than people at making guesses, so the threat is much greater if the impostor can get a computer to do the dirty work. Before the days of networks and PCs, this just meant that operating systems needed to ensure that they got passwords from terminals rather than programs. But now that login requests can come in over networks and lots of terminals are really PCs emulating terminals, there's really no way to prevent this. One seemingly attractive mechanism for preventing password guessing is to keep track of the number of consecutive incorrect passwords for an account and when the number exceeds a threshold, say five, "lock" the account and refuse access, even with a correct password, until the system is administratively reset. This technique is used with PINs on ATM cards—three wrong guesses and the machine eats your card. You have to show up at the bank with ID to get it back. An important downside of this approach is that a computer vandal (someone who simply wants to annoy people and disrupt operations) can, possibly with the aid of a computer, guess five bad passwords against all the accounts on a system and lock them all up, effectively shutting the system down until it is 204 AUTHENTICATION OF PEOPLE 8.2 administratively reset (assuming it can be administratively reset when all of the administrators' accounts are locked!). This represents little work for the vandal and can cause serious disruption. So this approach is unacceptable in most environments. Another approach to slow down a guesser is to only allow a limited number of account/password guesses per connection attempt. If the system is based on dialing in via modem or restarting a big slow application, the overhead of having to restart for each five attempts can slow down the guessing rate considerably (even with an auto-dial modem). A variant on this theme is simply to have incorrect passwords (or even all passwords) be processed s l o w l y . If you can't prevent guessing, maybe you can catch the guesser. By auditing invalid password attempts, a system manager can be alerted to the fact that an attempt is being made to penetrate the system. It might then be possible to try to trace the connection or take other corrective action. False alarms in audit logs can be prevented by filtering out the common case where a user mistypes a password and then immediately thereafter types it correctly. Events where a password fails and the connection then switches to a different account or disconnects are substantially more suspicious. A method for distributing the detective work is for systems to report to users when they log in the time of their previous login and the number of unsuccessful password attempts since the last successful login. In practice, few users will check the time of last successful login to see whether it looks right, but most will notice reports of failed login attempts if they did not mistype the password themselves. This doesn't work for "stale" accounts that the user never logs into. Such accounts are subject to other threats as well; it's good security policy to eliminate accounts not in regular use. The expected time to guess a password is the expected number an impostor has to guess before getting it right divided by the guess rate. So far, we've concentrated on limiting the rate of password guesses. Another promising approach is to ensure that the number of passwords an attacker would need to search is large enough to be secure against an off-line, unaudited search with a lot of CPU power. If passwords are randomly chosen eight-character strings made from the characters a–z, then even if an attacker could