13

Secure Communications over Insecure Networks

13.1 An Introduction to Cryptography

It is sometimes necessary to communicate over insecure links without exposing one's systems. Cryptography—the art of secret writing—is the usual answer.

The most common use of cryptography is, of course, secrecy. A suitably encrypted packet is incomprehensible to attackers. In the context of the Internet, and in particular when protecting wide-area communications, secrecy is often secondary. Instead, we are often interested in the implied authentication provided by cryptography. That is, a packet that is not encrypted with the proper key will not decrypt to anything sensible. This considerably limits the ability of an attacker to inject false messages.

Before we discuss some actual uses for cryptography, we present a brief overview of the subject and build our cryptographic toolkit. It is of necessity sketchy; cryptography is a complex subject that cannot be covered fully here. Readers desiring a more complete treatment should consult any of a number of standard references, such as [Kahn, 1967], [Denning, 1982], [Davies and Price, 1989], or [Schneier, 1994].

We next discuss the Kerberos Authentication System, developed at MIT. Apart from its own likely utility—the code is widely available and Kerberos is being considered for adoption as an Internet standard—it makes an excellent case study, since it is a real design, not vaporware, and has been the subject of many papers and talks and a fair amount of experience.

Selecting an encryption system is comparatively easy; actually using one is less so. There are myriad choices to be made about exactly where and how it should be installed, with trade-offs in terms of economy, granularity of protection, and impact on existing system. Accordingly, Sections 13.3, 13.4, and 13.5 discuss the trade-offs and present some security systems in use today.

In the discussion that follows, we assume that the *cryptosystems* involved—that is, the cryptographic algorithm and the protocols that use it, but not necessarily the particular implementation—

are sufficiently strong, i.e., we discount almost completely the possibility of cryptanalytic attack. Cryptographic attacks are orthogonal to the types of attacks we describe elsewhere. (Strictly speaking, there are some other dangers here. While the cryptosystems themselves may be perfect, there are often dangers lurking in the cryptographic protocols used to control the encryption. See, for example, [Moore, 1988]. Some examples of this phenomenon are discussed in Section 13.2 and in the box on page 213.) A site facing a serious threat from a highly competent foe would need to deploy defenses against both cryptographic attacks and the more conventional attacks described elsewhere.

One more word of caution: in some countries the export, import, or even use of any form of cryptography may be regulated by the government. Additionally, many useful cryptosystems are protected by a variety of patents. It may be wise to seek competent legal advice.

13.1.1 Notation

Modern cryptosystems consist of an operation that maps a *plaintext* (P) and a *key* (K) to a *ciphertext* (C). We write this as

$$C \leftarrow K[P].$$

Usually, there is an inverse operation that maps a ciphertext and key K^{-1} to the original plaintext:

$$P \leftarrow K^{-1}[C].$$

The attacker's usual goal is to recover the keys K and K^{-1}. For a strong cipher, it should be impossible to recover them by any means short of trying all possible values. This should hold true no matter how much ciphertext and plaintext the enemy has captured.

It is generally accepted that one must assume that attackers are familiar with the encryption function; the security of the cryptosystem relies entirely on the secrecy of the keys. Protecting them is therefore of the greatest importance. In general, the more a key is used, the more vulnerable it is to compromise. Accordingly, separate keys, called *session keys*, are used for each job. Distributing session keys is a complex matter, about which we will say little; let it suffice to say that session keys are generally transmitted encrypted by a *master key*, and often come from a centralized *Key Distribution Center*.

13.1.2 Private-Key Cryptography

In conventional cryptosystems—sometimes known as secret-key or symmetric cryptosystems—there is only one key. That is,

$$K = K^{-1};$$

writing out K^{-1} is simply a notational convenience to indicate decryption. There are many different types of symmetric cryptosystems; here, we will concentrate on the *Data Encryption Standard* (*encryption, DES*) [NBS, 1977] and its standard modes of operation [NBS, 1980]. Note, though, that most things we say are applicable to other modern cipher systems, with the obvious exception of such parameters as encryption block size and key size.

Types of Attacks

Cryptographic systems are subject to a variety of attacks. It is impossible to give a complete taxonomy—but we discuss a few of the more important ones.

Cryptanalysis: Cryptanalysis is the science—or art—of reading encrypted traffic without prior knowledge of the key.

"Practical" cryptanalysis: "Practical" cryptanalysis is, in a sense, the converse. It refers to stealing a key, by any means necessary.

Known-plaintext attack: Often, an enemy will have one or more pairs of ciphertext and a known plaintext encrypted with the same key. These pairs, known as *cribs*, can be used to aid in cryptanalysis.

Chosen-plaintext: Attacks where you trick the enemy into encrypting your messages with the enemy's key. For example, if your opponent encrypts traffic to and from a file server, you can mail that person a message and watch the encrypted copy being delivered.

Exhaustive search: Trying every possible key. Also known as *brute force*

Passive eavesdropping: A passive attacker simply listens to traffic flowing by.

Active attack: In an active attack, the enemy can insert messages and—in some variants—delete or modify legitimate messages.

Man-in-the-middle: The enemy sits between you and the party with whom you wish to communicate, and impersonates each of you to the other.

Replay: Take a legitimate message and reinject it into the network at a later time.

Cut-and-paste: Given two messages encrypted with the same key, it is sometimes possible to combine portions of two or more messages to produce a new message. You may not know what it says, but you can use it to trick your enemy into doing something for you.

Time-resetting: In protocols that use the current time, try to confuse you about what the correct time is.

Birthday attack: An attack on hash functions where the goal is to find any two messages that yield the same value. If exhaustive search takes 2^n steps, a birthday attack would take only $2^{n/2}$ tries.

DES is a form of encryption system known as a *block cipher*. That is, it operates on fixed-size blocks. It maps 64-bit blocks of plaintext into 64-bit blocks of ciphertext and vice versa. DES keys are 64 bits long, including 8 seldom-used parity bits.

Encryption in DES is performed via a complex series of permutations and substitutions. The result of these operations is exclusive-OR'd with the input. This sequence is repeated 16 times, using a different ordering of the key bits each time. Complementing one bit of the key or the plaintext will flip approximately 50% of the output bits. Thus, almost no information about the inputs is leaked; small perturbations in the plaintext will produce massive variations in the ciphertext.

DES was developed at IBM in response to a solicitation for a cryptographic standard from the National Bureau of Standards (NBS, now known as the National Institute of Standards and Technology or NIST). It was originally adopted for nonclassified federal government use, effective January 15, 1978. Every five years, a recertification review is held. The last one, in 1993, reaffirmed DES for financial and authentication use. It is unclear what will happen in 1998.

IDEA [Lai, 1992] is similar in overall structure to DES. It derives its strength from its use of three different operations—exclusive-OR, modular addition, and modular multiplication—in each round, rather than just using exclusive-OR. Additionally, it uses a 128-bit key to guard against exhaustive search attacks. The IDEA algorithm is patented, but the patent holders have granted blanket permission for noncommercial use. Although IDEA appears to be a strong cipher, it is relatively new, and has not been subject to much scrutiny as yet. Some caution may be in order.

Recently, a new block cipher, *Skipjack*, was announced by NIST. Skipjack is used in the so-called *Clipper* and *Capstone* encryption chips [Markoff, 1993a; NIST, 1994b]. These chips are controversial not because of their technical merits (though those are as-yet largely classified), but because the chips implement a *key escrow* system. Transmissions contain an encrypted header containing the session key; government agencies with access to the header-encryption keys will be able to decrypt the conversation. The government claims that a court order will be required for such access.

Politics aside, Skipjack appears to be a conventional block cipher. It uses a 64-bit block size, an 80-bit key size, and 32 internal rounds. Because of the requirement for the escrow mechanism, only hardware implementations of Skipjack will be available. An outside review panel concluded that the algorithm was quite strong and that "There is no significant risk that Skipjack can be broken through a shortcut method of attack" [Brickell *et al.*, 1993].

13.1.3 Modes of Operation

Block ciphers such as DES, IDEA, and Skipjack are generally used as primitive operators to implement more complex *modes of operation*. The four standard modes are described next. All of them can be used with any block cipher, although we have used DES in the examples.

Electronic Code Book Mode

The simplest mode of operation, *Electronic Code Book* (*ECB*) mode, is also the most obvious: DES is used, as is, on 8-byte blocks of data. Because no context goes into each encryption, every time the same 8 bytes are encrypted with the same key, the same ciphertext results. This allows an enemy to collect a “code book” of sorts, a list of 8-byte ciphertexts and their likely (or known) plaintext equivalents. Because of this danger, ECB mode should be used only for transmission of keys and initialization vectors (see below).

Cipher Block Chaining Mode

Cipher Block Chaining (*CBC*) is the most important mode of operation. In CBC mode, each block of plaintext is exclusive-OR’d with the previous block of ciphertext before encryption. That is,

$$C_n \leftarrow K[P_n \oplus C_{n-1}].$$

To decrypt, we reverse the operation:

$$P_n \leftarrow K^{-1}[C_n] \oplus C_{n-1}.$$

Two problems immediately present themselves: how to encrypt the first block when there is no C_0, and how to encrypt the last block if our message is not a multiple of 8 bytes in length.

To solve the first problem, both parties must agree upon an *initialization vector* (*IV*). The IV acts as C_0, the first block of cipher; it is exclusive-OR’d with the first block of plaintext before encryption. There are some subtle attacks possible if IVs are not chosen properly; to be safe, IVs should be (a) chosen randomly; (b) not used with more than one other partner; and (c) either transmitted encrypted in ECB mode or chosen anew for each separate message, even to the same partner [Voydock and Kent, 1983].

Apart from solving the initialization problem, IVs have another important role: they disguise stereotyped beginnings of messages. That is, if the IV is held constant, two encryptions of the same start of a message will yield the same cipher text. Apart from giving clues to cryptanalysts and traffic analysts, in some contexts it is possible to replay an intercepted message. Replays may still be possible if the IV has changed, but the attacker will not know what message to use.

Dealing with the last block is somewhat more complex. In some situations, length fields are used; in others, bytes of padding are acceptable. One useful technique is to add padding such that the last byte indicates how many of the trailing bytes should be ignored. It will thus always contain a value between 1 and 8.

A transmission error in a block of ciphertext will corrupt both that block and the following block of plaintext when using CBC mode.

Output Feedback Mode

For dealing with asynchronous streams of data, such as keyboard input, *output feedback mode* (*OFB*) is sometimes used. OFB uses DES as a random number generator, by looping its output back to its input, and exclusive-OR’ing the output with the plaintext:

$$\begin{aligned} DES_n &\leftarrow K[DES_{n-1}] \\ C_n &\leftarrow P_n \oplus DES_n. \end{aligned}$$

If the P_n blocks are single bytes, we are, in effect, throwing away 56 bits of output from each DES cycle. In theory, the remaining bits could be kept and used to encrypt the next 7 bytes of plaintext, but that is not standard. As with CBC, an IV must be agreed on. It may be sent in the clear, because it is encrypted before use. Indeed, if it is sent encrypted, that encryption should be done with a different key than is used for the OFB loop.

OFB has the property that errors do not propagate. Corruption in any received ciphertext byte will affect only that plaintext byte. On the other hand, an enemy who can control the received ciphertext can control the changes that are introduced in the plaintext: a complemented ciphertext bit will cause the same bit in the plaintext to be complemented.

Cipher Feedback Mode

Cipher Feedback (*CFB*) mode is a more complex mechanism for encrypting streams. If we are encrypting 64-bit blocks, we encipher as follows:

$$C_n \leftarrow P_n \oplus K[C_{n-1}].$$

Decryption is essentially the same operation:

$$P_n \leftarrow C_n \oplus K[C_{n-1}].$$

That is, the last ciphertext block sent or received is fed back into the encryptor. As in OFB mode, DES is used in encryption mode only.

If we are sending 8-bit blocks, CFB_8 mode is used. The difference is that the input to the DES function is from a shift register; the 8 bits of the transmitted ciphertext are shifted in from the right, and the leftmost 8 bits are discarded.

Errors in received CFB data affect the decryption process while the garbled bits are in the shift register. Thus, for CFB_8 mode, 9 bytes are affected. The error in the first of these bits can be controlled by the enemy.

As with OFB mode, the IV for CFB encryption may, and arguably should, be transmitted in the clear.

One-Time Passwords

Conventional cryptosystems are often used to implement the authentication schemes described in Chapter 5. In a challenge/response authenticator, the user's token holds the shared secret key K. The challenge *Ch* acts as plaintext; both the token and the host calculate $K[Ch]$. Assuming that a strong cryptosystem is used, there is no way to recover K from the challenge/response dialog.

A similar scheme is used with time-based authenticators. The clock value T is the plaintext; $K[T]$ is displayed.

PINs can be implemented in either form of token in a number of different ways. One technique is to use the PIN to encrypt the device's copy of K. An incorrect PIN will cause an incorrect copy of K to be retrieved, thereby corrupting the output. Note the host does not need to know the PIN, and need not be involved in PIN-change operations.

How Secure Is DES?

There has been a fair amount of controversy about DES over the years; see, for example, [Diffie and Hellman, 1977]. Some have charged that the design was deliberately sabotaged by the National Security Agency (NSA), or that the key size is just small enough that a major government or large corporation could afford to build a machine that tried all 2^{56} possible keys for a given ciphertext. That said, the algorithm has successfully resisted attack by civilian cryptographers for almost two decades. Moreover, recent research results [Biham and Shamir, 1991, 1993] indicate that the basic design of DES is actually quite strong, and was almost certainly not sabotaged. If your enemy does not have significant resources, DES is adequate protection.

However, a design has recently been presented for an "economical" DES-cracker based on exhaustive search [Wiener, 1994]. Wiener estimates that a machine can be built for $1,000,000 that will find any DES key in about 7 hours; an average search would take half that time. The design scales nicely in both directions; a $10,000,000 version would find any key in 0.7 hours, or 42 minutes, while a smaller $100,000 machine would succeed in 70 hours, which is quite adequate in many cases.

Clearly, it is worth taking extra precautions with sensitive information, especially when using master keys. An enemy who cracks a session key can read that one session, but someone who cracks a master key can read all traffic, past, present, and future. The most sensitive message of all is a session key encrypted by a master key, since two brute force attacks—first to recover the session key and then to match that against its encrypted form—will reveal the master [Garon and Outerbridge, 1991]. Accordingly, *triple encryption* is recommended if you think your enemy is well financed.

To perform triple encryption, use two DES keys, K_1 and K_2:

$$C \leftarrow K_1[K_2^{-1}[K_1[P]]].$$

Note that the middle encryption is actually a decryption. This is done for two reasons. First, it was originally suggested that double encryption with two keys K_1 and K_2 might actually be equivalent to simple encryption with a third key, K_3, unknown to the legitimate recipients but recoverable by a cryptanalyst. It is now known that that is not possible: there is no such K_3 [Campbell and Wiener, 1993]. Second, and more important, by setting $K_1 = K_2$, we have backward compatibility with systems that only do single encryption.

This form of triple encryption gives you 112 bits of key strength. Simply doing double encryption isn't as strong against an enemy who can afford lots of storage [Merkle and Hellman, 1981]. You can make triple encryption even stronger by choosing three independent keys K_1, K_2, and K_3. Again, there is compatibility with single encryption if the three keys are equal.

13.1.4 Public Key Cryptography

With conventional cipher systems, both parties must share the same secret key before communication begins. This is problematic. For one thing, it is impossible to communicate with someone for whom you have no prior arrangements. Additionally, the number of keys needed for a complete communications mesh is very large, n^2 keys for an n-party network. While both problems can be solved by recourse to a trusted, centralized KDC, it is not a panacea. If nothing else, the KDC must be available in real time to initiate a conversation. This makes KDC access difficult for store-and-forward message systems.

Public key, or asymmetric, cryptosystems [Diffie and Hellman, 1976] offer a different solution. In such systems, $K \neq K^{-1}$. Furthermore, given K, the encryption key, it is not feasible to discover the decryption key K^{-1}. We write encryption as

$$C \leftarrow E_A[P]$$

and decryption as

$$P \leftarrow D_A[C].$$

for the keys belonging to A.

Each party publishes its encryption key in a directory, while keeping its decryption key secret. To send a message to someone, simply look up their public key and encrypt the message with that key.

The best known, and most important, public key cryptosystem is known as *RSA*, for its inventors, Ronald Rivest, Adi Shamir, and Leonard Adleman [Rivest *et al.*, 1978]. Its security relies on the difficulty of factoring very large numbers. It is protected by a U.S. patent. Legal commercial versions are available; there is also a free but licensed package, RSAREF, that is available on the Internet. Commercial use of RSAREF is barred, as is export from the United States. Other significant restrictions apply as well; you should check the latest version of the RSAREF license before using the code.

To use RSA, pick two large prime numbers p and q; each should be at least several hundred bits long. Let $n = pq$. Pick some random integer d relatively prime to $(p-1)(q-1)$, and e such that

$$ed \equiv 1 \ (\mathrm{mod}\ (p-1)(q-1)).$$

That is, when the product ed is divided by $(p-1)(q-1)$, the remainder is 1.

We can now use the pair (e, n) as the public key, and the pair (d, n) as the private key. Encryption of some plaintext P is performed by exponentiation modulo n:

$$C \leftarrow P^e \ (\mathrm{mod}\ n).$$

Decryption is the same operation, with d as the exponent:

$$\begin{aligned} P \leftarrow C^d \ (\mathrm{mod}\ n) &\equiv (P^e)^d \ (\mathrm{mod}\ n) \\ &\equiv P^{ed} \ (\mathrm{mod}\ n) \\ &\equiv P \ (\mathrm{mod}\ n). \end{aligned}$$

No way to recover d from e is known that does not involve factoring n, and that is believed to be a very difficult operation.

Public key systems suffer from two principal disadvantages. First, the keys are very large compared with those of conventional cryptosystems. This might be a problem when it comes to entering or transmitting the keys, especially in secure mail messages (discussed later). Second, encryption and decryption are much slower. Not much can be done about the first problem. The second is dealt with by using such systems primarily for key distribution. Thus, if A wanted to send a secret message M to B, A would transmit something like

$$E_B[K], K[M] \tag{13.1}$$

where K is a randomly generated session key for DES or some other conventional cryptosystem.

13.1.5 Exponential Key Exchange

A concept related to to public-key cryptography is *exponential key exchange*, sometimes referred to as *Diffie-Hellman* [Diffie and Hellman, 1976]. Indeed, it is an older algorithm; the scheme was first described in the same paper that introduced the notion of public-key cryptosystems, but without providing any examples.[1]

Exponential key exchange provides a mechanism for setting up a secret but *unauthenticated* connection between two parties. That is, the two can negotiate a secret session key, without fear of eavesdroppers. However, neither party has any strong way of knowing who is really at the other end of the circuit.

In its most common form, the protocol uses arithmetic operations in the *field* of integers modulo some large number β. When doing arithmetic $\pmod{\beta}$, you perform the operation as usual, but then divide by β, discarding the quotient and keeping the remainder. In general, you can do the arithmetic operations either before or after taking the remainder. Both parties must also agree on some integer α, $1 < \alpha < \beta$.

Suppose A wishes to talk to B. They each generate secret random numbers, R_A and R_B. Next, A calculates and transmits to B the quantity

$$\alpha^{R_A} \pmod{\beta}.$$

Similarly, B calculates and transmits

$$\alpha^{R_B} \pmod{\beta}.$$

Now, A knows R_A and $\alpha^{R_B} \pmod{\beta}$, and hence can calculate

$$\begin{aligned}(\alpha^{R_B})^{R_A} \pmod{\beta} &\equiv \alpha^{R_B R_A} \pmod{\beta} \\ &\equiv \alpha^{R_A R_B} \pmod{\beta}.\end{aligned}$$

Similarly, B can calculate the same value. But an outsider cannot; the task of recovering R_A from $\alpha^{R_A} \pmod{\beta}$ is believed to be very hard. (This problem is known as the *discrete logarithm* problem.) Thus, A and B share a value known only to them; it can be used as a session key for a symmetric cryptosystem.

Again, caution is indicated when using exponential key exchange. As noted, there is no authentication provided; *anyone* could be at the other end of the circuit, or even in the middle, relaying

[1]Exponential key exchange is protected by a patent in the United States.

messages to each party. Simply transmitting a password over such a channel is risky, because of "man-in-the-middle" attacks. There are techniques for secure transmission of authenticating information when using exponential key exchange; see, for example, [Rivest and Shamir, 1984; Bellovin and Merritt, 1992, 1993, 1994]. But they are rather more complex and still require prior transmission of authentication data.

13.1.6 Digital Signatures

Often the source of a message is at least as important as its contents. *Digital signatures* can be used to identify the source of a message. Like public key cryptosystems, digital signature systems employ public and private keys. The sender of a message uses a private key to sign it; this signature can be verified by means of the public key.

Digital signature systems do not necessarily imply secrecy. Indeed, a number of them do not provide it. However, the RSA cryptosystem can be used for both purposes.

To sign a message with RSA, the sender *decrypts* it, using a private key. Anyone can verify—and recover—this message by *encrypting* with the corresponding public key. (The mathematical operations used in RSA are such that one can decrypt plaintext, and encrypt to recover the original message.) Consider the following message:

$$E_B[D_A[M]].$$

Because it is encrypted with B's public key, only B can strip off the outer layer. Because the inner section $D_A[M]$ is encrypted with A's private key, only A could have generated it. We therefore have a message that is both private and authenticated. We write a message M signed by A as

$$S_A[M].$$

There are a number of other digital signature schemes besides RSA. Perhaps the most important one is the *Digital Signature Standard* (*DSS*) recently proposed by NIST [NIST, 1994a]. Apparently by intent, its keys cannot be used to provide secrecy, only authentication. This makes products using the standard exportable, but some have charged that the U.S. government wishes to protect its ability to use wiretaps [Markoff, 1991; Denning, 1993]. Nevertheless, it is likely to be adopted as a federal government standard. If the history of DES is any guide, it will be adopted by industry as well, although patent license fees may impede that move.

How does one know that the published public key is authentic? The cryptosystems themselves may be secure, but that matters little if an enemy can fool a publisher into announcing the wrong public keys for various parties. That is dealt with via *certificates*. A certificate is a combination of a name and a public key, collectively signed by another, and more trusted, party T:

$$S_T[A, E_A].$$

That signature requires its own public key of course. It may require a signature by some party more trusted yet, etc.:

$$\begin{gathered} S_{T_1}[A, E_A] \\ S_{T_2}[T_1, E_{T_1}] \\ S_{T_3}[T_2, E_{T_2}]. \end{gathered}$$

Certificates may also include additional information, such as the key's expiration date. One does not wish to use any one key for too long for fear of compromise, and one does not want to be tricked into accepting old, and possibly broken, keys.

A concept related to digital signatures is that of the *Message Authentication Code* (*MAC*). A MAC is formed by running a block cipher in CBC mode over the input. Only the last block of output is kept. A change to any of the input blocks will cause a change to the MAC value, thus allowing transmission faults or tampering to be detected. This is essentially a fancy checksum.

When MACs are used with encrypted messages, the same key should not be used for both encryption and message authentication. Typically, some simple transform of the encryption key, such as complementing the bits, is used in the MAC computation.

13.1.7 Secure Hash Functions

It is often impractical to apply an encryption function to an entire message. A function like RSA can be too expensive for use on large blocks of data. In such cases, a *secure hash function* can be employed. A secure hash function has two interesting properties. First, its output is generally relatively short—on the order of 128 bits. Second, and more important, it must be infeasible to create an input that will produce an arbitrary output value. Thus, an attacker cannot create a fraudulent message that is authenticated by means of an intercepted genuine hash value.

Secure hash functions are used in two main ways. First, and most obvious, any sort of digital signature technique can be applied to the hash value instead of to the message itself. In general, this is a much cheaper operation, simply because the input is so much smaller. Thus, if A wished to send to B a signed version of message (13.1), A would transmit

$$E_B[K], K[M], S_A[H(M)]$$

where H is a secure hash function. As before, K is the secret key used to encrypt the message itself. If, instead, we send

$$E_B[K], K[M, S_A[H(M)]],$$

the signature, too, and hence the origin of the message, will be protected from all but B's eyes.

The second major use of secure hash functions is less obvious. In conjunction with a shared secret key, the hash functions themselves can be used to sign messages. By prepending the secret key to the desired message, and then calculating the hash value, one produces a signature that cannot be forged by a third party:

$$H(M, K), \tag{13.2}$$

where K is a shared secret string and M is the message to be signed.

This concept extends in an obvious way to challenge/response authentication schemes. Normally, in response to a challenge C_A from A, B would respond with $K[C_A]$, where K is a shared key. But the same effect can be achieved by sending $H(C_A, K)$ instead. This technique has sometimes been used to avoid export controls on encryption software: licenses to export authentication technology, as opposed to secrecy technology, are easy to obtain.

Observe that we have written $H(M, K)$ rather than $H(K, M)$. Under certain circumstances, the latter can be insecure [Tsudik, 1992]; the hash of a message can be used as the input value to the hash of a longer string that has the original message—including the secret key—as a prefix.

It is important that secure hash functions have a relatively long output, at least 128 bits. If the output value is too short, it is possible to find two messages that hash to the same value. This is much easier than finding a message with a given hash value. If a brute force attack on the latter takes 2^m operations, a birthday attack takes just $2^{m/2}$ tries. If the hash function yielded as short an output value as DES, two collisions of this type could be found in only 2^{32} tries. That's far too low. The name comes from the famous *birthday paradox*. On average, there must be 183 people in a room for there to be a 50% probability that someone has the same birthday as you. But only 23 people need to be there for there to be a 50% probability that *some* two people share the same birthday.

There are a number of well-known hash functions from which to choose. Some care is needed, because the criteria for evaluating their security are not well established [Nechvatal, 1992]. Among the most important such functions are MD2 [Kaliski, 1992], MD5 [Rivest, 1992], and NIST's Secure Hash Algorithm [NIST, 1993], a companion to its digital signature scheme. The two-pass version of Merkle's *snefru* algorithm [Merkle, 1990a] has been broken, and the three-pass version has known weaknesses. It is not recommended for use with less than eight passes, but that makes it very slow. As of this writing, the NIST algorithm appears to be the best choice.

On occasion, it has been suggested that a MAC calculated with a known key is a suitable hash function. Such usages are not secure [Winternitz, 1984; Mitchell and Walker, 1988]. Secure hash functions can be derived from block ciphers, but a more complex function is required [Merkle, 1990b].

13.1.8 Timestamps

Haber and Stornetta [Haber and Stornetta, 1991a, 1991b] have shown how to use secure hash functions to implement a *digital timestamp* service. Messages to be timestamped are *linked* together. The hash value from the previous timestamp is used in creating the hash for the next one.

Suppose we want to timestamp document D_n at some time T_n. We create a *link value* L_n by calculating

$$L_n \leftarrow H(T_n, H(D_n), n, L_{n-1}).$$

This value L_n serves as the timestamp. The time T_n is, of course, unreliable; however, L_n is used as an input when creating L_{n+1}, and uses L_{n-1} as an input value. The document D_n must therefore have been timestamped before D_{n+1} and after D_{n-1}. If these documents belonged to a different company than D_n, the evidence is persuasive. The entire sequence can be further tied to reality by periodically publishing the link values. Bellcore does just that, in a legal notice in the *New York Times*.[2]

[2]This scheme has been patented by Bellcore.

Note, incidentally, that one need not disclose the contents of a document to secure a timestamp; a hash of it will suffice. This preserves the secrecy of the document, but proves its existence at a given point in time.

13.2 The Kerberos Authentication System

The Kerberos Authentication System [Bryant, 1988; Kohl and Neuman, 1993; Miller *et al.*, 1987; Steiner *et al.*, 1988] was designed at MIT as part of Project Athena.[3] It serves two purposes: authentication and key distribution. That is, it provides to hosts—or more accurately, to various services on hosts—unforgeable credentials to identify individual users. Each user and each service shares a secret key with the Kerberos key distribution center; these keys act as master keys to distribute session keys, and as evidence that the KDC vouches for the information contained in certain messages. The basic protocol is derived from one originally proposed by Needham and Schroeder [Needham and Schroeder, 1978, 1987; Denning and Sacco, 1981].

More precisely, Kerberos provides evidence of a *principal*'s identity. A principal is generally either a user or a particular service on some machine. A principal consists of the three-tuple

$$\langle \textit{primary name}, \textit{instance}, \textit{realm} \rangle .$$

If the principal is a user—a genuine person—the *primary name* is the login identifier, and the *instance* is either null or represents particular attributes of the user, e.g., *root*. For a service, the service name is used as the primary name and the machine name is used as the instance, e.g., *rlogin.myhost*. The *realm* is used to distinguish among different authentication domains; thus, there need not be one giant—and universally trusted—Kerberos database serving an entire company.

All Kerberos messages contain a checksum. This is examined after decryption; if the checksum is valid, the recipient can assume that the proper key was used to encrypt it.

Kerberos principals may obtain *tickets* for services from a special server known as the *Ticket-Granting Server* (*TGS*). A ticket contains assorted information identifying the principal, encrypted in the private key of the service (notation is summarized in Table 13.1; a diagram of the data flow is shown in Figure 13.1):

$$K_s[T_{c,s}] = K_s[s, c, \textit{addr}, \textit{timestamp}, \textit{lifetime}, K_{c,s}]. \tag{13.3}$$

Since only Kerberos and the service share the private key K_s, the ticket is known to be authentic. The ticket contains a new private session key, $K_{c,s}$, known to the client as well; this key may be used to encrypt transactions during the session. (Technically speaking, $K_{c,s}$ is a *multi-session key*, since it is used for all contacts with that server during the life of the ticket.) To guard against *replay attacks*, all tickets presented are accompanied by an *authenticator*:

$$K_{c,s}[A_c] = K_{c,s}[c, \textit{addr}, \textit{timestamp}]. \tag{13.4}$$

[3]This section is largely taken from [Bellovin and Merritt, 1991].

Table 13.1: Kerberos Notation

c	Client principal
s	Server principal
tgs	Ticket-granting server
K_x	Private key of "x"
$K_{c,s}$	Session key for "c" and "s"
$K_x[\textit{info}]$	"*info*" encrypted in key K_x
$K_s[T_{c,s}]$	Encrypted ticket for "c" to use "s"
$K_{c,s}[A_c]$	Encrypted authenticator for "c" to use "s"
$addr$	Client's IP address

This is a brief string encrypted in the session key and containing a timestamp; if the time does not match the current time within the (predetermined) clock skew limits, the request is assumed to be fraudulent.

The key $K_{c,s}$ can be used to encrypt and/or authenticate individual messages to the server. This is used to implement functions such as encrypted file copies, remote login sessions, etc. Alternatively, $K_{c,s}$ can be used for MAC computation for messages that must be authenticated, but not necessarily secret.

For services where the client needs bidirectional authentication, the server can reply with

$$K_{c,s}[\textit{timestamp} + 1]. \tag{13.5}$$

This demonstrates that the server was able to read *timestamp* from the authenticator, and hence that it knew $K_{c,s}$; that in turn is only available in the ticket, which is encrypted in the server's private key.

Tickets are obtained from the TGS by sending a *request*

$$s, K_{tgs}[T_{c,tgs}], K_{c,tgs}[A_c]. \tag{13.6}$$

In other words, an ordinary ticket/authenticator pair is used; the ticket is known as the *ticket-granting ticket*. The TGS responds with a ticket for server s and a copy of $K_{c,s}$, all encrypted with a private key shared by the TGS and the principal:

$$K_{c,tgs}[K_s[T_{c,s}], K_{c,s}]. \tag{13.7}$$

The session key $K_{c,s}$ is a newly chosen random key.

The key $K_{c,tgs}$ and the ticket-granting ticket are obtained at session-start time. The client sends a message to Kerberos with a principal name; Kerberos responds with

$$K_c[K_{c,tgs}, K_{tgs}[T_{c,tgs}]]. \tag{13.8}$$

The client key K_c is derived from a noninvertible transform of the user's typed password. Thus, all privileges depend ultimately on this one key. Note that servers must possess private keys of

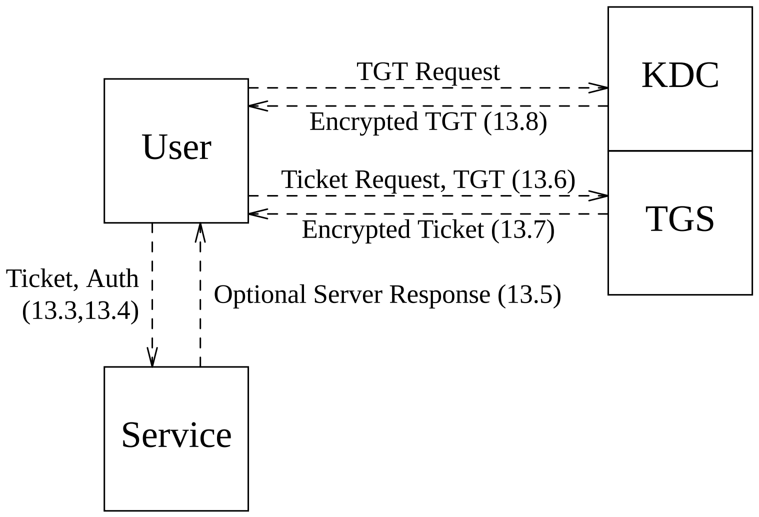

Figure 13.1: Data flow in Kerberos. The message numbers refer to the equations in the text.

their own, in order to decrypt tickets. These keys are stored in a secure location on the server's machine.

Tickets and their associated client keys are cached on the client's machine. Authenticators are recalculated and reencrypted each time the ticket is used. Each ticket has a maximum lifetime enclosed; past that point, the client must obtain a new ticket from the TGS. If the ticket-granting ticket has expired, a new one must be requested, using K_c.

Connecting to servers outside of one's realm is somewhat more complex. An ordinary ticket will not suffice, since the local KDC will not have a secret key for each and every remote server. Instead, an inter-realm authentication mechanism is used. The local KDC must share a secret key with the remote server's KDC; this key is used to sign the local request, thus attesting to the remote KDC that the local one believes the authentication information. The remote KDC uses this information to construct a ticket for use on one of its servers.

This approach, though better than one that assumes one giant KDC, still suffers from scale problems. Every realm needs a separate key for every other realm to which its users need to connect. To solve this, newer versions of Kerberos use a hierarchical authentication structure. A department's KDC might talk to a university-wide KDC, and it in turn to a regional one. Only the regional KDCs would need to share keys with each other in a complete mesh.

13.2.1 Limitations

Although Kerberos is extremely useful, and far better than the address-based authentication methods that most earlier protocols used, it does have some weaknesses and limitations [Bellovin and Merritt, 1991]. First and foremost, Kerberos is designed for user-to-host authentication, not host-to-host. That is reasonable in the Project Athena environment of anonymous, dataless workstations and large-scale file and mail servers; it is a poor match for peer-to-peer environments where hosts have identities of their own and need to access resources such as remotely mounted file systems on their own behalf. To do so within the Kerberos model would require that hosts maintain secret K_c keys of their own, but most computers are notoriously poor at keeping long-term secrets [Morris and Thompson, 1979; Diffie and Hellman, 1976].

A related issue has to do with the ticket and session key cache. Again, multiuser computers are not that good at keeping secrets. Anyone who can read the cached session key can use it to impersonate the legitimate user; the ticket can be picked up by eavesdropping on the network, or by obtaining privileged status on the host. This lack of host security is not a problem for a single-user workstation, to which no one else has any access—but that is not the only environment in which Kerberos is used.

The authenticators are also a weak point. Unless the host keeps track of all previously used live authenticators, an intruder could replay them within the comparatively coarse clock skew limits. For that matter, if the attacker could fool the host into believing an incorrect time of day, the host could provide a ready supply of postdated authenticators for later abuse.

The most serious problems, though, have to do with the way the initial ticket is obtained. First, the initial request for a ticket-granting ticket contains no authentication information, such as an encrypted copy of the user name. The answering message (13.8) is suitable grist for a password-cracking mill; an attacker on the far side of the Internet could build a collection of encrypted ticket-granting tickets and assault them off-line. The latest versions of the Kerberos protocol have some mechanisms for dealing with this problem. More sophisticated approaches detailed in [Lomas *et al.*, 1989] or [Bellovin and Merritt, 1992] can be used.

There is a second login-related problem: how does the user know that the login command itself has not been tampered with? The usual way of guarding against such attacks is to use challenge/response authentication devices, but those are not supported by the current protocol. There are some provisions for extensibility; however, since there are no standards for such extensions, there is no interoperability.

13.3 Link-Level Encryption

Link-level encryption is the most transparent form of cryptographic protection. Indeed, it is often implemented by outboard boxes; even the device drivers, and of course the applications, are unaware of its existence.

As its name implies, this form of encryption protects an individual link. This is both a strength and a weakness. It is strong, because (for certain types of hardware) the entire packet is encrypted, including the source and destination addresses. This guards against *traffic analysis*, a form of intelligence that operates by noting who talks to whom. Under certain circumstances—for example, the encryption of a point-to-point link—even the existence of traffic can be disguised.

However, link encryption suffers from one serious weakness: it protects exactly one link at a time. Messages are still exposed while passing through other links. Even if they, too, are protected

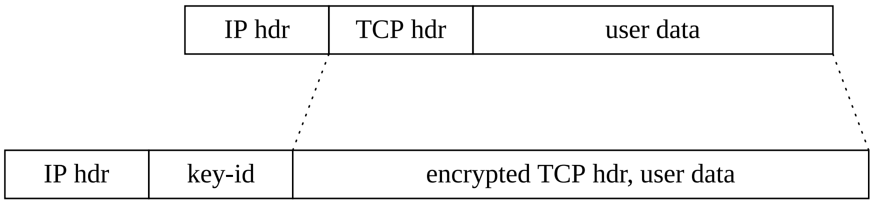

Figure 13.2: Transport-level encryption.

by encryptors, the messages remain vulnerable while in the switching node. Depending on who the enemy is, this may be a serious drawback.

Link encryption is the method of choice for protecting strictly local traffic (i.e., on one shared coaxial cable) or for protecting a small number of highly vulnerable lines. Satellite circuits are a typical example, as are transoceanic cable circuits that may be switched to a satellite-based backup at any time.

13.4 Network- and Transport-Level Encryption

Network- and transport-level encryption are, in some sense, the most useful ways to protect conversations. Like application-level encryptors, they allow systems to converse over existing insecure Internets; like link-level encryptors, they are transparent to most applications. This power comes at a price, though: deployment is difficult because the encryption function affects all communications among many different systems.

Although standard versions of these protocols are not yet available for the TCP/IP protocol suite, the corresponding OSI protocols are being used as models and the basic structure is clear. This exposition is based on the current draft international standards [ISO, 1991, 1992]; the corresponding proposals for TCP/IP and for *Secure Data Network Systems* (*SDNS*) drafts [SP3, 1988; SP4, 1988] are similar.

Both protocols rely on the concept of a *key-id*. The key-id, which is transmitted in the clear with each encrypted packet, controls the behavior of the encryption and decryption mechanisms. It specifies such things as the encryption algorithm, the encryption block size, what integrity check mechanism should be used, the lifetime of the key, etc. A separate key management protocol is used to exchange keys and key-ids.

The *Network Layer Security Protocol* (*NLSP*) and *Transport Layer Security Protocol* (*TLSP*) differ most notably in the granularity of protection. TLSP, as befits its name, is bound to individual connections such as TCP virtual circuits. As such, it provides protection to that level of granularity: different circuits between the same pair of hosts can be protected with different keys.

The entire TCP segment, including the TCP header, but not the IP header, is encrypted (Figure 13.2). This new segment is sent on to IP for the usual processing, albeit with a different protocol identifier. Upon reception, IP will hand the packet up to TLSP, which, after decrypting and verifying the packet, will pass it on to TCP.

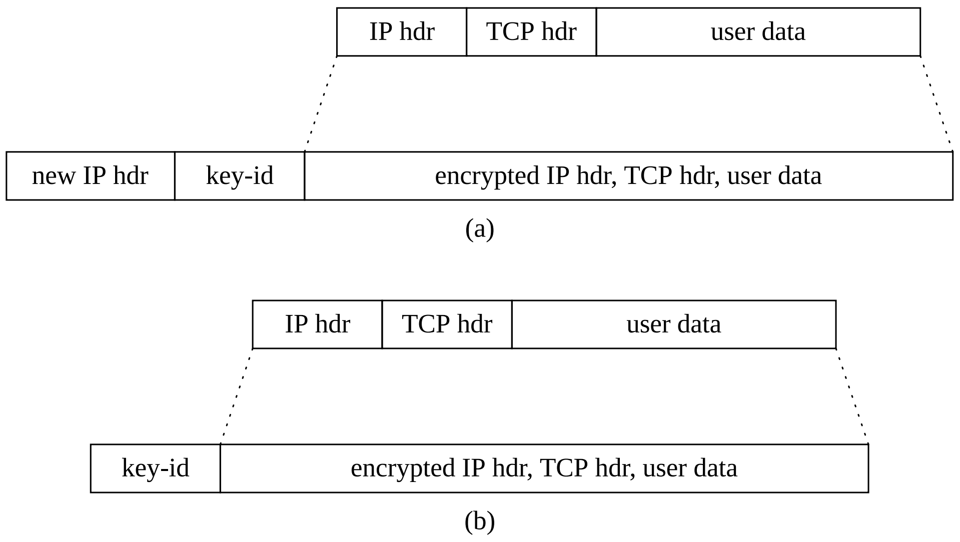
IP hdr
TCP hdr
user data
new IP hdr
key-id
encrypted IP hdr, TCP hdr, user data
(a)
IP hdr
TCP hdr
user data
key-id
encrypted IP hdr, TCP hdr, user data
(b)

Figure 13.3: Network-level encryption.

Note that TCP's error-checking, and hence acknowledgments, takes place *after* decryption and processing. Thus, packets damaged or deleted due to enemy action will be retransmitted via the normal mechanisms. Contrast this with an encryption system that operated above TCP, where an additional retransmission mechanism might be needed.

NLSP offers more choices in placement than does TLSP. Depending on the exact needs of the organization, NLSP may be installed above, in the middle of, or below IP. Indeed, it may even be installed in a gateway router and thus protect an entire subnet.

NLSP operates by encapsulation or tunneling. A packet to be protected is encrypted; following that, a new IP header is attached (Figure 13.3a). The IP addresses in this header may differ from those of the original packet. Specifically, if a gateway router is the source or destination of the packet, its IP address is used. A consequence of this policy is that if NLSP gateways are used at both ends, the real source and destination addresses are obscured, thus providing some defense against traffic analysis. Furthermore, these addresses need bear no relation to the outside world's address space, although that is an attribute that should not be used lightly.

If the two endpoints of an NLSP communication are attached to the same network, creation of the new IP header can be omitted, and the encrypted NLSP packet sent directly over the underlying medium (Figure 13.3b). This is, of course, quite close to link-level encryption; the crucial difference is that link control fields (i.e., checksums, addresses, HDLC framing, etc.) are not encrypted.

The granularity of protection provided by NLSP depends on where it is placed. A host-resident NLSP can, of course, guarantee the actual source host, though not the individual process or user. By contrast, router-resident implementations can provide no more assurance than that the message originated somewhere in the protected subnet. Nevertheless, that is often sufficient, especially if the machines on a given LAN are tightly coupled. Furthermore, it isolates the crucial cryptographic variables into one box, a box that is much more likely to be physically protected than is a typical workstation.

This is shown in Figure 13.4. Encryptors (labeled "E") can protect hosts on a LAN (A1 and A2), on a WAN (C), or an entire subnet (B1, B2, D1, and D2). When host A1 talks to A2 or C, it is assured of the identity of the destination host. Each such host is protected by its own encryption unit. But when A1 talks to B1, it knows nothing more than that it is talking to something behind Net B's encryptor. This could be B1, B2, or even D1 or D2.

One further caveat should be mentioned. Nothing in Figure 13.4 implies that any of the protected hosts actually can talk to each, or that they are unable to talk to unprotected host F. The allowable patterns of communication are an administrative matter; these decisions are enforced by the encryptors and the key distribution mechanism.

Other possibilities exist for network-level encryption. One such protocol, *swIPe*, is described in [Ioannidis and Blaze, 1993]. Others exist as draft RFCs.

13.5 Application-Level Encryption

Performing encryption at the application level is the most intrusive option. It is also the most flexible, because the scope and strength of the protection can be tailored to meet the specific needs of the application. Encryption and authentication options have been defined for a number of high-risk applications, though as of this writing none are widely deployed. We will review a few of them, though there is ongoing work in other areas, such as authenticating routing protocols.

13.5.1 The *Telnet* Protocol

The most critical area, in the sense that a serious ongoing problem already exists, is the *telnet* protocol. The risk is not so much the compromise of the full contents of a remote login session, (although that may be sensitive enough), but the strong need to prevent passwords from being sent over the Internet in the clear. Clearly, that could be prevented by encrypting the *telnet* sessions. That is not the preferred solution, however, for several reasons. First, and most obvious, encryption is often overkill. It is only the password that needs protecting in many cases, not the entire connection; there is no point to incurring the overhead the rest of the time. Second, encryption requires key distribution, and that in turn often requires some sort of authentication. For example, Kerberos will not distribute session keys until after authenticating the user; however, once that is done, there is an elegant mechanism that provides key distribution as part of the authentication process.

The third reason is that we need authentication in *telnet* anyway. This would provide for preauthenticated connections, along the lines of *rlogin*, but built on a more flexible footing. That is, it is desirable to permit users to connect to other machines on the network without the bother of reentering passwords.

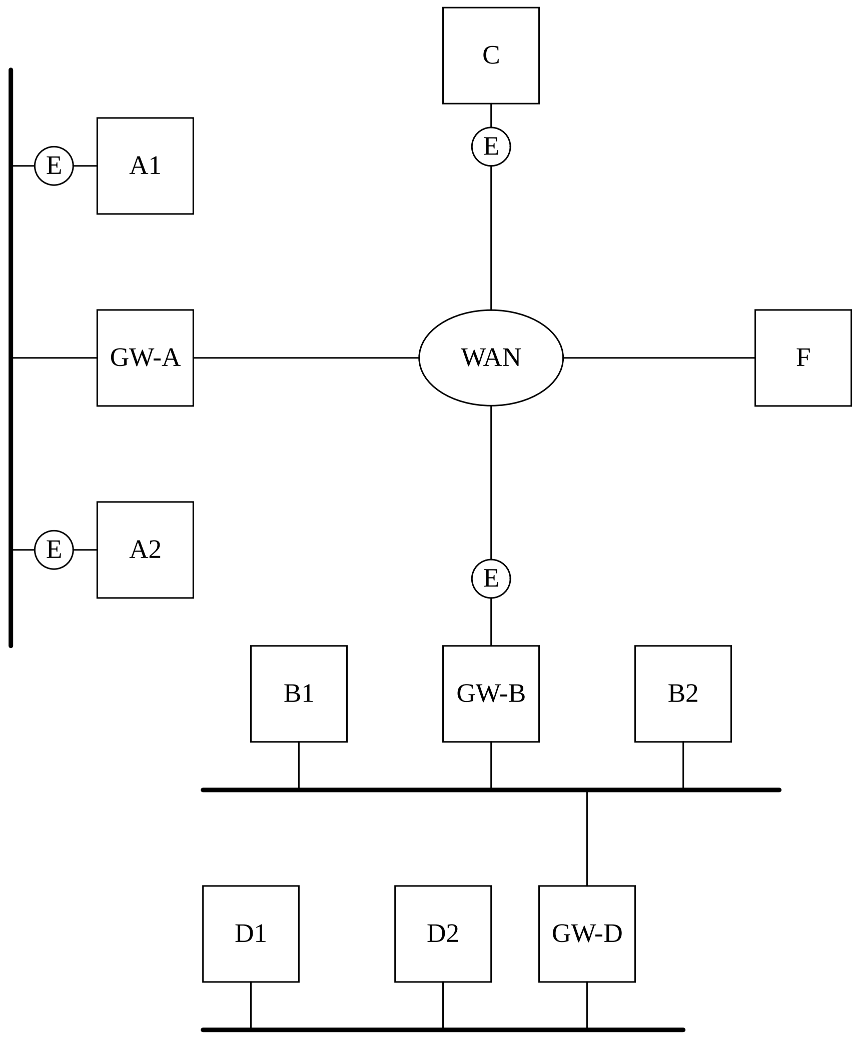

Figure 13.4: Possible configurations with NLSP.

Finally, as we have mentioned earlier, the export and use of encryption software is often heavily restricted. Authentication technology is not controlled to the same degree; it is easier to deploy and it solves some critical problems.

The framework for authentication is defined in [Borman, 1993b]. The client and server first negotiate whether or not authentication should be performed. If both sides agree, they then negotiate which scheme should be used and which side should be authenticated: the client, the server, or both. Finally, whatever messages are appropriate to the agreed-on option are sent. For Kerberos, for example, a ticket and authenticator are transmitted, with an additional reply used if bilateral authentication is desired [Borman, 1993a].

A *telnet* encryption mechanism has been proposed, but as of early 1994 has not yet been finalized. It is integrated with the authentication option: the two sides negotiate which authentication and encryption algorithms should be used and who should use them. Once that has been negotiated, encryption can be turned on, though it may be turned off during the session. Each direction is encrypted independently; that permits input encryption to remain on, to protect passwords, while permitting output to be sent in the clear for performance reasons.

There is a subtle trap to be wary of here. If encryption is used only during password entry, and if the key and initialization vector are held constant, an enemy can replay the encrypted string without ever knowing its contents. At least one of these values must be changed for each session.

Because remote login sessions generally involve character-at-a-time interactions, the CFB and OFB modes of operation are generally used. Error propagation is not at issue, since humans can easily recover from garbles in an interactive session.

13.5.2 Authenticating SNMP

The *Simple Network Management Protocol* (*SNMP*) [Case *et al.*, 1990] is used to control routers, bridges, and other network elements. The need for authentication of SNMP requests is obvious. What is less obvious, but equally true, is that some packets must be encrypted as well, if for no other reason than to protect key change requests for the authentication protocol. Accordingly, a security option has been developed for SNMP [Galvin *et al.*, 1992].

The first problem, authentication, is solved by using secure hash functions, as in message (13.2) on page 221. Both parties share a 16-byte string, which is prepended to the SNMP request; MD5 is used to generate a 16-byte hash code.

Secrecy is provided by using DES in CBC mode. The “key” actually consists of two 8-byte quantities: the actual DES key and the IV to be used for CBC mode. Note that the IV is constant for the lifetime of a key, which is probably a disadvantage. An MD5 hash is performed on the message for integrity checking.

To prevent replay attacks—situations where an enemy records and replays an old, but valid, message—secure SNMP messages include a timestamp. Messages that appear to be stale are discarded. As a consequence of this, it is unsafe to set a clock backwards, because that would create a window during which replays would be accepted as valid. Accordingly, clock skew is dealt with by advancing the slower clock. If it is ever necessary to reset a clock, the secrecy and authentication keys must be changed as well. Normally, this is done using SNMP itself; a detailed description of how to do this is given in the specification.

13.5.3 Secure Electronic Mail

The previous two sections have focused on matters of more interest to administrators. Ordinary users have most often felt the need for privacy when exchanging electronic mail. Unfortunately, an official solution has been slow in coming, so various unofficial solutions have appeared.

The three main contenders are *Privacy-Enhanced Electronic Mail* (*PEM*), the official standard in the TCP/IP protocol suite; *Pretty Good Privacy* (*PGP*), largely developed outside of the United States to avoid being bound by the RSA patent; and *RIPEM*, based on a free implementation of RSA made available with the consent of the patent owners. All three use the same general structure—messages are encrypted with a symmetric cryptosystem, using keys distributed via a public-key cryptosystem—but they differ significantly in detail.

One significant caveat applies to any of these packages. The security of mail sent and received is critically dependent on the security of the underlying operating system. It does no good whatsoever to use the strongest cryptosystems possible if an intruder has booby-trapped the mail reader or can eavesdrop on passwords sent over a local network. For maximum security, any secure mail system should be run on a single-user machine that is protected physically as well as electronically.

PEM

PEM [Linn, 1993b; Kent, 1993; Balenson, 1993; Kaliski, 1993] has the most elaborate structure. Multiple encryption, hash, and public-key algorithms are supported. At present DES is the only standard encryption algorithm for the body of a message, though a triple-DES version is being developed. RSA is used to encrypt the DES keys and certificates. The message is hashed with MD5 prior to signing; certificates are hashed with MD2, a function that is slower but believed to be more secure.

Certificates for PEM contain a fair amount of information. Apart from the usual identifiers and organizational affiliations, they also contain expiration dates. This is partly for conventional cryptographic reasons, partly so that organizational affiliation data remains current, and—within the United States—partly to enforce payment of royalties for use of RSA.

The certification structure is a “forest,” a group of trees. Each root, known as a *Top-Level Certifying Authority*, must cross-certify all other roots. Elaborate trustworthiness and inspection requirements are imposed on lower level certifying authorities, to prevent their possible compromise. On the other hand, provision is explicitly made for *anonymous certificates* which are issued to individuals who wish to be able to sign and receive mail messages using pseudonyms.

Currently, no key server protocols are defined. User mail agents are supposed to cache received certificates. One special form of mail message, the *Certificate Revocation List* (*CRL*), is defined. Such messages contain lists of certificates that are no longer valid. Any authority that issues certificates may generate cancellation messages for the certificates it has issued.

PEM supports a number of different modes of operation. First, there is encrypted, signed mail. One can also send authenticated cleartext messages. Preferably such messages are encoded before being signed or transmitted to avoid problems with varying character sets, line representations, etc. But that would require some software support on the recipient’s end to decode even nonencrypted messages. Accordingly, PEM supports signed but unencoded messages, albeit with the warning that the verification process may fail.

RIPEM

RIPEM [Riordan, 1992] is more or less a subset of PEM. It uses the same message formats and the same basic encryption algorithms: DES and triple DES in CBC mode, MD5 for message-hashing, and RSA for signatures and message key distribution. The primary difference is that RIPEM does not implement certificates. That is, public keys are not signed by anyone; each user of RIPEM must assess the validity of any given key.

RIPEM keys are distributed by a variety of mechanisms, including the *finger* command and a dedicated key server. The code also supports inclusion of the keys in the message itself. Since such keys are not signed, this method is of dubious security. Users can maintain their own collection of keys. These can be gathered from a variety of sources, including RIPEM-signed mail messages. A version of RIPEM that will support signed certificates is under development; it should be available by mid-1994.

Because RIPEM is based on the RSAREF package, use of it is restricted by the provisions of the RSAREF license.

PGP

The PGP package [Zimmerman, 1992] differs from the others in not using DES. Rather, it encrypts messages using IDEA. The MD5 algorithm is used for message-hashing.

The most intriguing feature of PGP is its certificate structure. Rather than being hierarchical, PGP supports a more or less arbitrary "trust graph." Users receive signed key packages from other users; when adding these packages to their own *keyrings*, they indicate the degree of trust they have in the signer, and hence the presumed validity of the enclosed keys. Note that an attacker can forge a chain of signatures as easily as a single one. Unless you have independent verification of part of the chain, there is little security gained from a long sequence of signatures.

Use of the free PGP within the United States has been somewhat controversial, notably because of its unauthorized use of RSA. But at least one vendor has negotiated proper licensing agreements with the various patent holders, and is offering a commercial version for various platforms. These include MS-DOS and several versions of UNIX. Also, MIT has recently made available a version of PGP based on the RSAREF package; use of this version is legal for noncommercial purposes. However, the messages it generates after September 1, 1994 will not be readable by older versions of PGP.

13.5.4 Generic Security Service Application Program Interface

The *Generic Security Service Application Program Interface* (*GSS-API*) [Linn, 1993a; Wray, 1993], is a common interface to a variety of security mechanisms. The idea is to provide programmers with a single set of function calls to use, and also to define a common set of primitives that can be used for application security. Thus, individual applications will no longer have to worry about key distribution or encryption algorithms; rather, they will use the same standard mechanism.

GSS-API is designed for credential-based systems, such as Kerberos or DASS [Kaufman, 1993]. It says nothing about how such credentials are to be acquired in the first place; that is left up to the underlying authentication system.

Naturally, GSS-API does not guarantee interoperability unless the two endpoints know how to honor each other's credentials. In that sense, it is an unusual type of standard in the TCP/IP community: it specifies host behavior, rather than what goes over the wire.