password that will work on all servers the user is entitled to use. In the old scheme, a user could certainly use the same password on all servers, but when the user Alice changed her password, she'd have to remember to change her password at all the servers. The NT scheme works like the authentication facilitator node we discussed in §7.1.2 Storing User Passwords. The domain controller (which is what we called the authentication facilitator node), stores a secret for each server in the domain, which enables a server in the domain and the 21.7.1 MICROSOFT WINDOWS SECURITY 563 domain controller to communicate securely. The domain controller also stores security information for each human user in the domain, including a hash of the user's password, a list of groups to which the user belongs, when to prod the user to change her password, and the hours she's allowed to log in. When a user wishes to log into a server in the user's domain, that server sends a challenge to the user's workstation, which then generates a cryptographic response based on the user's password and the challenge (see Figure 21-8). Since the server does not store any security information for the user, it cannot evaluate the response. Instead, in an encrypted conversation to the domain controller, the server forwards its challenge and the workstation's response. The domain controller answers yes or no in the encrypted reply to the server. If the answer is yes, the domain controller also sends information about the user, such as to which groups the user belongs, and the user's hashed password. (This information, like all the sensitive information between the domain controller and the server, is encrypted.) The hashed password is transmitted so that the server can use it to compute a session key for protection of the remainder of the client-server conversation. It is possible for a client in one domain D1 to be authenticated by a server in another domain D2 provided that D1 and D2 have been preconfigured to trust each other. There is no transitivity, i.e., if Alice is in domain D1 and Bob is in domain D2, they can communicate if D1 and D2 have a trust relationship. It is not possible to transit through domain D3 in the case where D1 and D3 trust each other and D3 and D2 trust each other but D1 and D2 do not trust each other. This is similar to Kerberos V4. Alice Alice, pwd Alice's Workstation computes K = hash(pwd) I want Bob forgets pwd Alice Bob C computes R = K(C) R Domain Controller Knows KAlice = hash(passwordAlice) KBob{Alice,C,R} checks R = KAlice{C} KBob{yes,Alice's info} or KBob{no} Figure 21-8. Microsoft Windows NT Authentication 564 MORE SECURITY SYSTEMS 21.7.2 Names include a domain name, for example D1\Alice and D2\Bob. When Alice's workstation contacts Bob, it transmits Alice's name (D1\Alice). Bob contacts Bob's own domain controller, as in the single domain protocol. But Bob's domain controller can't make the decision since D2 has no information stored for D1\Alice. But D2 can communicate securely with D1, and so D2 forwards the information (D1\Alice, C, R) to D1, which replies to D2 with yes (plus group information, Alice's hashed password, and so on) or no, and then D2 relays this information to Bob (see Figure 21-9). The cryptographic aspects of the protocol are straightforward. RC4 is used to encrypt the conversation between the server and the domain controller. The domain controller has a long-term secret key stored for each server in the domain, and uses that to establish initial mutual authentication with the server, but as a side-effect of the authentication they establish a session key, and use that to encrypt subsequent communication. The session key for the conversation with the domain controller is obtained by adding the two challenges used in the mutual authentication and encrypting the result with the long-term secret. There is an authenticator in every packet that proves the message is in proper sequence, which prevents session hijacking, but is not used to provide integrity protection of the contents of the message. Encryption is only used on select fields in the packet, i.e., only where necessary. 21.7.2 Windows 2000 Kerberos With Windows 2000 came another step in the evolution of Windows authentication. User authentication information is moved from the Domain Controller to Active Directory, which serves as both KDC and directory service. The protocol is a modified

version of Kerberos V5. D1\Alice's Workstation D1\Alice D2\Bob C R Domain Controller D1 Domain Controller D2 KBob{D1\Alice,C,R} K1-2{D1\Alice,C,R} K1-2{yes,Alice's security info} or K1-2{no} KBob{yes, Alice's security info} or KBob{no} Figure 21-9. Microsoft Windows NT Interdomain Authentication 21.7.2 MICROSOFT WINDOWS SECURITY 565 As with OSF DCE, the designers at Microsoft were faced with the challenge that while Kerberos is designed to provide authentication of the names of communicating principals, what the OS needs for users is not a name but rather a UID (user ID) and a list of GIDs (group IDs), since that is the information used to make access control decisions. Otherwise there would need to be a separate translation mechanism available to every server. The Microsoft design is similar to the OSF DCE design, but simpler and more controversial. Both use the AUTHORIZATION-DATA field in the Kerberos ticket to hold the additional information. But unlike OSF DCE, which stood on its head creating a virtual Privilege Server entity and had many additional round trips when logging in users, Win2K Kerberos simply had the KDC add the AUTHORIZATION-DATA field to the ticket when it was generated. Technically, this did not comply with the Kerberos specification (though there have been legalistic arguments over whether it does or does not) while the OSF DCE approach did. But only lawyers worry about compliance. Users and system administrators worry about interoperability, and both OSF DCE and Windows 2000 Kerberos offer only backward compatibility with "standard" Kerberos applications. They offer none with one another nor can their services authenticate users not registered with their own registries. Another source of controversy is that the syntax of the authorization extensions is an "open secret". At the time of this writing, the design was posted at http://www.microsoft.com/technet/ security/kerberos/ and was downloadable by anyone, but required readers to acknowledge that the information is a trade secret of Microsoft, that they may only read it for purposes of reviewing its security, and that they may not discuss it with anyone who had not agreed to the same terms. Whether a company can retain trade secret rights to published information is an interesting legal question. In the court of public opinion, most observers are unwilling to call such a design "open". Another practical problem in Kerberos in the multi-realm case is that Kerberos doesn't specify how clients should find a sequence of realms that will allow them to authenticate to a remote server and doesn't specify how servers should decide whether a given sequence of realms is acceptable. This complex set of configuration issues is left to client and server implementations. In most cases, it would be more practical to configure this information in the KDCs. In the Windows 2000 Kerberos protocol, a client does not need to figure out the path to a remote service. It requests a service ticket from its local KDC, which instead returns a TGT to the next KDC in the chain. The client then repeats this process until it reaches the service's KDC, which gives it a service ticket. The KDCs should (and may) also check incoming requests to assure that the client has come through an acceptable sequence of KDCs on its way to the server and refuse to issue the next ticket if not. In this case, neither the client nor the server would need any knowledge of trust relationships. If authentication succeeds, it implies that the intermediate KDCs did all of the appropriate checks and it was OK. This extension has been proposed for standardization in the next revision of Kerberos. 566 MORE SECURITY SYSTEM 21.8 21.8 NETWORK DENIAL OF SERVICE The scheme in this section has not been commercially deployed, but it is being built on an experimental basis. It was devised by Radia Perlman and written up in [PERL88]. Routing in today's networks depends on the cooperation of all the routers. If a router were to generate confusing routing messages, or simply flood the network with enough garbage data to saturate the links, it can make a network inoperative. To some extent a network can be designed so that pieces are reasonably independent. For instance in a hierarchical network, although a rogue router in one piece can disable that piece of the network, if routing is properly designed the

disruption will remain in that one section of the network. Some routing protocols are being enhanced with authentication information to ensure that only routing messages from authorized routers are accepted. However, such a network would still be disabled by a saboteur corrupting a legitimate router. A fail-stop failure is one in which a computer is working perfectly one moment and then reverts instantaneously to halting. In real life, failures are not usually so civilized. Often a node starts acting erratically due to software failure, hardware failure, or misconfiguration. Without the help of saboteurs, many of today's networks have suffered collapse due to a sick node. It is only a matter of time before the vandals that have delighted in designing and deploying viruses expand their exploits into network sabotage. But it is possible to design a network that will continue to function even when being attacked by saboteurs. Our scheme will guarantee that two nodes will be able to converse provided that some path of properly functioning routers and links connects the two nodes. (If no such path exists, then there is no scheme that could possibly work.) So with this scheme, a network can operate properly even with several corrupted routes. The first part of the scheme involves robust broadcast, in which each message generated by a router is delivered to all the other routers. The robust broadcast mechanism will be used to distribute public key information for all the routers, so that routers need to start out knowing only a single public key. It will also be used to distribute routing information. The second part of the robust network design involves robust data packet delivery, in which a message generated by a node is delivered only to the specified destination. 21.8.1 Robust Broadcast Flooding is a simple routing mechanism in which a router that receives a message forwards the message on each link except the one from which the message was received. Flooding has the basic property we want—a message is guaranteed to get delivered as long as at least one properly functioning path exists—but with one serious problem. Flooding will only work if we assume that the 21.8.1 NETWORK DENIAL OF SERVICE 567 routers have infinite storage for buffered messages and the links have infinite bandwidth. Few networks have such characteristics, so simple flooding will not solve our problem. The basic idea behind robust broadcast is that each router guarantees a fraction of its memory and bandwidth resources to each other router. Each router keeps a database of messages to be forwarded and reserves a portion of the database, say one buffer, for each other router. The messages are transmitted over each link, round robin, so each router's message will have a chance to be delivered over each link. How do we ensure that the buffer reserved for a given router will be holding a message generated by that router? We do that with public key signatures. Each router signs each message it generates, and only a message with a valid signature can occupy a buffer. How do we ensure that it is the most recently generated message from a router that occupies the buffer? We do that by using a sequence number. Let's say router R1 receives a message with source R2, a valid signature, and a higher sequence number than the message stored for R2. Then R1 overwrites the message it had for R2. How does R1 know R2's public key, so that R1 can verify R2's signature? We could use a standard off-line CA. Each router would know the public key of the CA. Each router could include its certificate in each message. But that would make revocation difficult. Another possibility is to have the CA on-line, and have the CA generate and broadcast a list of nodes and public keys whenever the list changes. What happens if node R2 is malfunctioning and generates a message with the highest possible sequence number (assuming the sequence number is a finite-sized field)? In that case, R2 cannot generate any more messages until it gets a new public key and registers that new public key with the CA. At that point, R2 can start over again with sequence number 1. For efficiency, it would be nice to stop transmitting a particular message over a particular link once the neighbor on that link has received the message. This is done by marking each message in the database with an indication, for each neighbor,

specifying whether that message needs to be transmitted to that neighbor or whether an acknowledgment for that message needs to be transmitted to that neighbor. When a message (with valid signature and new sequence number) with source R2 is received by R1 from neighbor N, R1 stores the message in the database, marking it as needing to be acknowledged to N, and needing to be transmitted to all the other neighbors. If a duplicate message is received from neighbor N, then that message is marked as needing to be acknowledged to N (and not needing to be transmitted to N). If an acknowledgment for a message is received from neighbor N, that message is marked as not needing to be transmitted or acknowledged to N. When the link to N is available, the next marked message is transmitted, while the database is traversed in round-robin order. If the message is marked as needing to be transmitted, then the message is transmitted. If the message is marked as needing to be acknowledged, then an acknowledgment for that message is transmitted to N. The following table shows a database with messages from four different sources: R1, R2, R3, and R4. The node keeping this database has four neighbors: N1, N2, N3, and N4. Each message is 568 MORE SECURITY SYSTEMS 21.8.2 marked, for each neighbor, with transmit if the message needs to be transmitted to that neighbor, ack if an acknowledgment needs to be sent, or OK if neither needs to be sent. The database is scanned round robin per link, and messages are transmitted as marked. 21.8.2 Robust Packet Delivery Now we know how to robustly broadcast messages. We could use broadcast for delivery of data messages, but it would be inefficient. Instead we will use the robust broadcast mechanism to reliably deliver routing information, and use the routing information to compute routes. Once a route is computed, it is set up with a special cryptographically protected route setup packet, but then data packets can be forwarded without any cryptographic overhead. The type of routing protocol we will use is known as a link state protocol. In a link state protocol, each router is responsible for figuring out who its neighbors are and generating a packet known as a link state packet (LSP), which gives the identity of the source router and the list of neighbors of that router. Each LSP is broadcast to all the other routers, and each router is responsible for maintaining a database of the most recently generated LSP from each other router. Given this database, it is possible to efficiently compute routes. For details on routing protocols, see [PERL99]. Now assume that source S computes a path to destination D. Assume S is lucky enough to compute a properly functioning path (all the routers and links along the path are working properly). S transmits a special route setup packet, cryptographically signed and with a sequence number, that causes all the routers along the path to remember, for source/destination pair S/D, from which link they should expect to receive packets and to which link they should forward packets. The route setup packet has a sequence number, and a router R is only required to remember the highest numbered route from S to D. With that rule, we bound the maximum number of routes a router will need to maintain to n2, where n is the number of nodes. In practice, it would probably suffice to have a router remember some much smaller number of routes; if a router were asked to remember a route when it had run out of resources, it could complain and force the source to generate a route which did not include that router. In conventional route setup, a router only needs to remember the outbound link for a particular source/destination pair. We are requiring the router to also remember the link from which it should expect to receive packets for that source/destination pair. If a router checks to make sure that N1 N2 N3 N4 message source R1 transmit ack ack OK R2 OK OK OK OK R3 transmit transmit transmit ack R4 ack ack OK OK 21.9 CLIPPER 569 a packet was received from the proper link, then as long as the source was lucky enough to choose a properly functioning path, there is nothing a node off the path can do to disrupt communication on that path. And data packets do not need to be cryptographically protected. If S is not lucky enough to choose a correct path, it has a complete map of the

network, and can therefore choose an alternate path. There isn't any completely satisfactory way of doing this, since there are an exponential number of possible paths between any pair of nodes. But in practice, since there are unlikely to be more than one or two corrupted routers, a source can start getting suspicious of routers that appear on paths that don't work, and avoid them. And in the worst case, if a source were to try a lot of paths, it could revert to broadcasting the packet, which would guarantee delivery provided any path exists. We were vague about the difference between a node and a router. It is possible to only have the routers participate in the cryptographic protection, and have each router generate and sign route setup packets on behalf of the endnodes serviced by that router. 21.9 CLIPPER Telephones are easy to tap, and cellular telephones make eavesdropping even easier. Because of this, people have developed encrypting telephones, which so far have been too expensive to catch on, but today it is technically feasible to make encrypting telephones inexpensively. This makes the government nervous because criminals are sometimes convicted using evidence gathered through wiretaps. To fill the need for encryption without the U.S. government giving up the ability to wiretap (with legitimate reason and through a court order), the U.S. government has proposed the Clipper chip. The Clipper proposal offers high-grade encryption while preserving the ability of the U.S. government to wiretap. The politics of Clipper are at least as exciting as the technical aspects. We discuss the politics in §1.10 Key Escrow for Law Enforcement. Here we concentrate on the technical aspects. Technically the concept of Clipper is reasonably simple. Each Clipper chip manufactured contains a unique 80-bit key and a unique 32-bit ID. For each key K, an 80-bit random number K1 is selected, and then K2=K⊕K1 is computed. Neither quantity K1 nor K2 gives any information about K, but if you know both K1 and K2, you can compute K easily, since it is K1⊕K2. K1 (and the unique ID) is given to one federal agency, and K2 (and the unique ID) is given to a different federal agency. It hasn't been decided which agencies they'll be, but they should be agencies that everyone trusts, like the IRS, and that are independent of one another without likelihood of collusion, like the Army and the Navy. Let's say that Alice, using a telephone containing a Clipper chip, wants to talk to Bob, who has a similar device. Alice's chip has been registered with the two agencies (as has Bob's). Let's 570 MORE SECURITY SYSTEMS 21.9 say Alice's chip has unique ID IDA and secret key KA. With a court order, the government can obtain the two components of Alice's chip's key and then reconstruct KA. Without a court order, KA remains secret. What key will Alice and Bob use for communicating? It can't be KA or KB (Bob's chip's secret key) because neither side wants to reveal its secret key. So Alice and Bob use some mechanism, unspecified in the Clipper standard, to produce a shared secret key S. A reasonable choice is Diffie-Hellman (see §6.4 Diffie-Hellman), which was what was implemented in the Clipper phones. Alice feeds S to her Clipper chip and Bob feeds S to his Clipper chip. The chips use S to encrypt and decrypt the data. So where does KA come in, and how would the government, knowing KA, be able to decrypt the conversation? Also, how does the government know the unique ID of Alice's chip in order to obtain (with court order) KA? It would be an administrative nightmare for the U.S. government to try to keep track of who owns each Clipper chip. The information the government needs is in a field known as the LEAF, for Law Enforcement Access Field, that Alice's and Bob's Clipper chips transmit along with the encrypted data. Although Bob's chip does not utilize any of the information in the LEAF it receives from Alice's chip, Bob's chip refuses to communicate unless it receives a valid-looking LEAF. The central challenge in the design of Clipper is preventing someone from building a device that uses a Clipper chip for secure communication but substitutes garbage for the LEAF before transmitting the data. The Clipper design is very clever. The LEAF that Alice transmits to Bob contains IDA, KA{S}, and a checksum C. The field IDA enables the government to retrieve KA and then decrypt the field KA{S} to obtain S. Since

the same key S is used in both directions, the government only needs one of the keys KA or KB in order to decrypt the entire conversation. How does Bob's chip know whether the LEAF is valid? Bob's chip can't know whether IDA is the correct value and it can't decrypt KA{S} to check if it's the correct encrypted key. Bob's chip makes its decision based on the value of the field C. C is basically some sort of message digest of the other fields in the LEAF (IDA and KA{S}) and the key S. Bob's chip computes the message digest of the values in the received LEAF for IDA and KA{S}, along with S (which Bob's chip knows). If the computed checksum matches the received C, then the entire LEAF is assumed to be valid. The message digest algorithm used to compute C need not be secret, so what's to prevent someone from foiling the government's ability to wiretap by modifying the quantity KA{S} (or IDA) and sending the matching C? To prevent modification of the LEAF, all Clipper chips share an 80-bit secret family key, F. The quantity IDA|KA{S}|C is encrypted with F. Since Bob's chip knows F, it can decrypt the LEAF to extract IDA, KA{S}, and C. It can't verify that either IDA or KA{S} is correct, but knowing S, it can compute C. If the C in the LEAF doesn't match the computed C, the chip will refuse to decrypt the conversation. With the LEAF thus guaranteed, the government can (with a valid court order) tap Alice's line, read IDA from the LEAF, and then retrieve KA. Then it can read KA{S} from the LEAF and decrypt it to obtain S. At this point, it can decrypt the recorded conversation. 21.9 CLIPPER 571 It is rather astonishing that all Clipper chips will know the value F, and yet the expectation is that the value will remain secret. Clipper chips are carefully manufactured so that it should not be possible, by taking one apart, to obtain F. The encryption algorithm (SKIPJACK) was originally intended to be kept secret, and the same technology that prevents someone from reverse-engineering the encryption algorithm would protect F. But eventually the government decided to declassify SKIPJACK. How much of a disaster would it be if someone discovered F? It would not make Clipper-protected conversations less secure. However, it would mean someone could build a device that interoperated with Clipper devices and foiled wiretapping by generating garbage for IDA|KA{S}, computing the proper value for C based on the garbage and the session key S, and encrypting it all with F. It has been pointed out that people could, with less effort, build a device that encrypted the data before transmitting it to the Clipper chip. Provided the receiver had a compatible device (Clipper plus the extra encryption device), the output stream would look like normal Clipper output, until someone, under court order, decrypted the conversation and realized they were obtaining ciphertext. If you had a device capable of encrypting conversations, why would you bother going through the extra step of sending the encrypted stream through a Clipper chip? Perhaps you don't have complete faith in your cryptographic algorithm. Another reason surfaces if it becomes illegal or suspicious to use non-Clipper encryption. If you send your encrypted stream through a Clipper chip, the government would not know that you were using your own encryption, until after it obtained the court order and decrypted your conversation. One wonders what penalties could be associated with using non-Clipper encryption that would deter the sorts of people who are worried about being wiretapped. Prosecutor: Did you murder your wife? Defendant: No. Prosecutor: Do you know the penalty for perjury? Defendant: No, but I bet it's less than the penalty for murder! — classic joke If someone discovered F and built a Clipper-compatible device, then Alice, with that device, could foil wiretapping while talking to Bob, who had an ordinary Clipper device. If the government had Alice under suspicion, and therefore had obtained her Clipper key, it would still need to get a court order for Bob's Clipper key before it could decrypt the Alice-Bob conversation. If both Alice and Bob had one of these Clipper-compatible devices, then the government would discover, only after court order, that Alice and Bob were not using real Clipper devices, just as it would discover if Alice and Bob were using an extra encryption step before the Clipper chip. 572 MORE SECURITY SYSTEMS 21.9.1 Matt Blaze [BLAZ94] discovered

an interesting property of the Clipper design that makes it possible for Alice, with a lot of effort, to forge a LEAF in a way that Bob will accept, but will prevent government wiretaps. The problem is that the quantity C is only 16 bits long. Remember that Bob's chip cannot verify that either of the quantities IDA or KA{S} is correct. All the chip can do is verify that C is based on those quantities and the proper key S. The LEAF is 128 bits long, consisting of 32 bits of ID, 80 bits of encrypted key, and 16 bits of checksum. Bob's chip will accept any sequence of bits for the ID and the encrypted key as long as those fields, plus the Alice-Bob session key, produce the correct 16-bit checksum. The implication of this is that if Alice were to send a 128-bit random number, it would have 1 chance in 216 of having the correct checksum, when decrypted with the family key F. Now we get to Matt Blaze's attack. Alice can use any Clipper chip as a tester. She tells it she wants to converse using the key she's agreed upon with Bob. Then she feeds it random 128-bit LEAF values. On average she'll only have to try 215 random numbers before one will wind up with the correct checksum. Now that she knows a LEAF value that will be accepted by Bob's chip for the key she's sharing with Bob, she needs a special piece of hardware that takes the output stream from her Clipper device, removes the LEAF values it transmits, and substitutes the one she found. Bob's device won't know there's a problem, and the problem will only be detected when the government attempts to wiretap. Matt Blaze's attack is not important in practice, since it would either take a lot of parallel hardware or cause an unacceptable delay in conversation startup, but the discovery was useful in embarrassing the designers for having missed such an "obvious" flaw. 21.9.1 Key Escrow The Clipper chip has become synonymous with key escrow in the public debate, and has given key escrow a bad name. But there are good reasons other than keeping Big Brother employed for wanting to back up keys. It's certainly important to keep secret the key with which you encrypt your data. It's possible to be really careful about that. It could be that the only copy of your key is on your smart card, and the smart card is carefully engineered so that it is impossible to obtain the key from the smart card. Then one day your three-year-old finds it and flushes it down the toilet. Is it OK with you to accept that all your work is irretrievably gone? Maybe not. A similar problem occurs with house keys. When you arrive home during a blizzard, and discover you've lost your key, you really don't want to sit outside until some other member of your family gets home. Typically you've hidden a key under the WELCOME mat (which is equivalent to having a key hung from a chain from the doorknob), or you've given a copy of the key to a neighbor, who hopefully won't break in but will be home when you need the key. 21.10 HOMEWORK 573 What can you do with your cryptographic key to ensure you can retrieve the key if you forget or lose the key? You could write it on a piece of paper and stick it in an obscure drawer at home, or you could lock it in your safe deposit box. There are various mechanized approaches that don't involve obsolete technology like paper and safe deposit boxes. One possibility is to store your key on some sort of server. If you're paranoid someone will break into that server you could break your key into pieces, like the Clipper proposal does, where all the pieces need to be $\oplus$'d in order to recover the key. Then you could store each piece on an independent server. Contacting all the servers to give them their pieces of your key might be some trouble. An elegant solution is to take each piece, encrypt it with the public key of the server to which you'd like to give that piece, and store it on your own machine. Then if you lose the key you can give the corresponding quantities to the corresponding servers and recover it. 21.10 HOMEWORK 1. In §21.4.5 DASS Delegation we describe how Alice manages to simultaneously establish a session key with Bob, pass along her login private key to Bob, and authenticate to Bob. Analyze the protocol as described. Why can't an eavesdropper discover Alice's login private key? Why can Bob discover Alice's login private key? How can Bob be sure that the authenticator came from Alice? In the non-delegation case, he knows that only Alice knows the session key (that

encrypted the authenticator) because the session key, when passed to Bob, was signed with Alice's private key. But the signing step is omitted in the delegation case. 2. Propose an alternative design for Clipper where each chip knows only a public key and the government holds the corresponding private key, and where each party sends the session key encrypted under the public key as part of its transmission. How might Bob verify that the LEAF in your design is correct? What would the advantages and disadvantages of this scheme be over Clipper? 3. As described in §21.2.1 NetWare's Guillou-Quisquater Authentication Scheme, a GQ key, as NetWare uses it, expires. Describe how, using one of the strong credential download schemes in §10.4 Strong Password Credentials Download Protocols, and using only RSA keys, a workstation can create a temporary authentication key and forget the user's long-term key. 4. With the scheme described in §21.5.2 Coping with Export Controls, 24 bits of the 64-bit key are made accessible to the U.S. government. So is this scheme more secure than the traditional exportable scheme that uses 40-bit keys? 574 MORE SECURITY SYSTEMS 21.10 5. Why aren't the escrowed keys for Clipper indexed by the name of the person whose phone will be tapped, or the telephone number to be tapped, rather than the ID of the device? 575 22 FOLKLORE Whenever I made a roast, I always started off by cutting off the ends, just like I'd seen my grandmother do. Someone once asked me why I did it, and I realized I had no idea. It had never occurred to me to wonder. It was just the way it was done. Eventually I remembered to ask my grandmother. "Why do you always cut off the ends of a roast?" She answered "Because my pan is small, and otherwise the roasts would not fit." —anonymous Many things have become accepted security practice. Most of these are to avoid problems that could be avoided other ways if you really knew what you were doing. It's fine to get in the habit of doing these things, but it would be nice to know at least why you're doing them. A lot of these issues have been discussed throughout the book, but we summarize them here. 22.1 PERFECT FORWARD SECRECY Perfect forward secrecy (PFS) (see also §14.3 Perfect Forward Secrecy) is a protocol property that prevents someone who records an encrypted conversation from being able to later decrypt the conversation, even if they have since learned the long-term cryptographic secrets of each side. A protocol designed with PFS cannot be deciphered by a passive attacker, even one with knowledge of the long-term keys, though an active attacker with knowledge of the long-term key can impersonate one of the parties or act as a man-in-the-middle. PFS is considered an important protocol property to keep the conversation secret from: • an escrow agent who knows the long-term key • a thief who has stolen the long-term key of one of the parties without this compromise being detected • someone who records the conversation and later manages to steal the long-term key 576 FOLKLORE 22.2 • law enforcement that has recorded the conversation and subsequently, through a court order, obtains the long-term key PFS isn't free. There are two ways to do it. One is what is done in IKE, which is to do a Diffie-Hellman exchange, and have both sides forget the Diffie-Hellman information after the conversation concludes. Diffie-Hellman is computationally more expensive than would be necessary for setting up a session key without worrying about PFS. The other method is what is done in SSL/TLS, which is to have one side, say Bob, generate an ephemeral public/private key pair, and have the other side, say Alice, send a random number encrypted with the ephemeral public key. This is less expensive than Diffie-Hellman for Alice, but much more expensive for Bob. To get "perfect" PFS, Bob would have to forget the ephemeral private key after the first session created with that ephemeral key concludes. Generating a new ephemeral key pair is much more expensive than doing a Diffie-Hellman exchange. But to get "pretty good" forward secrecy, it would be fine for Bob to only generate a new ephemeral key pair every few hours or so. Therefore, the cost of creating the key pair would be amortized over many sessions. But even if Bob doesn't need to generate a new ephemeral public key pair for each session, he does

have to do a private key operation (decryption). In addition to the performance costs of providing for PFS, PFS makes getting export approval more difficult. 22.2 CHANGE KEYS PERIODICALLY With many encryption schemes, the more examples of ciphertext you can see, the more likely it is you will be able to break the encryption and find the key. With an adequately strong encryption scheme, this shouldn't be a problem. But to be on the safe side, people like to change keys (do key rollover) before any key has been used on more than some amount of data. For instance, when encrypting in CBC mode, it is desirable to change keys before it is likely that two cipherblocks will be equal. For example, with 64-bit blocks, by the birthday problem (see §5.1 Introduction) it is likely that after 232 blocks, two ciphertext blocks will be equal. (What information would this leak? See Homework Problem 1.) Another reason to do key rollover (with perfect forward secrecy) is in case the data encryption key were stolen without your knowledge in the middle of the conversation. Yet another reason to rollover keys is when export rules require use of a key which is too small to really be secure. For instance, the SKIP protocol allowed changing the key on every packet by sending with each packet the data encryption key (of inadequate strength) encrypted with a key which was of adequate strength. With this approach, even if 40-bit data encryption keys were being 22.3 MULTIPLEXING FLOWS OVER A SINGLE SA 577 used, if 220 packets were sent during a conversation, it would take up to 260 operations to break all the keys for the entire conversation. 22.3 MULTIPLEXING FLOWS OVER A SINGLE SA Folklore says that different conversations should not be multiplexed over the same security association. For instance, suppose two machines, F1 and F2, are talking with an IPsec SA, and forwarding traffic between A and B and between C and D (where A and C are behind F1 and B and D are behind F2). F1 and F2 might be firewalls, with A, B, C, and D machines behind those firewalls. Or A and C might be processes on F1 and B and D might be processes on F2. Some people would advocate creating two SAs between F1 and F2, one for the A-B traffic, and one for the C-D traffic. Creating multiple SAs is obviously more expensive in terms of state and computation. What are the reasons people advocate F1 and F2 creating multiple SAs rather than sending all the traffic they are forwarding to each other over a single SA? 22.3.1 The Splicing Attack This attack is only relevant if F1 and F2 are doing encryption-only (no integrity protection). Suppose different conversations are multiplexed over this tunnel. Suppose C (or an accomplice) is capable of eavesdropping on the F1-F2 link, as well as injecting packets. It is possible for C to do a splicing attack and see the decrypted data for the A-B conversation. Assume that the beginning of the plaintext, say the first 16 octets, identifies the conversation to F2, most likely by specifying the source and destination. The splicing attack involves the following steps: • record an encrypted packet from the C-D conversation • record an encrypted packet from the A-B conversation • overwrite the first 16 octets of ciphertext from the C-D packet onto the first 16 octets of the encrypted A-B packet • inject the spliced packet into the ciphertext stream When F2 decrypts the packet, it will observe from the first 16 octets that the packet should be delivered to D. The remainder of the packet is the data from the A-B conversation. If it were encrypted in ECB mode, all of the data from the A-B plaintext packet will be delivered to D. But if it were encrypted in, say, CBC mode, the first block of data will be garbled, but the remainder will be the plaintext from the A-B conversation. (See Homework Problem 4.) 578 FOLKLORE 22.3.2 Note that this flaw is only relevant if the F1-F2 link does encryption-only. If it has cryptographic integrity protection, this attack is not possible. However, since people found the flaw with encryption-only, some people think it would be safer to use separate SAs, just in case a similar flaw is found when integrity is also used. Even if there is no security flaw due to multiplexing traffic from different conversations over an encrypted and integrity protected tunnel, if F1 is a machine at an ISP (Internet Service Provider) serving multiple customers, the customers often feel safer if their traffic is carried over its own SA.

Since the ISP needs the customers, it is advantageous for it to make the customers feel safer. 22.3.2 Service Classes Suppose some types of traffic being forwarded between F1 and F2 get expedited service, or some different routing that would make it likely for packets transmitted by F1 to get widely out of order. If F2 is protecting against replays based on a sequence number in the packet, a common implementation of this is for F2 to remember the highest sequence number seen so far, say n, and remember which sequence numbers in the range n-k to n-1 have already been seen. If F1 marks the packets with different classes of service, then packets with a lower priority might take a lot longer to arrive at F2. If more than k high priority packets launched by F1 can arrive before a lower priority packet launched earlier by F1, then the low priority packet will be discarded by F2 as out of the window. For this reason, some people advocate creating a different SA for each class of service, so that each service class has sequence numbers from a different space. 22.3.3 Different Cryptographic Algorithms Some people advocate different SAs for different flows because each flow might require a different level of security. Some flows might require integrity-only. Some encrypted traffic might require more security, and thus a longer key, than other traffic. One might think this could be solved by using the highest level of security for all traffic. But the higher security encryption might be a performance problem if it was used for all the traffic. Another reason to use different cryptographic algorithms is if some of the flows are for customers who would, for their own reasons, like to use particular cryptographic algorithms. It might be vanity crypto developed by their own company or country, they might feel safer using different algorithms than what others use because they might have heard (different) rumors about potential weaknesses, or there might be legal reasons why particular algorithms might only be usable for traffic of certain customers. 22.4 USE DIFFERENT KEYS IN THE TWO DIRECTIONS 579 22.4 USE DIFFERENT KEYS IN THE TWO DIRECTIONS This avoids reflection attacks such as in §11.2.1 Reflection Attack. There are other ways of avoiding reflection attacks, for instance: • have the initiator generate odd challenges, and the responder even challenges • have the response to the challenge consist of a function of the challenged side's name in addition to the key and the challenge, e.g., h(name, key, challenge) or key{name | challenge} If it's inconvenient to obtain large random numbers, a unique nonce such as a sequence number can be turned into an unpredictable nonce by hashing it with a secret known only by the sender. 22.5 USE DIFFERENT SECRET KEYS FOR ENCRYPTION VS. INTEGRITY PROTECTION With CBC residue as an integrity check there are problems when the same key is used for encryption as well as integrity protection, as described in section §4.3.1 Ensuring Privacy and Integrity Together. But if the integrity protection were, say, keyed MD, then there are no known weaknesses of using the same key. However, people are worried that since there was a problem with using the same key for both purposes with CBC residue, perhaps someone will later find a weakness with other schemes, and using two different keys avoids the issue. Another reason using two keys became popular in protocols was because of U.S. export laws. The allowed key-length for exportable encryption was 40 bits, which was definitely very weak. But the U.S. government did allow stronger integrity checks. They only wanted to read the data, not tamper with it, so it was okay with them if the integrity check were reasonably strong. As a result, protocols often wound up with two keys: a 40-bit one for encryption, and one of adequate size for integrity protection. PEM, when calculating a CBC residue as an integrity check, used a variant of the encryption key for calculating the CBC residue (the encryption key $\oplus$'d with F0F0F0F0F0F0F0F016). Unfortunately, CBC residue is completely insecure as an integrity check for email (see §18.16 DES-CBC as MIC Doesn't Work). The problem isn't a subtle interaction when using a variant of the same key for encryption as integrity protection. The problem is that CBC residue does not have the property that a cryptographic hash has; it is easy to create an arbitrary message with a

given CBC residue. Using different keys for integrity and encryption has the disadvantage that it requires two cryptographic passes over the data (twice the computation). For many years people yearned for an 580 FOLKLORE 22.6 algorithm that would be able to do a single cryptographic pass, simultaneously encrypting and calculating an integrity check on the data. There were several proposed, and then later found to be broken. There are a few that have been proposed recently, and not (yet?) found to be broken. 22.6 USE DIFFERENT KEYS FOR DIFFERENT PURPOSES If the same key is used for, say, decrypting a challenge, and for decrypting data encryption keys in headers of encrypted messages, it is possible that someone could get tricked into decrypting or signing something. An extreme example is of a blind signature (see §13.8.5 Anonymous Groups). By definition, with a blind signature the signer has no idea what he's signing. So that key had better not be used for any other purposes, such as signing purchase orders. But even when the application isn't purposefully designed to prevent the key owner from knowing what he's doing, it is very likely that one application might not recognize an action that would have meaning in another application. For example, if the private decryption key is used to decrypt a challenge in a challenge/response protocol, the "challenge" given could be an encrypted data encryption key extracted from an email header. One method of preventing such cross-application confusion is to encode something application-specific in the padding of the information to be signed or encrypted using public key cryptography. So in a challenge/response protocol in which someone would be asked to sign the challenge, the challenge could be a 128-bit number, and the signer would pad the challenge, before signing, with an application-specific constant such as a hash of the string "authentication protocol QXV challenge-response". Likewise, a signature on, say, a PGP message could be defined as containing the hash of the string "PGP message signature". If all applications using that same key were carefully coordinated, no confusion would arise. However, if any application didn't insist on those rules, there could be vulnerabilities. Therefore it would be safest, in theory, to certify a key as only applicable for a given application. The downside of this is that maintenance of so many different keys could present its own vulnerabilities as well as performance issues. 22.7 USE DIFFERENT KEYS FOR SIGNING VS. ENCRYPTION If a signature private key is forgotten, it's not a big deal. You can just generate a new one. But if an encryption private key is lost, you'd lose all the data encrypted with it. Therefore, it is common to err on the side of making it hard to lose the encryption key, by keeping extra copies of the private 22.8 HAVE BOTH SIDES CONTRIBUTE TO THE MASTER KEY 581 key, perhaps in lots of different places. But this makes the key easier to steal, a trade-off considered acceptable in some cases for an encryption key, given the high cost of losing all the encrypted data. But in most cases there's no advantage in keeping copies of a signature private key. It's better to make it less vulnerable to theft by not keeping copies. Another reason for separate keys is for law enforcement or for a company that wants to be able to decrypt anything encrypted for any of its employees on any of its equipment. They would like the private encryption key readily accessible to people other than the person associated with that key. But they have no need to be able to forge something, so they do not need to have a copy of the person's signature private key. In fact, it's to their advantage not to have access to the signature key. Then the person cannot repudiate his signature on something, for instance, claiming that the death threat signed by his key was an attempt by law enforcement to frame him. Yet another reason is that you'd like to treat old expired encryption keys differently than old signature keys. There's no reason to archive old private signature keys. Once the key is changed, just throw away the old one. Irretrievably discarding a private signature key once it isn't supposed to be used anymore safeguards against someone stealing the old key and backdating a document. But old encryption keys are likely to still be needed in order to decrypt data encrypted with them. 22.8 HAVE BOTH SIDES

CONTRIBUTE TO THE MASTER KEY It is considered good cryptographic form for both sides of a communication to contribute to a key. For instance, consider the protocol in which Alice and Bob each choose a random number, send it to the other side encrypted with the other side's public key, and use a hash of the two values as the shared secret. Although this protocol does not have PFS, it has "better" forward secrecy than something like SSL/TLS, because only by learning both side's private keys can you decrypt the recorded conversation. Another reason for having both sides contribute to the master key is that if either side has a good random number, the result will be a good random number. 22.9 DON'T LET ONE SIDE DETERMINE THE KEY In addition to ensuring that each side contributes to the key, folklore says it is a good idea to ensure that neither side can force the key to be any particular value. For instance, if each side sends a random number encrypted with the other side's public key, if the shared key were the $\oplus$ of the two values, then it would be easy for whichever side sent the last value to ensure that the result would be a 582 FOLKLORE 22.10 particular value. If instead the shared key were a cryptographic hash of the two values, this would be impossible. Why is this an important property? In most cases it wouldn't matter. However, here is a subtle example where it would matter. Suppose the world were to do IPsec, but without any per-machine secrets. The purpose of IPsec would be to efficiently get encryption and integrity protection without the hassle of distributing keys. Unfortunately, without per-machine keys, there is no way to detect men-in-the-middle. But we'll assume that applications have keys, and can authenticate each other. We want to be able to authenticate and detect men-in-the-middle using the application keys, but once the applications know to whom they are talking, and are assured that there are no men-in-themiddle, we have the applications depend on the cryptographic protection provided by IPsec, since that will be more efficient than providing cryptographic protection at the application layer. So suppose application Bob is on machine B which is supposed to be at IP address x. Machine C (owned by, say, Chaos Computer Club), impersonates IP address x and waits for someone to connect. Application Alice is on machine A and attempts to communicate with IP address x. IPsec establishes a tunnel to IP address x, which will be machine C. Machine C establishes a tunnel to machine B. This is a classic man-in-the-middle scenario of C interposing itself between A and B. A way of detecting C is for the applications Alice and Bob to authenticate each other over the tunnel, using their secrets for a true end-to-end handshake. Since C can simply relay the handshake messages, the handshake would not automatically detect C. However, if machine A gave Alice a hash of the shared key (call it H), and machine B gave Bob a hash of the shared key, then they could use H in the handshake. For instance, if they have public signature keys, they could each sign H. If C were able to force the key to be the same between itself and A and between itself and B, then this protocol would fail to detect C, since C could just forward the signed H. 22.10 HASH IN A CONSTANT WHEN HASHING A PASSWORD Users tend to use the same password in multiple places. Suppose there is a protocol in which a secret key is derived from the user's password and used in a challenge-response protocol. (W=h(pwd), Bob sends challenge R, Alice sends response f(W,R)). W is a password-equivalent, because if someone knows W they can directly impersonate the user, even though W is not the string that the user types. In such a protocol, the server will contain a database of each user's password-equivalent value. This database might be specific to a server, to one application, or to a machine that stores password-equivalents for all users in a domain. To prevent theft of one database from allowing someone to impersonate users using the same password on a different server (or different application, or different domain), have the server hash a constant into the password hash, such as the server 22.11 HMAC RATHER THAN SIMPLE MD 583 name. In that way, a user with the password "albacruft" on server 1 would have a different hashed password on server 2, even if she used the same password on both. If someone were to steal server 1's

database, they could impersonate her on server 1. But since the hash would be different on server 2, someone would not be able to impersonate her on server 2, unless they did a dictionary attack and discovered her actual password. 22.11 HMAC RATHER THAN SIMPLE MD Folklore today says that in order to do a keyed cryptographic hash, you should use HMAC (see §5.7 HMAC). It is computationally slower than, say, doing hash(message|key). But it is possible, as we explain in §5.2.2 Computing a MAC with a Hash, to do the keyed hash incorrectly. If you do hash(key|message) and divulge the entire hash, then it is possible for someone who sees the message and the entire keyed hash to append to the message and generate a new keyed hash, even without knowing the key with which the hash is produced. With HMAC this attack is impossible. There are other ways of avoiding that attack that are simpler and faster and likely to be as secure, for instance only divulging half the hash (see §5.2.2 Computing a MAC with a Hash). But as described in §5.7 HMAC, HMAC comes with a proof, so it is gaining popularity despite its performance disadvantage. 22.12 KEY EXPANSION Key expansion is the technique of using a small number of random bits as a seed and from it deriving lots of bits of keys. For instance, from a 128-bit random seed, you might want to derive 128-bit encryption keys and 128-bit integrity keys in each direction. Or you might want to periodically do key rollover (without PFS) lots of times, using the same seed. The alternative to key expansion is obtaining independent random numbers for each key. It might be expensive to obtain that many random bits. For instance, if the random number is sent encrypted with the other side's public key, it is convenient to limit the size of the random number to fit within an RSA block. And if an unpredictably many future keys also need to be derived from the same keying material, you wouldn't know how large a random number you'd need. If the random number is sufficiently large (say 80 bits or more), then it can be used as the seed for an unlimited number of keys. It is important to do this in such a way that divulging one key 584 FOLKLORE 22.13 does not compromise other keys. So each key is usually derived from the random number seed and some other information (such as a key version number or timestamp), using a one-way function. The random number from which all other keys are derived is often known as the master key or key seed. In many protocols, creating and/or communicating the initial secret is expensive. For instance, in SSL/TLS it involves sending something encrypted with the other side's public key, and decrypting it will therefore be expensive. Another example is IPsec, in which creating the initial secret involves a Diffie-Hellman exchange, which is expensive for both sides. Reusing the original secret for generating new keys will be less expensive than creating and sending a new random seed. In the cryptography community, companies with outrageous claims and bogus security products are known as "snake-oil" companies. A common claim made by the snake-oil marketeers is that the company's algorithm uses keys of some huge size, perhaps millions of bits, and that that makes them much more secure than, say, the 128-bit keys used by other companies. (And they usually have a "patented", proprietary encryption scheme.) There are many aspects of such a statement that should raise the suspicion that the claims are snake-oil. Often the million-bit key is generated from a very small seed, perhaps 32 bits, which through key expansion turns into many bits. So even though the key used might be a million bits, if the seed from which it is computed is 32 bits, then the key itself is equivalent to a 32-bit key. 22.13 RANDOMLY CHOSEN IVS Folklore says that the IV should not be something like a sequence number in which there would be a very small hamming distance (the number of bit positions in which one IV differs from the other) between IVs. The reason is that the IV gets $\oplus$'d with the first block of plaintext, and then encrypted. It is common for the first block of plaintext to be highly non-random, for instance being the length of the message. Therefore, if the IVs only differ in a few bit positions, it becomes very likely that an IV $\oplus$ 'd with its first plaintext block will equal a different IV $\oplus$ 'd with its corresponding first plaintext block. This will

be readily observable, since the first ciphertext block of the two messages will be equal. Especially if the IVs are known, this leaks information about what the two plaintext blocks are (one is, say, the same as the other except for the low-order bit). If the IVs are randomly chosen, then it is highly unlikely that the ⊕ of any IV with its plaintext will equal the ⊕ of a different IV with its plaintext. Although folklore says that the IVs should be randomly chosen, they don't really need to be random or unpredictable. They just need to be very different from each other. A strategy such as using a hash of a sequence number will work. 22.14 USE OF NONCES IN PROTOCOLS 585 22.14 USE OF NONCES IN PROTOCOLS Nonces are values that should be unique to a particular running of the protocol. Protocols use nonces as challenges (Bob gives Alice a challenge, and she proves she knows her secret by returning a function of the challenge and her secret), or perhaps as one of the inputs into the session key derived from the exchange. Some protocols would be insecure if the nonces were predictable, whereas other protocols only require that the nonce be unique, and a sequence number would be perfectly secure. If you don't want to bother analyzing whether your protocol requires unpredictable nonces, it is usually safest to generate them randomly. See §9.5 Nonce Types for examples of both types of protocol (those that require unpredictable nonces and those that do not). It has become fashionable to put lots of nonces into protocols, and furthermore, to require them all to be randomly chosen. For instance, IKE has nonces (which it says must be randomly chosen) as well as session identifiers (which it calls cookies and also requires to be randomly chosen), as well as message IDs (which should be sequence numbers, but IKE also requires them to be randomly chosen because randomly chosen has to be more secure, right?). All these values get thrown into the hash for the key, along with the Diffie-Hellman parameters which the spec says should also be unique for each session and of course must be randomly chosen. In fact, if the Diffie-Hellman values are unique for each exchange there is no need for the nonces in IKE at all. And the message IDs are more secure if they are a sequence number rather than a randomly chosen value. The message IDs are used to recognize replays, so therefore sequence numbers work much better. With randomly chosen message IDs, you'd have to remember every message ID you'd ever seen or generated. IKE uses the same cryptographic keys in both directions, so if you don't remember a message ID you received, you will not recognize a replay of that message as a replay, and if you don't remember a message ID you generated, you will be vulnerable to a reflection attack. 22.15 DON'T LET ENCRYPTED DATA BEGIN WITH A CONSTANT Folklore says you shouldn't let encrypted data begin with a constant. This worry is due to the fact that if there is no IV and the first block of data is a constant, and the key is short (perhaps 56 bits), then it would be possible to build a table mapping that known constant plaintext to its ciphertext with all possible keys. This is not a problem if there is an IV, or the key is of adequate length. 586 FOLKLORE 22.16 22.16 DON'T LET ENCRYPTED DATA BEGIN WITH A PREDICTABLE VALUE Folklore also says to avoid having the beginning of your data being a predictable value such as a sequence number. The worry here is that if you were doing a brute-force attack, you'd be able to easily recognize the plaintext. Sometimes data is sufficiently recognizable that avoiding starting with a predictable value will not help, for instance, if the data is 7-bit ASCII with parity. 22.17 COMPRESS DATA BEFORE ENCRYPTING IT Obviously, compressing data saves bandwidth, though this advantage might be offset by the disadvantage of additional computation to compress at the source and decompress at the destination. On the other hand, compression might provide a computation benefit because there are fewer octets to encrypt/decrypt after the data has been compressed. On the third hand (we1,2,3 have 6 hands between us), there might be hardware support for the encryption algorithm, but no hardware support for the compression algorithm. If encryption is used, then compression must occur before the data is encrypted. This is because compression algorithms depend on the

data being somewhat predictable. Encrypted data looks random and therefore would not compress. Another reason cryptographers think compression is a good idea is that they assume a bruteforce attacker would have a harder time recognizing correct compressed data after a trial decryption. One would think that a perfect compression algorithm would yield data that was harder to recognize because it would have less redundancy than the uncompressed data. However, in practice, most compression algorithms make compressed data easier to recognize than the original plaintext. Some compression algorithms start with a fixed header. For others there are sequences that are detectably illegal by the decompression algorithm. If the compression algorithm is known, doing the extra decompression step should add negligible computation load on the attacker, because a good compression algorithm can be decompressed quickly. 22.18 DON'T DO ENCRYPTION ONLY 587 22.18 DON'T DO ENCRYPTION ONLY It is common for people to assume that if something is encrypted, it's also integrity protected. But there are many attacks that can occur if encryption is used without integrity protection. For instance, a common form of encryption is RC4, which generates a pseudorandom stream of bits that is ⊕'d with the plaintext. Imagine a secret-key-based network security system in which a magic box, let's call it MB, is configured with user secrets, and shares a secret key with each server that a user might log into. Alice connects to server Bob, claiming to be Alice. Bob sends Alice a challenge and gets her response, and then sends to MB, over the encrypted tunnel, "Alice sent Z to challenge X." MB responds either "authentication success" or "authentication failure". The problem with this protocol is that if Trudy would like to impersonate Alice, and can act as a man-in-the-middle between MB and Bob, she can fool Bob by doing the following: • Connect to Bob, claiming to be Alice. • Bob will send her challenge X. • Trudy responds with anything, say Y. She knows this is almost certainly the wrong answer. • Bob sends to MB "Alice sent Y to challenge X." • MB will almost certainly respond "authentication failure". • Trudy captures the ciphertext stream from MB to Bob, ⊕s it with the string "authentication failure", and then ⊕s the result with "authentication success", and transmits that to Bob. • Bob will ⊕ what he receives with the RC4 stream to obtain the string "authentication success" and believe Trudy is Alice. 22.19 AVOIDING WEAK KEYS As described in §3.3.6 Weak and Semi-Weak Keys, some algorithms (like DES) have some keys that are weak, in the sense that ciphertext encrypted with those keys would be easier to cryptanalyze. Folklore says that when choosing an encryption key, care should be taken to avoid choosing one of the weak keys. Having any weak keys was considered a disadvantage of a cryptographic algorithm for the purpose of finding reasons for choosing among the AES contenders. In practice, as long as the probability of choosing a weak key is vanishingly small (16 out of 256 in the case of DES), weak keys are really not a problem. In fact, even if a cryptographic algorithm had no weak keys in the sense that the cryptographic community defines weak keys, in prac- 588 FOLKLORE 22.20 tice any key which happens to be one of the first searched by a brute-force attacker would be dangerous to use. For instance, if brute force attackers tend to search the key space sequentially from the bottom, then it would be inadvisable to choose a numerically small key. 22.20 MINIMAL VS. REDUNDANT DESIGNS Some protocols are designed so minimally that everything in them is essential. If you cut corners anywhere and don't follow the spec carefully, you will wind up with vulnerabilities. In contrast, it has become fashionable to over-engineer protocols, in the sense of accomplishing what is needed for security several different ways, when in fact n−1 of those ways are unnecessary as long as any one of them are done. An example is throwing in lots of nonces and other variables, each required to be random, whereas for security only one of them would be required to be random. We believe this fashion of redundancy is a dangerous trend, since the protocols become much more difficult to understand. And as the person writing the code for

choosing the nonce notices that there is no reason for the nonce to be random, and therefore doesn't bother, and the person writing the code for a different variable notices that it doesn't need to be random (because the nonce will be), you wind up likely to have a protocol with flaws. It's OK to cut n-1 corners, but not n corners. IKE is an excellent example of something that is so redundant and complex that the protocol design, as well as implementations based on it, wound up with security flaws. 22.21 OVERESTIMATE THE SIZE OF KEY U.S. law enforcement was always trying to find the "right" key size, big enough to be "secure enough" but small enough that it would be breakable, if necessary, by law enforcement. Of course, such a key size couldn't be really secure, because "secure" means nobody, including them, would be able to break it. Anyway, the assumption was that they were smarter, or had more computation power, than the bad guys from whom you were trying to protect yourself. Even if that were true (e.g., that law enforcement had ten times as much computation power as organized crime), given that computers keep getting faster and encrypted data often needs to be protected for many years, something breakable by law enforcement today would be breakable by organized crime in a few years. But it's also absurd to assume that the bad guys (whoever they are) would have significantly less computation power at their disposal than law enforcement. So it's dangerous to try to pick the smallest "sufficiently secure" key size. It's better to pick the largest you are willing to live with in 22.22 HARDWARE RANDOM NUMBER GENERATORS 589 terms of performance, or pick the size that you think is definitely going to be secure for ten years and double it just to be safe. This practice also makes it less likely that in a few years you'll have to go through the annoyance of reconfiguring everything to use larger keys. 22.22 HARDWARE RANDOM NUMBER GENERATORS Generating random numbers is tricky and there have been many examples of implementations that were insecure because of faulty random number generators. As a result, many people wish that computers would include a source of true randomness, which can be done at the hardware level by measuring noise of some sort. Indeed, it would be convenient and enhance security to have a hardware source of random numbers, but instead of depending on it as the sole source of randomness, it should be used to enhance other forms. However, suppose an organization was designing a hardware random number generator that was likely to become widely deployed. It would be very tempting to purposely (and secretly) design it so that it generated predictable numbers, but only predictable to the organization that designed the chip. It would be difficult or impossible for anyone to detect that this was done. For instance, each chip could be initialized with a true random 128-bit number as the first 128 bits of output, and each subsequent 128 bits of output could be computed as the previous 128 bits of output hashed with a large secret known only to the organization that designed and built the chips. Assuming the hash was good, the output would be indistinguishable from truly random. It would be more difficult to embed such a trap in software, since it is more easily reverse engineered. Even if a hardware random number generator might have such a trap, it is still useful, and can be used in such a way that would foil even the organization that embedded the trap. The way it should be used is to enhance any other scheme for generating random numbers. So random numbers should be generated as they would be without the chip (for instance, using mouse and keyboard inputs and low-order bits of a fine-granularity clock). But the output of the hardware random number generator should also be hashed into the random number calculation. 22.23 TIMING ATTACKS People want smart cards to be tamper proof, with no way of being able to obtain the private key from the card. An interesting attack that has been found on smart cards to gain information about the key is to observe how long it takes to do various operations on various types of input. This 590 FOLKLORE 22.24 yields information such as how many 1 bits are in the exponent, because exponentiation takes longer when the exponent has a lot of 1's. Similarly, the smart card might

use a different amount of power when it is exponentiating than when it is reducing mod n, so by measuring the power used, and knowing, for various inputs, when that input would need to be reduced given different sized exponents, more information about that smart card's private key can be obtained. To defend against such attacks, it is possible to do things that make it harder for the system using the smart card to control what operations the smart card is doing. Instead of just exponentiating the supplied number, for example, the smart card could multiply the supplied number by a randomly chosen number, do the exponentiation, and then divide by the exponentiated random number. Techniques like this either add noise to the information returned by the timing attack or pad things out to a fixed value. The techniques usually come with some performance cost. Some people misapply this smart card protection to servers speaking across a network, and attempt to yield no timing information about the exponent by introducing a random amount of delay when the server performs private key operations. This is usually unnecessary since any timing differences based on computation with a particular exponent would be dwarfed by network delays and delays due to server loading. There are some operations for which computation might be significant compared to the other delays. One example is the computation delays in PDM (see §10.3 Strong Password Protocols) of computing a safe prime from the user's password. This can be avoided by having the workstation compute the prime before it connects to the server, so an observer on the network cannot tell how long the workstation took to compute the prime. A related problem in the network case is traffic analysis. For instance, as shown in SONG01, information can be gleaned by observing the pattern of packets. One example is being able to see how many characters are in a user's password, since some implementations would send a separate packet for each character typed. (To a computer, the delays between character strokes are really long: "Geesh. That human is sure a slow input device. I'm certainly not waiting around for it to type another character!") 22.24 PUT CHECKSUMS AT THE END OF DATA Some protocols use a format for integrity-protected data where the integrity check is before the data, or possibly even in the middle of the data. It is better instead to have the integrity check at the end (as is done in ESP). If the integrity check is at the end, then it is possible to be computing it while transmitting the data, and then just append the computed integrity check at the end. The alternative, where the integrity check comes before the data (as is done in AH), requires the data to be buffered on output until the integrity check is computed and tacked onto the beginning. This adds delay and complexity. This rule is mainly for transmitted data. For received data, most likely it 22.25 FORWARD COMPATIBILITY 591 would be undesirable to act on the data before the integrity check is verified, so the data would need to be buffered no matter where the integrity check was stored. Having the integrity check in the middle of the data (as is done in IKE) complicates both the specification and the implementation. Space for the integrity check must be provided, with the field set to zero, and then the integrity check calculated and stored into the field. On receipt, the integrity check must be copied to some other location, and then zeroed in its location in the message, so that the integrity check can be verified. Especially complex is when an integrity check is computed based on selected fields, especially when some are optional and ordering is not strictly specified. The integrity check must be computed identically by the generator and verifier of the check. 22.25 FORWARD COMPATIBILITY History shows that protocols evolve. It is important to design a protocol in such a way that new capabilities can be added. This is a desirable property of any sort of protocol, not just security protocols. There are some special security considerations in evolving protocols, such as preventing an active attacker from tricking two parties into using an older, possibly less secure version of the protocol. 22.25.1 Options It is useful to allow new fields to be added to messages in future versions of the protocol. Sometimes these fields can be ignored by implementations that don't

support them. Sometimes a packet with an unsupported option should be dropped. In order to make it possible for an implementation that does not recognize an option to skip over it and parse the rest of the message, it is essential that there be some way to know where the option ends. There are two techniques: • Having a special marker at the end of the option. This tends to be computation-intensive, since the implementation must read all the option data as it searches for the end marker. • TLV encoding, which means each option starts with a TYPE field indicating the type of option, a LENGTH field indicating the length of the data in this option, and a VALUE field, which gives option-specific information. TLV encoding is more common because it is more efficient. However, the "L" must always be present, and in the same units, in order for implementations to be able to skip unknown options. Sometimes protocol designers who don't quite understand the concept of TLV encoding do clever 592 FOLKLORE 22.25.2 things like notice that the option they are defining is fixed length, so they don't need the LENGTH field. Or that one option might be expressed in different units. For instance, although AH (see §15.3 AH (Authentication Header) is designed to look like an IPv6 extension header, its length is expressed in units of 32-bit words, when all the other IPv6 options are expressed in units of 64-bit words. It is also useful to be able to add some options which should simply be skipped over by an implementation that does not support it, and to add other options which must be understood or else the packet must be dropped. But if an implementation does not recognize the option, how would it know whether it could be safely ignored or not? There are several possible solutions. One is to have a flag in the option header known as the critical bit, which if not set on an unknown option, indicates the option can simply be skipped over and ignored. Another possibility is to reserve some of the type numbers for critical options, and some of them for noncritical options (those that can be safely skipped if unrecognized). 22.25.2 Version Numbers A lot of protocols have a field for version number, but don't specify what to do with it. IKE and SSL are typical culprits. The purpose of a version number field is to allow the protocol to change in the future without confusing old implementations. One way of doing this without a version number is to declare the modified protocol to be a "new protocol", which would then need a different multiplexor value (a different TCP port for instance). With a version number, you can keep the same multiplexor value, but there have to be rules about handling version numbers so that an old implementation won't be confused by a redesigned packet format. 22.25.2.1 Version Number Field Must Not Move If versions are to be differentiated based on a version number field, then the version number field must always be in the same place in the message. Although this might seem obvious, when SSL was redesigned to be version 3, the version number field was moved! Luckily, there is a way to recognize which version an SSL message is (in version 2's client hello message, the first octet will be 128, and in version 3's client hello message, the first octet will be something between 20 and 23). 22.25.2.2 Negotiating Highest Version Supported Typically, when there is a new version of a protocol, the new implementations support both versions for some time. If you support both versions, how do you know what version to speak when talking to another node? Presumably the newer version is superior for some reason. So you typically first attempt to talk with the newer version, and if that fails, you attempt again with the older version. 22.25.2.3 FORWARD COMPATIBILITY 593 With this strategy it is important to make sure that two nodes that are both capable of speaking the new version wind up speaking the new version, and not getting fooled into speaking the older version because of lost messages or active attackers sending, deleting, or modifying messages. Why would an active attacker care enough to trick two nodes into speaking an earlier version of the protocol? Perhaps the newer version is more secure, or has features that the attacker would prefer the nodes not be able to use. (We'd hope there was some benefit to be gained by having designed a new version of the protocol!) The right thing to

do if you see a message with a higher version number than you support is to drop the message and send an error report to the other side indicating you don't support that version. But there will be no way for that error message to be cryptographically integrity-protected since the protocol has not been able to negotiate a key. So unless care is taken, nodes could be tricked into speaking the older version if an active attacker or network flakiness deleted the message of the initial attempt, or an attacker sent an error: unsupported version number message. One method of ensuring that two nodes don't get tricked into talking an older version is to have two version numbers in the packet. One would be the version number of the packet. The other would be the highest version the sender supports. But a single bit suffices, indicating that the sender can support a higher version number than the message indicates. If you establish a connection with someone, using version n, and you support something higher than n, and you receive authenticated messages with the HIGHER VERSION NUMBER SUPPORTED flag, then you can attempt to reconnect with a higher version number. 22.25.2.3 Minor Version Number Field Another area in which protocol designers get confused is the proper use of a MINOR VERSION NUMBER field. Why should there be both MAJOR VERSION NUMBER and MINOR VERSION NUMBER fields? The proper use of a minor version number is to indicate new capabilities that are backward compatible. The major version number should change if the protocol is incompatible. The minor version number is informational only. If the node you are talking to indicates it is version 4.7 (where 4 is the major version number and 7 is the minor version), and you are version 4.3, then you ignore the minor version number. But the version 4.7 node might use it to know that there are certain fields you wouldn't support, so it won't send them. For instance, ISAKMP handles minor version numbers incorrectly. It has an 8-bit VERSION NUMBER field, split into 4 bits for MAJOR VERSION, and 4 bits for MINOR VERSION. The specification says you should reject a message if the major version is higher, or if the major versions are the same and the minor version is higher than yours. So the result is exactly the same as if it was just an 8-bit field, but it's more complicated to understand and will run out of numbers more quickly than an 8- bit field since there are only 16 major version numbers. (Unless someone worries about numbers getting used up and mandates that you must use up all the minor version numbers before you're allowed to bump the major version number.) 594 FOLKLORE 22.25.3 Most likely the confusion about the proper use of the minor version number is because software releases have major and minor version numbers, and the choice as to which to increment is a marketing decision. 22.25.3 Vendor Options Another type of option is a vendor-defined option. The only difference between a vendor-defined option and the type of option we described in §22.25.1 Options is that a vendor-defined option might not be able to obtain one of the compact T values for specifying that option in TLV encoding. The organization that assigns T values might require that the option be publicly documented in order to obtain a number, and the vendor might want to keep their use proprietary. Or the vendor might want to experiment with the option before bothering to request a number. Or the vendor might fail to convince a standards organization to adopt the option, and T values might only be given to options that have been standardized. If an option does not have one of the compact assigned T values, there must be a way for it to be assigned a unique number. ASN.1 defines a way to obtain OIDs, but OIDs are variable length. This complicates encoding of a vendor option. IKE encodes each vendor option in an explicit envelope with a type and length. The type code in the envelope is simply "vendor option" (i.e., the same for all vendor options). Inside the envelope is the specific vendor option data, and it is up to the vendor to define an encoding. Ideally the encoding is such that there is no potential ambiguity between different vendors' vendor options. This could be done by using an OID, or defining the first 16 octets to be a hash of information guaranteed to be unique, such as the vendor name, location, name of the

capability, etc. 22.26 NEGOTIATING PARAMETERS It is common today for security protocols to start out by negotiating which cryptographic algorithms they will use. It would certainly be simpler if the protocol specified the algorithms. So why all the complexity of negotiating cryptographic algorithms? • There might be reasons to support different algorithms and trade off performance versus strength. Perhaps the application, knowing that the data was particularly sensitive, might negotiate a particularly strong suite of crypto. 22.27 HOMEWORK 595 • In order for domestic implementations capable of strong crypto to interwork with exported implementations, it was necessary to negotiate crypto, in order to use strong crypto when legal and weak crypto otherwise. • A crypto algorithm might get broken. Therefore, if implementations support several cryptographic algorithms, removing a broken algorithm requires only a configuration change (stop using that algorithm) rather than new code to support a new algorithm. • Some countries, companies, or governments, might want to use their own cryptographic algorithms (this is sometimes known as vanity crypto), in addition to being able to interwork with standard implementations. Negotiation of cryptographic algorithms can either be done by defining suites of algorithms (as is done in SSL/TLS), or by individually negotiating choices for each specific algorithm (encryption, integrity, signature,...). The individual algorithms are not necessarily mix-and-match. Some algorithms can only be used with specific other algorithms, for instance DSS requires SHA-1 as the hash. Proposing each combination individually can lead to an exponential explosion of suite proposals, as in ISAKMP/IKE (see §16.5.5 Negotiating Cryptographic Parameters). Care must be taken that an active attacker cannot trick Alice and Bob into talking weaker crypto. This is usually done by having Alice and Bob sign or otherwise integrity-protect the negotiation messages in order to detect tampering. 22.27 HOMEWORK 1. Suppose you were able to observe ciphertext that you knew had been encrypted in CBC mode, and you saw that two ciphertext blocks, say c2 and c5, were equal. Why would this leak information? (Hint: look at Figure 4-5 and compare c1 $\oplus$ c4 with m2 $\oplus$ m5. What would happen if you knew one of the plaintexts, say m2?) 2. Suppose Alice and Bob negotiate a 64-bit key, and use the low-order 40 bits of it for encryption (for export reasons), and use the entire 64 bits for integrity protection. How much work would it be to brute-force break the key and construct a forged encrypted message using that key? 3. Consider the following protocol. Must the challenge be unpredictable, or is it sufficient to ensure Bob never chooses the same challenge twice, for instance, by using a sequence number? 596 FOLKLORE 22.27 4. As described in §22.3.1 The Splicing Attack, if a C-D conversation is being multiplexed over the same encryption-only tunnel as an A-B conversation, C and D can collude with a splicing attack in order to decrypt the A-B traffic. Suppose encryption is CBC with 64-bit blocks. Assume that the first 16 octets of the plaintext packet consist of the source and destination address, and the remainder of the packet is data. The splicing attack will allow D to see all but the first block of data, which will be garbled because of the splice. Explain why the first block will be garbled, and why subsequent blocks will not be. How can C create the ciphertext so that D will receive all of the data? (Hint: assume that it is legal to create a larger ciphertext packet than the one recorded.) Alice I'm Alice Bob KAlice-Bob{R} (KAlice-Bob+1){R+1} 597 BIBLIOGRAPHY ALAG93 Alagappan, K., Telnet Authentication: SPX, RFC 1412, January 1993. BALE85 Balenson, D., "Automated Distribution of Cryptographic Keys Using the Financial Institution Key Management Standard", IEEE Communications, Vol. 23 #9, September 1985, pp. 41–46. BALE93 Balenson, D., Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers, RFC 1423, February 1993. BELL74 Bell, D. E. and LaPadula, L. J., Secure Computer Systems: Mathematical Foundations and Model, M74-244, Mitre Corp., October 1974. BELL90 Bellovin, S. M. and Merritt, M., "Limitations of the Kerberos Authentication System", Computer Communications Review, Vol. 20 #5,