guess a password every millisecond, it would take (on average) over three years to guess it. This would make all the more obtrusive and expensive preventive mechanisms unnecessary. Some administrations assign such passwords (randomly chosen 8-character strings). These systems have another problem: Users hate them...and forget them...and write them down. As discussed later, when users write down passwords rather than memorizing them, a whole new class of threats open up. Many otherwise fine minds are genetically incapable of memorizing randomly chosen eight-character passwords. Sometimes the random password generator is clever enough to generate pronounceable strings, which makes it a little easier for the human to remember the password. Constraining the generated passwords to pronounceable strings of the same length limits the number of possible passwords by at least an order of magnitude, but since a 10-character pronounceable string is probably easier to remember than an 8-character completely random string, and is about as secure, generating pronounceable strings is a good idea if the administrator wants to impose passwords. 8.3 OFF-LINE PASSWORD GUESSING 205 A better approach is to let users choose their own passwords but to warn them to choose "good" ones and enforce that choice where possible. Usually the best combination of memorability and difficulty of guessing is a "pass-phrase" with intentional misspelling or punctuation and odd capitalization, like GoneFi$hing or MyPassworDisTuff!, or the first letter of each word of a phrase, like Mhall;Ifwwas (Mary had a little lamb; Its fleece was white as snow). The program that lets users set passwords should check for easy-to-guess passwords and disallow them. It might, for example, run them through a spell-checking dictionary and reject them if they are spelled correctly! (A favorite attack is to guess all the words in a dictionary as passwords; a large dictionary might contain 500000 words, and it is not difficult to check that many.) If possible, the dictionary should be extended with common first and last names. The program could also require mixed cases, non-alphabetic characters, a minimum length, and a minimum number of different characters. Surprisingly, most variants of UNIX limit password length to eight characters (or more precisely, they only check the first eight). VMS allows long passwords, and allows the administrator to set policies such as how long they must be, how often they must be changed, and how many old passwords the system remembers in order to prevent a user from reusing an old password. 8.3 OFF-LINE PASSWORD GUESSING In the previous section, we discussed on-line password guessing, where guessing can be slowed down and audited. Passwords do not have to be very strong if the only threat is on-line password guessing. But sometimes it is possible for an attacker, through either eavesdropping or reading a database, to obtain a cryptographic hash of the password or a quantity encrypted with the password. In such a case, even though the attacker cannot reverse the hash of the password, the attacker can guess a password, perform the same hash, and compare it with the stolen quantity. Once the attacker has obtained the hash of the password, the attacker can perform password guessing without anyone knowing it, and at a speed limited only by procurable compute power. We've talked a lot about how users should protect their passwords. Remember that passwords must be stored in two places—users must know their own, and systems must be able to verify passwords. If the system simply directly stores passwords, the password list on a system must be protected at least as well as anything else on the system. Even that is not sufficient. Most systems keep backup tapes for use in the case where a disk fails or a user accidentally destroys a file. If those tapes contain password lists, they must be protected as carefully as the system; anyone who can read them will subsequently be able to impersonate all users. Several techniques can lessen this threat. Some operating systems, for instance UNIX and VMS, do not store actual user passwords. They take passwords as typed, run them through a 206 AUTHENTICATION OF PEOPLE 8.3 one-way algorithm such as a cryptographic hash, and store the results. With such a system, the

password file needs to be carefully protected from modification, but disclosure of the file is not immediately of use to an attacker. Standard UNIX systems, in fact, place such confidence in this technique that they make the password file publicly readable. This turns out to be a bad idea. Even though it is impossible to compute the password from the hash, it is possible to guess a password and verify whether you got it right by hashing it and comparing it to the stored value. This gives attackers an unaudited and fairly high-performance way of guessing passwords. Many systems have now been modified to prevent unprivileged programs from reading the password file. Another interesting downside of this technique is that if a user forgets a password, the system administrator can't look it up. Instead, the user must choose a new password and the system administrator must install it. The only time this is a problem is when an attacker has changed the user's password but the user isn't sure whether it was changed nefariously or merely forgotten. Being able to read the old value might make it easier to tell the difference. When disclosure of whole files full of hashed passwords is a concern, another useful technique is to apply salt. Rather than guessing passwords against a single user account, an attacker with a file full of hashed passwords might hash all the words in a dictionary and check to see whether any of the passwords match any of the stored hashed values. This would be a much more efficient means of attack, particularly when the attacker just wanted any account on the system or, Password Hash Quirk There was an operating system (which we'll allow to remain nameless) that stored hashes of passwords, and the password hash used by that operating system had an interesting property. We are purposely not giving the exact details, but the general idea is that there was a magic character sequence X such that the hash of any string S was the same as the hash of X concatenated with S. This is not exactly a security flaw, but instead is a wonderfully user-friendly feature. How can this odd feature be useful? Suppose the system administrator made the common policy decision that user passwords had to be some minimum length. This particular operating system enforced the policy only when you set your password (as opposed to when you logged in). Now suppose you wanted a password you could type quickly, like FOO. The operating system would not let you set your password to FOO because it's not long enough. But it would let you set it to X concatenated with FOO. Let's say the magic string was %#v27dR678riwueyru3ir3. You'd set your password to %#v27dR678riwueyru3ir3FOO. That would be the last time you'd have to type %#v27dR678riwueyru3ir3FOO. From then on, FOO would work just fine, since the hash of FOO is the same as the hash of %#v27dR678riwueyru3ir3FOO. 8.4 HOW BIG SHOULD A SECRET BE? 207 more likely, as many accounts as possible on the system. Tests at universities have shown 30% hit rates with fairly small dictionaries. A way to slow down such an attacker is as follows: When a user chooses a password, the system chooses a random number (the salt). It then stores both the salt and a hash of the combination of the salt and the password. When the user supplies a password during authentication, the system computes the hash of the combination of the stored salt and the supplied password, and checks the computed hash against the stored hash. The presence of the salt does not make it any harder to guess any one user's password, but it makes it impossible to perform a single cryptographic hash operation and see whether a password is valid for any of a group of users. Another technique is to encrypt the password file. This doesn't eliminate the problem of keeping passwords secret; it just reduces it to the problem of protecting the key that decrypts the password file. This might be done by having the key held in operating system memory, not backed up to disk or tape, and having it either stored in special nonvolatile memory in the machine or reentered by the operator on boot-up. 8.4 HOW BIG SHOULD A SECRET BE? From how big a space must a secret be chosen in order to be secure? To thwart an on-line attack the secret does not have to be chosen from a large space, because the intruder is detected after a small number of guesses or because the

amount of time the intruder needs to guess correctly is worth much more than the payoff. For instance, many ATM systems have only four decimal digits worth of secret, which means there are only 10000 different secrets, and this is really sufficiently secure because you only get three guesses. If there's the opportunity to do off-line password guessing, the secret must be chosen from a much larger space. The general rule of thumb is that a secret needs about 64 bits of randomness, since it is considered computationally infeasible to search 264 possibilities. Therefore a 64-bit secret cryptographic key is reasonably secure. Humans aren't usually willing to remember or type 64-bit (about 20-digit) random numbers (though one of our children is perfectly happy to remember dozens of digits of , and another of user ID salt value password hash Mary 2758 hash(2758|passwordMary) John 886d hash(886d|passwordJohn) Ed 5182 hash(5182|passwordEd) Jane 1763 hash(1763|passwordJane) 2 208 AUTHENTICATION OF PEOPLE 8.5 our children seems to know innumerable 7-digit phone numbers). How big does a password have to be in order to have the equivalent security of a 64-bit random number? If the text string is truly randomly chosen from upper- and lower-case letters, the 10 digits, plus a couple of punctuation marks, there are 64 possibilities per keystroke (6 bits), so an 11-character password would be necessary. A human will not remember a randomly chosen 11-character string. What about a randomly chosen pronounceable password? If a computer were to generate such strings (as opposed to allowing a human to choose), the pronounceability constraint means that about every third character must be a vowel. To be memorizable, the password should be a case-insensitive string of letters, giving only about 4½ bits per character, and only about 2½ bits per vowel (since there are 6 vowels in English). The combination of limiting characters to case-insensitive letters and having a reasonable percentage of vowels yields randomness of about 4 bits per character. This would require a 16-character computer-generated pronounceable string, which is also too long for a person to willingly memorize or type. What about if people are allowed to choose their own passwords? The general wisdom is that the randomness achievable in human-generated passwords is about 2 bits per character, which would result in a 32-character password, which is also too long. The conclusion is that a secret a person would be willing to memorize and type will not be as good as a 64-bit random number, and therefore passwords will be open to off-line password-guessing attacks. 8.5 EAVESDROPPING A second important weakness in using passwords is that they must be "uttered" to be used, and there is always a chance of eavesdropping when they are uttered. The lowest-tech form of eavesdropping is to watch as someone types a password. There is an etiquette around the operation of ATMs. Even when there is a substantial line, the second person stands a discreet distance behind the person operating the machine and looks away. The etiquette is not so highly evolved in an office environment, and it often isn't difficult to pick up passwords this way. The only prevention beyond training people to be more paranoid is to pick passwords involving shift and control characters so more fingers are involved. Of course, it's easier to watch the screen than fingers. Most systems have the sophistication to not display passwords as they are being typed. A more high-tech method of eavesdropping is to place a wiretap on the communications line and watch all the passwords (and everything else, for that matter) go by. Whether this is easy or hard depends on the environment. There are also software-based keystroke logs that can be covertly installed on PCs and even hardware logging devices that can be embedded in keyboards or cables. 8.6 PASSWORDS AND CARELESS USERS 209 If you're going to give up on memorization and accept the risks of written-down passwords, an effective mechanism is one-time passwords. Here, the user and the system have a list of valid passwords, but each one is only valid once. After it's used, it is crossed off the list (at both ends). Periodically, the user must get a new list from the system administrator. This mechanism is nearly impervious to eavesdropping. A variant is to have a numbered list of

passwords and have the system ask for specific ones on each authentication. An attacker would have to eavesdrop on many authentications before obtaining enough information to impersonate the user. With this technique, the list can be somewhat shorter and can be used for longer than the one-time password list, but at some loss of security and convenience. (See Homework Problem 2.) 8.6 PASSWORDS AND CARELESS USERS At a lecture on computer security, a professor asked, “Are there any advantages of passwords over biometric devices?” A helpful student replied “When you want to let someone use your account, with a password you just give it to them, while with a biometric device you have to go with them until they are logged in.” This is the sort of remark that sends chills down the back of security administrators and makes them think of their users as adversaries rather than the customers they are trying to protect. Security people need to remember that most people regard security as a nuisance rather than as needed protection, and left to their own devices they often carelessly give up the security that someone worked so hard to provide. The solution is to educate users on the importance of security, helping them to understand the reasons for the procedures they are asked to follow, and making those procedures sufficiently tolerable that they don’t develop contempt for the process. Passwords are particularly easy to abuse. The classic tale is one where the system manager’s password is posted on the console because it is too long and complex to remember. One particular danger is users’ including passwords on-line in accessible places. In addition to doing this as a substitute for little black books, users might find it convenient to include passwords in scripts used to automate access to other systems, or they might include passwords in messages sent via electronic mail. Electronic mail messages are frequently archived. Both schemes allow an attacker who has gotten into one account to cascade those rights into other accounts on other systems. 210 AUTHENTICATION OF PEOPLE 8.6.1 8.6.1 Using a Password in Multiple Places One of the tough trade-offs is whether to recommend that users use the same password in multiple places or keep their passwords different for different systems. All things being equal, use of different passwords is more secure because if one password is compromised it only gives away the user’s rights on a single system. Things are rarely equal, however. When weighed against the likelihood that users will resort to writing passwords down if they need to remember more than one, the trade-off is less clear. An issue that weighs in favor of different passwords is that of a cascaded break-in. An attacker that breaks in to one system may succeed in reading the password database. If users have different passwords on different systems, this information will be of no use. But if passwords are common, a break-in to a system that was not well protected because it contained no “important” information might in fact leak passwords that are useful on critical systems. 8.6.2 Requiring Frequent Password Changes Security is a wet blanket. —apologies to Charles Schulz A technique that offers some security but is probably overused, is requiring frequent password changes. The idea behind frequent password changes is that if someone does learn your password, it will only be useful until it next changes. This protection may not be worth much if a lot of damage can be done in a short time. The problem with requiring frequent password changes is that users are more likely to write passwords down and less likely to give much thought or creativity to choosing them. The result is observable and guessable passwords. It’s also true that users tend to circumvent password change policy unless enforcement is clever. In a spy-vs.-spy escalation, the following is a common scenario: 1. The system administrator decides that passwords must be changed every 90 days, and the system enforces this. 2. The user types the change password command, resetting the password to the same thing as before. 3. The system administrator discovers users are doing this and modifies the system so that it makes sure the password, when changed, is set to a different value. 4. The user then does a change password procedure that sets the password to something new, and then immediately

sets it back to the old, familiar value. 8.6.3 PASSWORDS AND CARELESS USERS 211 5. The system administrator then has the system keep track of the previous n password values and does not allow the password to be set to any of them. 6. The user then does a change password procedure that goes through n different password values and finally, on the n+1st password change, returns the password to its old familiar value. 7. The system is modified to keep track of the last time the password was changed and does not allow another password change for some number of days. 8. The user, when forced to change passwords, constructs a new password by appending '1' to the end of the previous password. Next time the user replaces the '1' with a '2', ... 9. The system, in looking for guessable passwords, looks for passwords that look "too much like" one of the previous n passwords. 10. The users throw up their hands in disgust, accept impossible-to-remember passwords, and post them on their terminals. All of these techniques have been tried on some system or another. In general, it's impossible to make systems secure without the cooperation of the legitimate users. If frequent password changes address a real threat, users must be educated to fear that threat so they will strive for good passwords. If the threat cannot be made real to the users, the inconvenience won't be worth the trouble. 8.6.3 A Login Trojan Horse to Capture Passwords A threat as old as timesharing is to leave a program running on a public terminal that displays a login prompt. An unsuspecting user then enters a user name and password. The Trojan horse program logs the name and password to a file before the program terminates in some way designed to minimize suspicion. There is no protection from this threat given a sufficiently naive user. Approaches toward minimizing it are based on making it difficult for a program to "look like" a normal login prompt. For example: • On some systems, any program request for input is preceded by an automatically inserted '?'. If the real login prompt is made to have no '?', a program can't produce a believable counterfeit. • On the bit-mapped screens associated with workstations and X-windows terminals, the login screen takes up the entire screen and has no border. If the protocol prevents ordinary programs from displaying such a screen, they can't counterfeit a login prompt very reasonably. • On most systems, there is some way to interrupt running programs. Training users to enter the interrupt key sequence before logging in would then thwart such Trojan horses. For exam- 212 AUTHENTICATION OF PEOPLE 8.6.4 ple, newer Windows systems require the user to type Ctrl-Alt-Delete as part of the login sequence. In (almost) any scenario other than a login prompt, this key sequence would interrupt a running program. Unfortunately, many systems allow programs to disable interrupts; for those systems, this technique would not be useful. Even if the Trojan horse program can do the login prompt exactly, it might not be able to exactly duplicate the way the system behaves after a user logs in. This will make an alert user suspicious. For example: • A reasonable action following the capture of a password is to log it, put out the message the user would get if the password were mistyped, and then terminate and return to a regular login prompt. The user will assume the password was mistyped the first time and try again. This can be prevented if a program can't end its session without a logout banner. Such a banner before the second login message is a dead giveaway. Mentioned earlier was having a message displayed at login telling users the number of unsuccessful login attempts to the account since the last successful attempt. This doubles as Trojan horse protection because the successful login would not report that there had just been a failed login attempt. • Another thing the Trojan horse program can do is hang after accepting the user's name and password, as if the system had gotten into a wedged state. If the program can prevent the user from using any escape characters, the user will just get frustrated and reboot or reconnect. Given how accustomed we all have become to the flakiness of our computers, very few users would find this behavior suspicious. • An optimally designed Trojan horse would actually use the username and password to log the user in and run a session. Some operating systems offer sufficient

modularity and generality to make this possible. It would be a security feature to make this evident to processes started in non-standard ways so that some sort of warning banner could be displayed. 8.6.4 Non-Login Use of Passwords Entering a single password to log into a system and subsequently having access to all the user's resources is, in most environments, a reasonable mix of security and convenience. It's not the only way to do it. Some systems permit password protection on individual files. A user could specially protect certain files so that someone learning the user's login password still couldn't get at those files. Similarly, applications could require their own authentication of a user before permitting access to certain databases (even after the user has logged in). 8.7 INITIAL PASSWORD DISTRIBUTION 213 8.7 INITIAL PASSWORD DISTRIBUTION Everything we've discussed so far describes how systems work once users have passwords and the systems know them. Another opportunity to get it wrong exists in the administrative procedures involved in getting to that state. After all, if you can impersonate the user to the system administrator, you can get the password by whatever mechanism the user could. A secure method for the initial distribution of passwords is for the user to appear at the terminal of the system administrator and authenticate by whatever means humans use to authenticate (driver's license, student ID, birth certificate and two major credit cards, whatever). The system administrator then sets up all the particulars of the account for the user except the password (name, rights, quotas, choice of shell, ...) and then lets the user choose a password. This method has two drawbacks: it may be inconvenient for the user to meet the system administrator; and it's a little scary to let this new user type to this highly privileged terminal session while the system administrator discreetly looks away. A skilled user could probably do substantial damage in a short time. Even this problem could be circumvented by giving the user access to a special keyboard that only accepts passwords. This mechanism is sometimes used in special-case environments like establishing the PIN for an ATM. For a bank this is a sufficiently high-volume and security-sensitive application to justify special hardware. Another variant on this theme is for the system administrator to create the account and an initial strong (randomly chosen from a large space) password, give it to the user, and instruct the user to use the password only for an initial login and then change it to something more easily remembered. Some systems support the notion of pre-expired passwords that require the user to change them as part of the login process. Even if the initial strong password is written down and the paper handed to a user, little security is lost because the password is no good after it is used and the user is likely to keep it safe that long. In the name of convenience, there are weaker versions of this protocol. The administrator could set up the account and distribute the password by (paper) mail, in which case the security is as good as the security of mail. A common but insecure practice is to have the initial password be either a constant or an easily determined property of the user. A school might, for example, set all student passwords initially to their student IDs. Then they could communicate passwords to students with a broadcast (published) message rather than sending individualized notes. The security of this might be vaguely acceptable if the only thing a student account password protects is the work of the student, which is presumably nothing until the student uses the account for the first time. Even so, an attacker could log into the student's account ahead of time and plant Trojan horses. The following story is true (or rather was told to me1 as true by a person supposedly involved). Names are omitted to protect the guilty. See how many things you can find wrong with the way security was administered. 214 AUTHENTICATION OF PEOPLE 8.8 At a large commercial timesharing service bureau, security of privileged accounts (those that could access other users' data) was considered paramount. One of the security measures used to protect these accounts was that the passwords were randomly chosen and changed weekly. Every Sunday afternoon a batch job was run, selecting new random passwords and changing all

the privileged accounts. Part of the batch job was to print a list of all the accounts and their passwords. This list was posted on a bulletin board so that as people came in Monday morning they could get their new passwords. This worked fine until one Sunday the printer was broken. All the passwords were changed, but they weren't printed. Monday morning the printer was fixed, but no one could log in to request that the list be printed. In fact, no one could log in at all (except the commercial users, who saw no disruption). Some systems have a way to perform a privileged login given physical access to the machine, but this operating system didn't—it would support such an override only upon also erasing the disks, which by that time included unbacked-up commercial data. Many systems that do have such an override require that the system be rebooted in single-user mode, which would have been an unacceptable interruption of commercial service. After thinking hard, some bright fellow noticed that the space from which the passwords were randomly selected was not unmanageably large, so they got all the people who couldn't do any work anyway (about 50) to repeatedly attempt logins to a single account with each of them assigned a range of passwords. By the end of the day Monday, one of them had succeeded and they were able to return to business as usual. 8.8 AUTHENTICATION TOKENS An authentication token is a physical device that a person carries around and uses in authenticating. In the breakdown of security according to what you know, what you have, and what you are, authentication tokens fall in the middle category (passwords are in the first, and biometric devices the third). Generally, authentication tokens offer security advantages and disadvantages over other mechanisms. Unless they are physically attached to users (which is unacceptable in our culture), they are subject to theft. Generally they must be coupled with one of the other two mechanisms to be secure. There are several forms of authentication token in use today. The most ubiquitous is the key that people use to unlock their home or car. Another common form of authentication token is the credit card. If a credit card includes a picture or a signature, it combines an authentication token with a primitive biometric device (the person who compares signatures or sees whether you look like the picture). 8.8 AUTHENTICATION TOKENS 215 Credit cards these days contain a magnetic strip that contains information. The advantage that magnetic strip cards offer over simple passwords is that they are not trivial to reproduce and they can conveniently hold a secret larger than most people are willing to memorize. Perhaps the biggest advantage is psychological—people tend to be less willing to "loan" a token to a friend than to share a password. There are a number of disadvantages: • Use of these tokens requires custom hardware (a key slot or card reader) on every access device. This may be expensive and it requires standardization. • Tokens can be lost or stolen. For reasonable security, tokens must be supplemented with a PIN or password. In most environments, a convenient override must be available for when "I forgot my card at home." That override should not be substantially less convenient than the one for "I forgot my password." These devices offer little or no protection against communications eavesdropping. Whatever information is sent "over the wire" can be collected just like a password and replayed later. To make use of the information using a "standard terminal" requires that a card be manufactured to regurgitate the stolen information, but someone who can eavesdrop can likely connect a nonstandard terminal to the network and replay the information without making a card. There is nothing conceptually difficult about copying a key or mag-stripe card; devices for doing so are readily available. A better form of authentication token is the smart card. This is a device about the size of a credit card but with an embedded CPU and memory. When inserted in a (misleadingly named) smart card reader, the card carries on a conversation with the device (as opposed to a magnetic strip, which simply dumps its contents). There are various forms of smart cards: • PIN protected memory card. With this card, there is information in the memory of the card that can only be read after a PIN is input to the card. Usually, after some number of

wrong PIN guesses, the card “locks” itself and will not give the information to anyone. Information stored on such a smart card is safer than that stored on a magnetic strip card because a stolen card is useless without the PIN. These cards are more difficult to duplicate than magnetic strip cards, but it’s still possible given the PIN. • Cryptographic challenge/response cards. With this card, there is a cryptographic key in memory, and the card is willing to encrypt or decrypt using the key but will not reveal the key even after the PIN is entered. A computer that knows the key in the card can authenticate the user by creating a random challenge and “challenging” the card to encrypt or decrypt it. If the correct answer is returned, the computer can have confidence that the smart card is present and the correct PIN was entered. These cards can be constructed so as to be nearly impossible to duplicate or to extract the key from. Since there is no way to directly extract the key, it can only be done by disassembling or inserting probes into the card. There is a reciprocating esca- 216 AUTHENTICATION OF PEOPLE 8.8 lation in the technologies for probing the card and for packaging it to be unreadable. For most practical purposes, the cards are unreadable. Like keys and magnetic strip cards, the serious practical problems with smart cards are the need for readers at every access point and the need for recovery when a card is lost or forgotten. The cryptographic card offers substantial protection against eavesdropping. • Cryptographic calculator (sometimes called a readerless smart card). A cryptographic calculator is like a smart card in that it performs cryptographic calculations using a key that it will not disclose. It is unlike a smart card in that it requires no electrical connection to the terminal. It has a display and usually a keyboard, and all interaction is through the user. One way it could work is by simulating a smart card: The user enters a PIN to unlock the device; the computer wishing to authenticate the user generates a random challenge and displays it to the user; the user types it into the calculator; the calculator encrypts the value and displays the result; the user enters the result on the terminal; the computer does the same calculation and compares the results. An alternative protocol that cuts the typing in half is for the calculator to encrypt the current time and display the result. The user types in this number in place of a password. There’s a little more work for the computer since it will not be sure of the exact time that the calculator thinks it is (clocks drift); it will have to do the calculation on several candidate time values to verify what the device said. It might then record the accumulated clock skew to make the next such calculation easier. It’s possible to eliminate the keyboard (needed for entering the PIN) from the calculator by having a PIN or password sent to the computer instead. It is important to have some form of PIN to prevent someone who steals the calculator from impersonating its owner. In addition to saving typing, another advantage of the time encryption protocol is that it fits the “form factor” of protocols designed for passwords. If a protocol has a “password” field and no way for the authenticating application to send a challenge, the encrypted time variant can still be made to work. The biggest advantage of these readerless smart cards is that they can be used from ordinary terminals with no special hardware. Their popularity is growing among companies that want to let their employees log in from home using laptops and modems but are afraid of opening their networks up to intruders. 8.9 PHYSICAL ACCESS 217 8.9 PHYSICAL ACCESS A low-tech way of performing user authentication is to have human guards do it “at the door”. If a system is only accessible from protected areas, and it requires some form of person-to-person authentication to enter those areas, then any authentication the system does is for purposes of differentiating legitimate users from each other as opposed to differentiating legitimate users from illegitimate users. For high-security applications, this may be perfectly reasonable. Many bank transactions, for example, can only be initiated at tellers’ terminals inside the bank. Alternatively, the location from which access is requested can be part of the authentication process, where fewer rights are granted from less secure access points. For

instance, ATMs and the tellers' terminals connect to the same computer, but support different transaction types. 8.10 BIOMETRICS Biometric devices sell best in places where users aren't afraid of technology or don't have a choice. —Wall Street Journal, Oct 13, 1992, William M. Bulkeley In dividing authentication mechanisms into what you know, what you have, and what you are, biometric devices authenticate you according to what you are. They measure your physical characteristics and match them against a profile. You can't "loan out" anything that would help someone fool a biometric authentication device; nor can anything be stolen. There are a variety of biometric devices available. All are too expensive to be in everyday use, but in some cases the costs are coming down to where we may see these. Technology available today includes: • Retinal scanner. This is a device that examines the tiny blood vessels in the back of your eye. The layout is as distinctive as a fingerprint and apparently easier to read. These devices are quite expensive and have a "psychologically threatening" user interface. • Fingerprint readers. This would seem an obvious technology since fingerprints have been used as a method of identification for many years. For some reason, automating this technology has never been very successful, though there are devices available. 218 AUTHENTICATION OF PEOPLE 8.10 • Face recognition. Looking at a digitized picture of a person, a computer can measure facial dimensions and do a good job of recognizing people. Just don't show up at work with a black eye and a swollen jaw. • Iris scanner. Like a retinal scanner, this maps the distinctive layout of the iris of your eye. It has the major advantage of having a less intimidating user interface—rather than requiring you to look into a laser device, iris scans can be done with a camera several feet away and might even be done covertly. • Handprint readers. These are more widely used than fingerprint readers. They measure the dimensions of the hand: finger length, width, and so on. They are not as accurate as fingerprints (more false positives), but they are less expensive and less problem-prone. • Voiceprints. It turns out that it's possible to do a frequency spectrum analysis of someone's voice and get identification nearly as accurate as a fingerprint. This technology is in use, but has not caught on in spite of the fact that it should be fairly cheap. It can be defeated with a tape recording, and it may refuse to authenticate someone whose voice has shifted due to illness. • Keystroke timing. The exact way in which people type is quite distinctive, and experiments have been done with identification based on the way people type. There is a problem that various injuries can throw off timing, and the networks that connect terminals and computers tend to lose the keystroke timing information before it reaches a processor that can use it. • Signatures. These are a classic human form of authentication, and there are human experts quite adept at determining whether two signatures were produced by the same person. Machines thus far have not been able to duplicate that ability. However, when not just the signature is recorded, but the actual timing of the movements that go into scribing the signature, there is sufficient information for authentication, and some systems use this method, with the user signing on an electronic tablet. It might seem tempting to build a portable fingerprint reader as a peripheral to a laptop or PC that lets you stick your finger in and turns it into bits. Such devices are commercially available. The bits are then sent to a remote computer as a form of authentication. The problem with this approach is that the biometric properties cannot reasonably be kept secret. Anyone who knows your fingerprint can transmit the relevant fingerprint data. The only way in which a remote biometric device can be secure is if the device possesses a tamper-resistant secret and communicates with the remote computer via a cryptographically protected exchange. 8.11 HOMEWORK 219 8.11 HOMEWORK 1. Design a password hash algorithm with the property stated in Password Hash Quirk on page 206. It should be impossible to reverse, but for any string S it should be easy to find a longer string with the same hash. 2. In §8.5 Eavesdropping we described a scheme in which a user has a numbered list of passwords and the system asks for some small

subset on each login. Is there any advantage in asking for more than one password per login? Note that the more passwords requested, the more information an eavesdropper will get. On the other hand, when the eavesdropper attempts to impersonate the user, the more passwords requested, the less likely the eavesdropper will know all of them. This page is intentionally left blank 221 9 SECURITY HANDSHAKE PITFALLS Knock Knock! Who's there? Alice. Alice who? and you'll have to read on to find secure ways of continuing... Security in communications almost always includes an initial authentication handshake, and sometimes, in addition, integrity protection and/or encryption of the data. Let's assume Alice and Bob wish to communicate. In order to communicate, they need to know some information about themselves and about the other party. Some of this information is secret. Some usually isn't, such as the names Alice and Bob. In §7.3 Cryptographic Authentication Protocols we described some example security hand-shakes. Although they may seem straightforward, minor variants of secure protocols can have secu-rity holes. As a matter of fact, many deployed protocols have been designed with security flaws. This stuff just isn't that hard. How come nobody gets it right? —Al Eldridge This book does not tell you the one "best" protocol. Different protocols have different tradeoffs. Some threats are more likely in some situations, and different resources are available in terms of computational power, specialized hardware, money to pay off patent holders, humans willing and able to be careful, and so forth. We mortals should never need to design our own cryptographic algorithms (like DES, RSA, or MD5)—we can leave that in the able hands of Ron Rivest and a handful of other specialists. But it is often the case that people outside the security community, such as implementers or protocol designers, have to design security features into protocols—including authentication handshakes. It is crucial that potential flaws be well understood. Even when a protocol is patterned after known protocols, the slightest alteration, even in something that "couldn't 222 SECURITY HANDSHAKE PITFALLS 9.1 conceivably matter", can introduce subtle security flaws. Or even if a new flaw is not introduced, a weakness that was not important in the original protocol might be serious in a different environment. In practice, the way that security protocols are designed is by starting with some design, which is almost certainly flawed, and then checking for all the weaknesses one can imagine, and fixing them. So the more educated we can become about the types of flaws likely to be in a protocol, the more likely it is that we'll understand all the properties of a protocol we deploy. In this chapter we describe some typical protocols and evaluate them according to performance (number of messages, processing power required, compactness of messages) and security. 9.1 LOGIN ONLY A lot of existing protocols were designed in an environment where eavesdropping was not a concern (rightly or wrongly), and bad guys were (rightly or wrongly) not expected to be very sophisticated. The authentication in such protocols generally consists of: • Alice (the initiator) sends her name and password (in the clear) across the network to Bob. • Bob verifies the name and password, and then communication occurs, with no further attention to security—no encryption, no cryptographic integrity protection. A very common enhancement to such a protocol is to replace the transmission of the cleartext password with a cryptographic challenge/response. First we'll discuss protocols based on shared secrets, using either secret key cryptographic algorithms or message digest algorithms. Then we'll discuss similar protocols using public key technology. 9.1.1 Shared Secret The notation f(KAlice-Bob,R) means that R is cryptographically transformed, somehow, with Alice and Bob's shared secret KAlice-Bob. This could be done by using KAlice-Bob as a secret key in some algorithm such as DES or AES, and using KAlice-Bob to encrypt R. Or it could be done by hashing R and KAlice-Bob, for instance by concatenating R and KAlice-Bob and computing a message digest on the result. When we explicitly mean encryption with KAlice-Bob we'll write KAlice-Bob{R}. When we explicitly mean a hash, we'll write h(KAlice-Bob,R) or hash(KAlice-

Bob,R). Consider Protocol 9-1. An eavesdropper will see both R and f(KAlice-Bob,R). It is essential that seeing the pair does not enable the eavesdropper to derive KAlice-Bob. 9.1.1 LOGIN ONLY 223 This protocol is a big improvement over passwords in the clear. An eavesdropper cannot impersonate Alice based on overhearing the exchange, since next time there will be a different challenge. However, there are some weaknesses to this protocol: • Authentication is not mutual. Bob authenticates Alice, but Alice does not authenticate Bob. If Trudy can receive packets transmitted to Bob's network address, and respond with Bob's network address (or through other means convince Alice that Trudy's address is Bob's), then Alice will be fooled into assuming Trudy is Bob. Trudy doesn't need to know Alice's secret in order to impersonate Bob—she just needs to send any old number R to Alice and ignore Alice's response. • If this is the entire protocol (i.e., the remainder of the conversation is transmitted without cryptographic protection), then Trudy can hijack the conversation after the initial exchange, assuming she can generate packets with Alice's source address. It's also useful to Trudy, but not absolutely essential, that she be able to receive packets transmitted to Alice's network layer address. (See Homework Problem 1.) • An eavesdropper could mount an off-line password-guessing attack (assuming KAlice-Bob is derived from a password), knowing R and f(KAlice-Bob,R). (Recall that an off-line passwordguessing attack is one in which an intruder captures information against which passwords can be tested in private, so in this context it means guessing a password, turning that password into a key K, and then seeing whether f(K,R) equals f(KAlice-Bob,R).) • Someone who reads the database at Bob can later impersonate Alice. In many cases it is difficult to protect the database at Bob. There might be many servers where Alice uses the same password, and although the administrators of most of the servers might be very conscientious about security (not letting unauthorized people get near their machines, and enforcing unguessable passwords), it only takes one unprotected server for an intruder to read the relevant information. Furthermore, protecting the database implies protecting all the backup media as well, by either preventing access to it (locking it in a safe) or encrypting the contents and somehow protecting the key with which it was encrypted. Alice I'm Alice Bob a challenge R f(KAlice-Bob,R) Protocol 9-1. Bob authenticates Alice based on a shared secret KAlice-Bob 224 SECURITY HANDSHAKE PITFALLS 9.1.1 Despite these drawbacks, if there are limited resources available for adding security, replacing the cleartext password transmission is the single most important security enhancement that can be done. A minor variant on Protocol 9-1 is the following: In this protocol Bob chooses a random challenge R, encrypts it, and transmits the result. Alice then decrypts the received quantity, using the secret key KAlice-Bob to get R, and sends R to Bob. This protocol has only minor security differences from Protocol 9-1: • This protocol requires reversible cryptography, for example a secret key cryptographic algorithm. Protocol 9-1 can be done using a hash function. For example, f(KAlice-Bob,R) could be the message digest of KAlice-Bob concatenated with R. But in Protocol 9-2, Alice has to be able to reverse what Bob has done to R in order to retrieve R. Sometimes there is a performance advantage to being able to use one of the message digest functions rather than having to use, say, DES. Sometimes there are export issues involved in having code for encryption available, even if it's only used for authentication, whereas using a message digest function would be less likely to create export problems. • Suppose KAlice-Bob is derived from a password and therefore vulnerable to a dictionary attack. If R is a recognizable quantity, for instance a 32-bit random number padded with 32 zero bits to fill out an encryption block, then Trudy can, without eavesdropping, mount a dictionary attack by merely sending the message I am Alice and obtaining KAlice-Bob{R}. If Trudy is eavesdropping, however, and sees both R and KAlice-Bob{R}, she can mount a dictionary attack with either protocol. It is often the case that eavesdropping is more difficult than merely sending a message claiming to be Alice.

Kerberos V4 (see Chapter 11 Kerberos V4) is an example of a protocol that has this security weakness. • If R is a recognizable quantity with limited lifetime, such as a random number concatenated with a timestamp, Alice authenticates Bob because only someone knowing KAlice-Bob could generate KAlice-Bob{R}. To accomplish mutual authentication, R must be limited lifetime to foil the replaying of an old KAlice-Bob{R}. Alice I'm Alice Bob KAlice-Bob{R} R Protocol 9-2. Bob authenticates Alice based on a shared secret key KAlice-Bob 9.1.1 LOGIN ONLY 225 Another variant on Protocol 9-1 is to shorten the handshake to a single message by having Alice use a timestamp instead of an R that Bob supplies: This modification requires that Bob and Alice have reasonably synchronized clocks. Alice encrypts the current time. Bob decrypts the result and makes sure the result is acceptable (i.e., within an acceptable clock skew). The implications of this modification are: • This modification can be added very easily to a protocol designed for sending cleartext passwords, since it does not add any additional messages—it merely replaces the cleartext password field with the encrypted timestamp in the first message transmitted by Alice to Bob. • The protocol is now more efficient. It goes beyond saving two messages. It means that a server, Bob, does not need to keep any volatile state (such as R in Protocol 9-1) regarding Alice (but see next bullet). This protocol can be added to a request/response protocol (such as RPC) by having Alice merely add the encrypted timestamp into her request. Bob can authenticate the request, generate a reply, and forget the whole thing ever happened. • Someone eavesdropping can use Alice's transmitted KAlice-Bob{timestamp} to impersonate Alice, if done within the acceptable clock skew. This threat can be foiled if Bob remembers all timestamps sent by Alice until they "expire" (i.e., they are old enough that the clock skew check would consider them invalid). • Another potential security pitfall occurs if there are multiple servers for which Alice uses the same secret KAlice-Bob. Then an eavesdropper who acts quickly can use the encrypted timestamp field Alice transmitted, and (if still within the acceptable time skew) impersonate Alice to a different server. This can be foiled by concatenating the server name in with the timestamp. Instead of sending KAlice-Bob{timestamp}, Alice sends KAlice-Bob{"Bob" | timestamp}. That quantity would not be accepted by a different server. • If our bad guy Trudy can convince Bob to set his clock back, she can reuse encrypted timestamps she had overheard in what is now Bob's future. In practice there are systems that are vulnerable to an intruder resetting the clock. Although it might be obvious that a password file would be something that needed to be protected, if the security protocols are not completely understood, it might not be obvious that clock-setting could be a serious security vulnerability. Alice I'm Alice, KAlice-Bob{timestamp} Bob Protocol 9-3. Bob authenticates Alice based on synchronized clocks and a shared secret KAlice-Bob 226 SECURITY HANDSHAKE PITFALLS 9.1.2 • If security relies on time, then setting the time will be an operation that requires a security handshake. A handshake based on time will fail if the clocks are far apart. If there's a system with an incorrect time, then it will be impossible to log into the system in order to manage it (in order to correct its clock). A plausible solution to this is to have a different authentication handshake based on challenge/response (i.e., not dependent on time) for managing clock setting. In Protocol 9-1, computing f(KAlice-Bob,R). may be done with a secret key encryption scheme using KAlice-Bob as a key, or by concatenating KAlice-Bob with R and doing a hash. When we're using timestamps the same is true (a message digest works), except for a minor complication. How does Bob verify that hash(KAlice-Bob,R) is reasonable? Suppose the timestamp is in units of minutes, and the believable clock skew is 10 minutes. Then Bob would have to compute hash(KAlice-Bob, timestamp) for each of the twenty possible valid timestamps to verify the value Alice sends (though he could stop as soon as he found a match). With a reversible encryption function, all he had to do was decrypt the quantity received and see if the result was acceptable. While checking twenty values might have

acceptable performance, this approach would become intolerably inefficient if the clock granularity allows a lot more legal values within the clock skew. For instance, the timestamp might be in units of microseconds. There are 600 million valid timestamps within a five-minute clock skew. This would be unacceptably inefficient for Bob to verify. The solution (assuming you wanted to use a microsecond clock and a hash function rather than a reversible encryption scheme) is to have Alice transmit the actual timestamp unencrypted, in addition to transmitting the hashed value. So the protocol would be: 9.1.2 One-Way Public Key With protocols in the previous section, which are based on shared secrets, Trudy can impersonate Alice if she can read Bob's database. If the protocols are based on public key technology instead, this can be avoided, as in Protocol 9-5. In this case [R]Alice means that Alice signs R (i.e. transforms R using her private key). Bob will verify Alice's signature [R]Alice using Alice's public key, and accept the login if the result matches R. This is very similar to Protocol 9-1. The advantage of this protocol is that the database at Bob is no longer security-sensitive to an attacker reading it. Bob's database must be protected from unauthorized modification, but not from unauthorized disclosure. Alice I'm Alice, timestamp, hash(KAlice-Bob,timestamp) Bob Protocol 9-4. Bob authenticates Alice based on hashing a high-resolution time and a shared secret KAlice-Bob 9.1.2 LOGIN ONLY 227 Protocol 9-5. Bob authenticates Alice based on her public key signature And as before, the same minor variant works: In this variant, Bob chooses R, encrypts it using Alice's public key, and Alice proves she knows her private key by decrypting the received quantity to retrieve R. A problem with this variant is that some public key schemes (such as DSS) can only do signatures, not reversible encryption. So in those cases this variant cannot be used. In both Protocol 9-5 and Protocol 9-6 there is a potential serious problem. In Protocol 9-5 you can trick someone into signing something. That means, if you have a quantity on which you'd like to forge Alice's signature, you might be able to impersonate Bob's network address, wait for Alice to try to log in, and then give her the quantity as the challenge. She'll sign it, and now you know her signature on that quantity. Protocol 9-6 has Alice decrypting something. So, if there's some encrypted message someone sent to Alice and you're wondering what's in it, you might again impersonate Bob's address, wait for Alice to log in, and then have Alice decrypt it for you. How can we avoid getting in trouble? The general rule is that you should not use the same key for two different purposes unless the designs for all uses of the key are coordinated so that an attacker can't use one protocol to help break another. An example method of coordination is to ensure that R has some structure. For instance, if you sign different types of things (say an R in a challenge/response protocol versus an electronic mail message), each type of thing should have a structure so that it cannot be mistaken for another type of thing. For example, there might be a type field concatenated to the front of the quantity before signing, with different values for authentication challenge and mail message. Part of the purpose of the PKCS standards (see §6.3.6 Public-Key Cryptography Standard (PKCS)) is to impose enough structure to prevent this sort of problem when the same RSA key is used for different purposes. Alice I'm Alice Bob R [R]Alice Alice I'm Alice Bob {R}Alice R Protocol 9-6. Bob authenticates Alice if she can decrypt a message encrypted with her public key 228 SECURITY HANDSHAKE PITFALLS 9.2 Note the chilling implication—you can design several schemes where each is independently secure, but when you use more than one, you can have a problem. Perhaps even more chilling, you could design a new protocol whose deployment would compromise the security of existing schemes (if the new protocol used the same keys). 9.2 MUTUAL AUTHENTICATION Suppose we want to do mutual authentication, i.e. Alice will know for sure she is communicating with Bob. We could just do an authentication exchange in each direction: 9.2.1 Reflection Attack The first thing we might notice is that the protocol is inefficient. We can reduce the protocol down to three messages (instead of five used above) by putting more than

one item of information into each message: Alice I'm Alice Bob R1 f(KAlice-Bob,R1) R2 f(KAlice-Bob,R2) Protocol 9-7. Mutual authentication based on a shared secret KAlice-Bob Alice I'm Alice, R2 Bob R1, f(KAlice-Bob,R2) f(KAlice-Bob,R1) Protocol 9-8. Optimized mutual authentication based on a shared secret KAlice-Bob 9.2.1 MUTUAL AUTHENTICATION 229 This version of the protocol has a security pitfall known as the reflection attack. Suppose Trudy wants to impersonate Alice to Bob. First Trudy starts Protocol 9-8, but when she receives the challenge from Bob, she cannot proceed further, because she can't encrypt R1 . "I can't explain myself, I'm afraid sir," said Alice, "because I'm not myself, you see." Alice in Wonderland However, note that Trudy has managed to get Bob to encrypt R2. So at this point Trudy opens a second session to Bob. This time she uses R1 as the challenge to Bob: Trudy can't go any further with this session, because she can't encrypt R3. But now she knows KAlice-Bob{R1}, so she can complete the first session. This is a serious security flaw, and there are deployed protocols that contain this flaw. In many environments it is easy to exploit this, since it might be possible to open multiple simultaneous connections to the same server, or there might be multiple servers with the same secret for Alice (so Trudy can get a different server to compute f(KAlice-Bob,R1) so that she can impersonate Alice to Bob). We can foil the reflection attack if we are careful and understand the pitfalls. Here are two methods of fixing the protocol, both of which are derived from the general principle don't have Alice and Bob do exactly the same thing: • different keys—Have the key used to authenticate Alice be different from the key used to authenticate Bob. We could use two totally different keys shared by Alice and Bob at the cost of additional configuration and storage. Alternatively we could derive the key used for authenticating Bob from the key used to authenticate Alice. For instance, Bob's key might be −KAlice-Bob, or KAlice-Bob+1, or KAlice-Bob⊕F0F0F0F0F0F0F0F016. Any of these would foil Trudy I'm Alice, R2 R1, f(KAlice-Bob,R2) Bob Figure 9-9. Beginning of reflection attack Trudy I'm Alice, R1 R3, f(KAlice-Bob,R1) Bob Figure 9-10. Second session in reflection attack 230 SECURITY HANDSHAKE PITFALLS 9.2.2 Trudy in her attempt to impersonate Alice to Bob since she would not be able to get Bob to encrypt anything using Alice's key. • different challenges—Insist that the challenge from the initiator (Alice) look different from the challenge from the responder. For instance, we might require that the initiator challenge be an odd number and the responder challenge be an even number. Or the name of the party that created the challenge might be concatenated with the challenge before encryption, so that if the challenge from Alice to Bob was R, Bob would encrypt Bob|R (the string Bob concatenated with R). This would foil Trudy, since in order to impersonate Alice to Bob, Trudy would need to get Bob to encrypt the string Alice concatenated with some number. Notice that Protocol 9-7 did not suffer from the reflection attack. The reason is that it follows another good general principle of security protocol design: the initiator should be the first to prove its identity. Ideally, you shouldn't prove your identity until the other side does, but since that wouldn't work, the assumption is that the initiator is more likely to be the bad guy. ...if you only spoke when you were spoken to, and the other person always waited for you to begin, you see nobody would ever say anything... —Alice (in Through the Looking Glass) 9.2.2 Password Guessing Another security weakness of Protocol 9-8 (which doesn't exist in Protocol 9-7) is that Trudy can mount an off-line password-guessing attack without needing to eavesdrop. All she needs to do is send a message to Bob claiming to be Alice and enclosing a number to be encrypted, and Bob will obligingly return the encrypted value. Then Trudy has the pair ⟨R, f(KAlice-Bob,R)⟩ which she can use to check password guesses. We could fix that by making the protocol one message longer (Pro-tocol 9-11). Alice I'm Alice Bob R1 f(KAlice-Bob,R1), R2 f(KAlice-Bob,R2) Protocol 9-11. Less optimized mutual authentication based on a shared secret KAlice-Bob 9.2.3 MUTUAL AUTHENTICATION 231 Now Trudy can't obtain a quantity with which to do off-line password guessing by claiming to be

Alice, but she can by impersonating Bob's address and tricking Alice into attempting a connection to her. The threat of having Trudy impersonate Bob should not be ignored, but it is much more difficult than impersonating Alice. 9.2.3 Public Keys Mutual authentication can also be done with public key technology, assuming that Bob and Alice know each other's public keys. It can be done with three messages: A variant is for Alice to send R2 and for Bob to return it signed (and similarly for Alice to sign R1). Public key mutual authentication presents some special challenges. How does Alice (or Alice's workstation) know Bob's public key? Often the situation is that Alice is a human, working on a generic workstation. In such cases Alice isn't going to remember Bob's public key, nor is Alice's workstation likely to have it stored. It could be done by having Alice attach to something, hoping it's Bob, and having the thing she's talking to send its public key. But that would not be secure if Trudy is impersonating Bob. We also have the problem of having Alice's workstation obtain Alice's private key when all Alice knows is a password. It is generally straightforward to convert a password into a secret key, because most secret key algorithms will accept any value of the right size as a key. Some public key algorithms—notably RSA—have private keys that take special forms and cannot easily be derived from passwords. The usual method of dealing with this is to have Alice's workstation retrieve Alice's private key, encrypted with her password, from a directory service, or perhaps from Bob. It is not much more trouble to store, in the same place, information that would allow Alice to reliably learn Bob's public key. Two possible techniques: • Store Bob's public key encrypted with Alice's password. If anyone is impersonating Bob, they will not be able to give Alice a quantity encrypted with her password for which they'd know a corresponding private key. Alice I'm Alice, {R2}Bob Bob R2, {R1}Alice R1 Protocol 9-12. Mutual authentication with public keys 232 SECURITY HANDSHAKE PITFALLS 9.2.4 • Store a certificate for Bob's public key, signed with Alice's private key. Once her workstation obtains her private key, it can validate the certificate for Bob's public key. For more information on this, see Chapter 13 PKI (Public Key Infrastructure) and §10.4 Strong Password Credentials Download Protocols. 9.2.4 Timestamps We can reduce the mutual authentication down to two messages by using timestamps instead of random numbers for challenges: This two message variant is very useful because it is easy to add onto existing protocols (such as request/response protocols), since it does not add any additional messages. But it has to be done carefully. In the diagram we have Bob encrypt a timestamp later than Alice's timestamp. Obviously Bob can't send the same timestamp back to Alice, since that would hardly be mutual authentication. (Alice would be assured that she was either talking to Bob or someone smart enough to copy a field out of her request!) So in the exchange, Alice and Bob have to encrypt different timestamps, use different keys for encrypting the timestamp, concatenate their name to the timestamp before encrypting it, or use any other scheme that will cause them to be sending different things. And the issues involved with the one-way authentication done with timestamps apply here as well (time must not go backwards, they must remember values used within the clock skew, etc.). Note that any modification to the timestamp would do. The idea of timestamp+1 comes from Needham-Schroeder, where they have one side use the incremented challenge of the other. We use timestamp+1 in our example because that's what Kerberos V4 uses, but timestamp+1 is probably not the best choice. timestamp+1 has the potential problem that Trudy eavesdropping could use KAlice-Bob{timestamp+1} to impersonate Alice. A better choice would be a flag concatenated with the timestamp indicating whether the initiator or responder is transmitting. Although the threat of Trudy using KAlice-Bob{timestamp+1} can be avoided if Bob keeps both timestamp and timestamp+1 in his replay cache, in general it is poor security practice to use something like +1, where there isn't anything intrinsically different between what Bob does and what Alice does. Also, if service Bob consists of multiple replicas all with the same key, where it would be

difficult for a Alice I'm Alice, f(KAlice-Bob,timestamp) f(KAlice-Bob,timestamp +1) Bob Protocol 9-13. Mutual authentication based on synchronized clocks and a shared secret KAlice-Bob 9.3 INTEGRITY/ENCRYPTION FOR DATA 233 replica to keep track of timestamps used at other replicas, then the quantity Alice encrypts should include something unique to the replica she's talking to, such as its name or IP address. 9.3 INTEGRITY/ENCRYPTION FOR DATA In order to provide integrity protection and/or encryption of the data following the authentication exchange it is necessary for Alice and Bob to use cryptography to encrypt and/or add integrity checks to the data messages. We described several bases for the authentication exchange: • §9.1.1 Shared Secret—Alice and Bob share a secret key KAlice-Bob. • §9.2.3 Public Keys—Alice and Bob know each other's public keys, as well as their own private keys. • §9.1.2 One-Way Public Key—only one side has a public key pair; authentication is one-way. As we discussed in §7.8 Session Key Establishment, it is desirable for Alice and Bob to establish a shared secret per-conversation key (known as the session key) to be used for integrity protection and encryption, even if they already know enough long-term secrets to be able to encrypt and add integrity checks to messages. So we'll want to enhance the authentication exchange so that after the initial handshake both Alice and Bob will share a session key. It is important that an eavesdropper not be able to figure out what the session key is. Once a session key is established, the workstation can forget the user's password, or at least a piece of untrusted software can proceed with a cryptographically protected conversation without being told any long-term secrets. First we'll discuss how to establish a session key for each of the three cases above. Then we'll discuss how to use the session key for encryption and/or integrity protection. 9.3.1 Shared Secret Alice and Bob have a shared secret key KAlice-Bob. The authentication exchange is shown in Protocol 9-14. Perhaps mutual authentication was done, in which case there are two Rs, R1 and R2 . Perhaps authentication was done using timestamps instead of random Rs. At any rate, there is sufficient information in this protocol so that Alice and Bob can establish a shared session key at this point in the conversation. They can, for example, use (KAlice-Bob +1){R} as the session key. More generally, they can take the shared secret KAlice-Bob and modify it in some way, then encrypt the challenge R using the modified KAlice-Bob as the key, and use the result as the session key. 234 SECURITY HANDSHAKE PITFALLS 9.3.1 Why do they need to modify KAlice-Bob? Why can't they use KAlice-Bob{R} as the key? The reason they can't use KAlice-Bob{R} is that KAlice-Bob{R} is transmitted by Alice as the third message in the authentication handshake, so an eavesdropper would see that value, and it certainly would not be secure as a session key. How about using KAlice-Bob{R+1} as the session key? There's a more subtle reason why that isn't secure. Suppose Alice and Bob have started a conversation in which Bob used R as the challenge. Perhaps Trudy recorded the entire subsequent conversation, encrypted using KAlice-Bob{R+1}. Later, Trudy can impersonate Bob's network layer address to Alice, thereby tricking Alice into attempting to communicate with Trudy instead of Bob, and Trudy (pretending to be Bob) can send R+1 as a challenge, to which Alice will respond with KAlice-Bob{R+1}: Then Trudy will be able to decrypt the previous Alice-Bob conversation. So, Alice and Bob, after the authentication exchange, know KAlice-Bob and R, and there are many combinations of the two quantities that would be perfectly acceptable as a session key, but there are also some that are not acceptable as a session key. What makes a good session key? It must be different for each session, unguessable by an eavesdropper, and should not consist of a quantity X encrypted with KAlice-Bob, where X is a value that can be predicted or extracted by an intruder (as just discussed for X = R+1). See Homework Problem 3. Alice I'm Alice Bob R KAlice-Bob{R} Protocol 9-14. Authentication with shared secret Alice I'm Alice Trudy as Bob R+1 KAlice-Bob{R+1} Figure 9-15. Trudy impersonates Bob 9.3.2 INTEGRITY/ENCRYPTION FOR DATA 235 9.3.2 Two-Way Public Key

Based Authentication Suppose we are doing two-way authentication using public key technology, so that Alice and Bob know their own private keys and know each other's public keys. How can they establish a session key? We'll discuss various possibilities, with their relative security and performance strengths and weaknesses. 1. One side, say Alice, could choose a random number R, encrypt it with Bob's public key, and send {R}Bob to Bob attached to one of the messages in the authentication exchange. This scheme has a security flaw. Our intruder, Trudy, could hijack the conversation by picking her own R, encrypting it with Bob's public key, and send that to Bob (while impersonating Alice's network layer address) in place of the encrypted key supplied by Alice. 2. Alice could, in addition to encrypting R with Bob's public key, sign the result. So she'd send [{R}Bob]Alice to Bob. Bob would take the received quantity, first verify Alice's signature using Alice's public key, and then use his private key on the result to obtain R. If Trudy were to attempt the same trick as in scheme 1, namely choosing her own R and sending it to Bob, she wouldn't be able to forge Alice's signature on the encrypted R. This scheme is reasonable. It has a minor security weakness that can be fixed partially (in 3, below) or completely (in 4, below). The flaw is that if Trudy records the entire Alice-Bob conversation, and then later overruns Bob (i.e., manages to take over node Bob and learn all Bob's secrets), she will be able to decrypt the conversation she'd recorded. It seems like the kind of threat that only a very energetic paranoid would bother worrying about, but we'll show how to fix it. First note that if Alice is careful to forget R after terminating the conversation with Bob, then overrunning Alice will not enable Trudy to decrypt the recorded conversation, since Trudy needs Alice's public key and Bob's private key to retrieve the session key R. She can only get that by overrunning Bob. 3. This is like 2, above, but Alice picks R1 and Bob picks R2. Alice sends {R1}Bob to Bob. Bob sends {R2}Alice to Alice. The session key will be R1⊕R2. Overrunning Alice will enable Trudy to retrieve R2. Overrunning Bob will enable Trudy to retrieve R1. But in order to retrieve R1⊕R2, she'll need to overrun them both. In 2 we had Alice sign her quantity (i.e., she sent [{R}Bob]Alice instead of merely {R}Bob). Why isn't it necessary for Bob and Alice to sign their quantities here? Alice and Bob don't need to sign their quantities because although Trudy is perfectly capable of inserting her own {R1}Bob, she cannot decrypt {R2}Alice. Trudy might therefore be able to inject confusion into the system by having Bob think R1⊕R2 is a key he shares with Alice, but since Trudy only knows R1, she can't actually see any data intended for Alice. 236 SECURITY HANDSHAKE PITFALLS 9.3.3 4. Alice and Bob can do a Diffie-Hellman key establishment exchange (see §6.4 Diffie-Hellman), where each signs the quantity they are sending. In Diffie-Hellman, Alice chooses a random RA. Bob chooses a random RB. They have already agreed on public numbers g and p. Alice transmits gRA mod p. Bob transmits gRB mod p. They will use gRARB mod p as their session key. When we say they each sign the quantity they send, we mean that Alice doesn't simply transmit gRA mod p. She actually transmits [gRA mod p]Alice. And Bob actually transmits [gRB mod p]Bob. In this scheme, even if Trudy overruns both Alice and Bob, she won't be able to decrypt recorded conversations because she won't be able to deduce either RA or RB. 9.3.3 One-Way Public Key Based Authentication In some cases only one of the parties in the conversation has a public/private key pair. Commonly, as in the case of SSL, it is assumed that servers will have public keys, and clients will not bother obtaining keys and certificates. Cryptographic authentication is one-way. The protocol assures the client that she is talking to the right server Bob, but if Bob wants to authenticate Alice, after the cryptographic session is established, Alice will send a name and password. Here are some ways of establishing a shared session key in this case. 1. Alice could choose a random number R, encrypt it with Bob's public key, send {R}Bob to Bob, and R could be the session key. A weakness in this scheme is that if Trudy records the conversation and later overruns Bob, she can decrypt the conversation (since she can retrieve R once she steals

Alice's private key). 2. Bob and Alice could do a Diffie-Hellman exchange, where Bob signs his Diffie-Hellman quantity. Alice can't sign hers because she doesn't have a public key. This is slightly more secure than scheme 1 because Trudy can't later overrun Bob and retrieve the session key (assuming Bob has diligently forgotten the session key after terminating the conversation with Alice). Note that neither of these schemes assure Bob he's really talking to Alice, but in either scheme Bob is assured that the entire conversation is with a single party. 9.3.4 Privacy and Integrity In §4.3 Generating MACs we discuss various methods of computing a MAC with secret key cryptography. In §5.2.2 Computing a MAC with a Hash we discuss how to do it with a message digest function. As noted in §4.3.1 Ensuring Privacy and Integrity Together, there is currently no standard algorithm for providing both privacy and integrity with a single key and a single cryptographic pass 9.3.4 INTEGRITY/ENCRYPTION FOR DATA 237 over the data. But see §4.3.5 Offset Codebook Mode (OCB). Until a standard is adopted, plausible solutions are: develop two keys in the authentication exchange and do the two operations independently; make a second key by modifying the first (by changing a few bits in a predictable way); use different cryptographic algorithms so a common key is (presumably) irrelevant; or use a weak checksum for integrity inside a strong algorithm for privacy. The messages exchanged on a connection once the keys are known are likely to be in the form of discrete messages, where the authenticity of each must be determined before the conversation can proceed. Even if a message is authentic, it could be misinterpreted if played out of order. An attacker might, for example, record a message and then replay it later on in the exchange. This can be prevented by having all of the messages contain sequence numbers so that an out-of-order message is detected. Sequence numbers are used in both versions of Kerberos. Alternatively, the integrity code can be computed using not just the current message, but information about all previous messages, so that a replayed message will not be valid. This technique is used by Novell (see §21.1 NetWare V3). Another form of attack is reflection. Here the attacker records a message going in one direction and replays it in the other. If the same sequence number could be valid in both directions, such a message could be misinterpreted. This can be avoided by using sequence numbers in different ranges for the two directions, by having a DIRECTION BIT somewhere in the message, or by having the integrity code computed by some subtly different algorithm in the two directions. Sequence numbers have to be very large or you face the possibility of running out during a conversation. If you reuse sequence numbers during a conversation (i.e. while using the same session key), an attacker can replay an old recorded message when its sequence number recurs. This is an unlikely threat, but it is good form to prevent it. The best method is to change keys periodically during the conversation. (This also limits the amount of material a cryptanalyst can gather all encrypted under one key.) Changing keys in the middle of a conversation is known as key rollover. The simplest method of rolling over a key is to have one end choose a random key, encrypt it under the existing key, and send it to the other end. Because an attacker might cryptanalyze one key and use it to decrypt the subsequent keys, a stronger design would periodically repeat the authentication protocol or roll over keys by doing a Diffie-Hellman exchange to get the new key and integrity-protecting the public numbers with the old key. Sequence number maintenance and key rollover can be significantly complicated if encryption is done using a communications protocol that does not automatically retry transmissions after errors and put messages back in order before decrypting them. 238 SECURITY HANDSHAKE PITFALLS 9.4 9.4 MEDIATED AUTHENTICATION (WITH KDC) In Chapter 7 Overview of Authentication Systems, we discussed the concept of a KDC, which has a database consisting of keys for all users. Any user Alice registered with the KDC can securely communicate with the KDC. Alice and the KDC can authenticate each other and encrypt their communication to one another because they each know Alice's key: In the

above exchange, the KDC does not know whether it was really Alice that asked to talk to Bob. Some bad guy, Trudy, could send the message to the KDC saying I am Alice, I'd like to talk to Bob. But that won't do Trudy any good because she cannot decrypt the encrypted KAB. Trudy is causing Bob to get a spurious message that Alice wants to talk to him using key KAB, but it doesn't do any harm. The only parties that can know KAB after this exchange are the KDC, Bob, and Alice, so after this exchange Alice and Bob can mutually authenticate using key KAB (using a protocol such as discussed in §9.2 Mutual Authentication). Protocol 9-16 is not the way it is done in practice. That protocol has some practical problems, the most important of which is that if Alice immediately sends a message to Bob based on the new shared key, it's possible, given the vagaries of networks, for Alice's message to arrive first, in which case Bob would not know how to decrypt it. It also is a lot of trouble for the KDC to initiate a connection to Bob. Instead, since Alice is going to communicate with Bob anyway, the KDC gives Alice the information the KDC would have sent to Bob. The Kerberos protocol, which we'll describe in Chapter 11 Kerberos V4 and Chapter 12 Kerberos V5, refers to the encrypted piece of information that the KDC gives to Alice to pass along to Bob as a ticket to Bob. The ticket is infor-mation that will allow Alice to access Bob: Alice Alice wants Bob KDC Bob K invents key KAB Alice{use KAB for Bob} KBob{use KAB for Alice} Protocol 9-16. KDC operation (in principle) 9.4.1 MEDIATED AUTHENTICATION (WITH KDC) 239 Protocol 9-17 is incomplete. It has to be followed by a mutual authentication exchange between Bob and Alice in which they prove to each other that they know key KAB. But in §9.2 Mutual Authentication we've described how they can do that. 9.4.1 Needham-Schroeder A classic protocol for authentication using KDCs was designed by Needham and Schroeder. It is very similar to the protocol we just described, and it completes the exchange by doing the mutual authentication between Alice and Bob. [NEED78]. The Needham Schroeder protocol is important because it is a classic KDC-arbitrated authentication protocol and many others have been modeled after it. The Kerberos protocol is based on it, and it is instructive to understand its strengths and weaknesses. The paper gives a few variants, but the basic Needham-Schroeder protocol is shown as Protocol 9-18. Let's analyze this protocol. First we must apologize for the arcaneness of the analysis. It's unfortunately necessary to go into this depth in order to really understand a protocol. Without a thorough understanding, there might be subtle security flaws. The terminology nonce refers to a number that is used only once. A nonce could be a sequence number (provided that state is not lost during crashes), or a large random number. It also could be a timestamp, if you believe that your clock never goes backwards, but Needham and Schroeder specifically wanted to avoid dependence on timestamps (see §9.5 Nonce Types for more detail on nonces). Alice Alice wants Bob KDC Bob invents key KAB KAlice{use KAB for Bob} ticket to Bob = KBob{use KAB for Alice} "I'm Alice", ticket = KBob{use KAB for Alice} Protocol 9-17. KDC operation (in practice) 240 SECURITY HANDSHAKE PITFALLS 9.4.1 Message 1 tells the KDC that Alice wants to talk to Bob. The purpose of the nonce N1 is to assure Alice that she is really talking to the KDC. The (admittedly far-fetched) threat the protocol is trying to avoid is where Trudy has stolen an old key of Bob's, and stolen the message where Alice had previously requested a key for Bob. Bob, realizing Trudy has stolen his key, has changed his key. Trudy waits around until Alice makes a request to the KDC to talk to Bob. Trudy then replays the stolen message, which looks like an ordinary reply from the KDC. Then Trudy successfully impersonates Bob to Alice, because Trudy knows the key that Alice thinks the KDC just created for Alice and Bob. As we've already said, it is hard to imagine a circumstance where this would be a practical threat, but adding the nonce costs virtually nothing, and it removes the need to think about whether the threat might be practical in any particular circumstance. In message 2, the KDC securely (i.e. encrypted and integrity-protected) gives Alice the key KAB it has generated for Alice and Bob to share. It puts in

the string “Bob” to make it impossible for Trudy to tamper with Alice’s request, substituting the string “Trudy” for “Bob” and then being able to trick Alice into talking to Trudy and thinking that Trudy is Bob. If Trudy were to tamper with message 1 by substituting her name for Bob’s, then Alice will discover, in message 2, that the KDC has just given her a key to communicate with Trudy, not Bob. In message 2, along with the encrypted key KAB and Bob’s name, the KDC also gives Alice a ticket to Bob. The ticket to Bob consists of the key KAB and Alice's name, encrypted with Bob's key. To Alice, the ticket will just be a pile of unintelligible bits. The Needham-Schroeder protocol has been criticized for doubly encrypting the ticket. The ticket is sent encrypted with Alice's key, though it would have been just as secure to send it in the clear. In message 3, Alice sends a challenge (N2) to Bob, encrypted with KAB, along with the ticket. Bob decrypts the ticket to find the shared key KAB. He uses KAB to extract N2. Alice knows that only someone who knows Bob's key can decrypt the ticket and thereby discover the shared key KAB. In message 4, Bob proves he knows KAB, since he was able to find N2. Likewise, Bob 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”} 2 ticket, KAB{N2} 3 KAB{N2−1,N3} 4 KAB{N3−1} 5 Protocol 9-18. Needham-Schroeder 9.4.2 MEDIATED AUTHENTICATION (WITH KDC) 241 assumes that it must be Alice if she knows KAB because the KDC puts Alice’s name in the ticket with KAB. In message 4, in addition to sending back N2 (actually a decremented version of N2), he includes a challenge N3 encrypted with KAB, and Alice sends back (in message 5) a message proving she knows KAB. There is an interesting reflection attack to which the protocol would be vulnerable if the encryption in message 4 were done using, say, DES in ECB mode. Suppose each of the nonces were 64 bits long. Because of the nature of ECB, N2−1 and N3 would each be separately encrypted. Suppose Trudy wants to impersonate Alice to Bob. First she eavesdrops on an authentication handshake where Alice talks to Bob. So Trudy sees messages 3 and 4. Later, she replays message 3 to Bob. Bob will respond with KAB{N2−1,N4} (where N4 is a nonce different from the one Bob chose last time he talked to Alice). Trudy can’t return KAB{N4−1}, but she can open a new connection to Bob, this time splicing in KAB{N4} instead of KAB{N2}. Bob will return KAB{N4−1,N5}. Then Trudy can take the first encrypted block, which will be KAB{N4−1} and return that as message 5 of her first connection. In fact she will never have found out what N4 was, but she didn’t need to know N4. She just needed to know KAB{N4−1}. If encryption were done in CBC mode instead of ECB mode, Trudy would not be able to accomplish this, because she couldn’t splice pieces of messages. But if encryption were done in CBC mode, there would be no reason to have Bob and Alice decrement each other’s nonces. Message 4 could just as securely be KAB{N2,N3}, and message 5 could just as securely be KAB{N3}. The lesson to be learned is that if pieces of a message must, for security reasons, be sent together, then the entire message must be integrity-protected so that parts of it cannot be modified by an attacker. 9.4.2 Expanded Needham-Schroeder There is a remaining security vulnerability with Protocol 9-18. Suppose Trudy finds out Alice’s key. Trudy can of course, at that point, claim to be Alice and get the KDC to give Trudy (which the KDC thinks is Alice), a shared key to Bob and a ticket to Bob. There’s really nothing we can do about Trudy impersonating Alice if Trudy steals Alice’s key. But we’d like it to be possible to prevent Trudy from doing any more damage once Alice changes her key. The problem is, with the protocol we’ve just described, the ticket to Bob stays valid even if Alice changes her key. The vulnerability can occur even if Trudy only manages to capture a previous key used by Alice, say JAlice, and not the key Alice is using now. Suppose that when Alice was using JAlice, Trudy had overheard (and recorded) Alice asking the KDC for a ticket for Bob. At that time, the KDC would have generated a shared key JAB, and Trudy would have seen JAlice{N1,“Bob”, JAB,KBob{JAB,“Alice”}}. At the time, Trudy could not decrypt the message, but we’ll assume she’s stored it away, hoping to capture an old key of Alice’s. Once