18

Secure Communications over Insecure Networks

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 authentication provided by cryptographic techniques. That is, we wish to utilize mechanisms that will prevent an attacker from forging messages.

This chapter concentrates on how to *use* cryptography for practical network security. It assumes some knowledge of modern cryptography. You can find a brief tutorial on the subject in Appendix A. See [Kaufman *et al.*, 2002] for a detailed look at cryptography and network security.

We first discuss the Kerberos Authentication System. Kerberos is an excellent package, and the code is widely available. It's an IETF Proposed Standard, and it's part of Windows 2000. These things make it an excellent case study, as it is a real design, not vaporware. It has been the subject of many papers and talks, and enjoys widespread use

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 systems. Accordingly, Sections 18.2, 18.3, and 18.4 discuss these 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] or [Bellovin, 1996]. Some examples of this phenomenon are

discussed in Section 18.1 and in the sidebar on page 336.) 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 is regulated by the government. Additionally, many useful cryptosystems are protected by a variety of patents. It may be wise to seek competent legal advice.

18.1 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.[1] 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 (KDC)*; 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 3-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 secret key of the service. (Notation is summarized in Table 18.1. A diagram of the data flow is shown in Figure 18.1; the message numbers in the diagram correspond to equation numbers in the text.)

$$K_s[T_{c,s}] = K_s[s, c, \textit{addr}, \textit{timestamp}, \textit{lifetime}, K_{c,s}] \tag{18.1}$$

Because only Kerberos and the service share the secret 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*, as 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*:

$$K_{c,s}[A_c] = K_{c,s}[c, \textit{addr}, \textit{timestamp}] \tag{18.2}$$

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

Table 18.1: 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, and so on. Alternatively, $K_{c,s}$ can be used for *message authentication code (MAC)* computation for messages that must be authenticated, but not necessarily secret.

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

$$K_{c,s}[\textit{timestamp} + 1] \tag{18.3}$$

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

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

$$s, K_{tgs}[T_{c,tgs}], K_{c,tgs}[A_c] \tag{18.4}$$

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:

$$K_{c,tgs}[K_s[T_{c,s}], K_{c,s}] \tag{18.5}$$

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

$$K_c[K_{c,tgs}, K_{tgs}[T_{c,tgs}]] \tag{18.6}$$

The client key K_c is derived from a non-invertible transform of the user's typed password. Thus, all privileges depend ultimately on this one key. (This, of course, has its weaknesses; see [Wu,

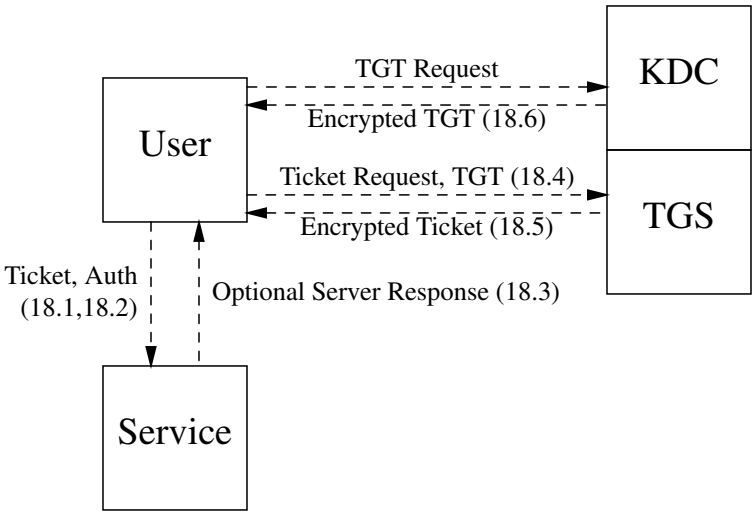

Figure 18.1: Data flow in Kerberos. The message numbers refer to the equations in the text.

1999].) Note that servers must possess secret keys of 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, as 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.

18.1.1 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 was reasonable in the Project Athena environment of anonymous, diskless 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]. (Of course, if they can't keep some secrets, they can't participate in any secure authentication dialog. There's a lesson here: Change your machines' keys frequently.)

A related issue involves the ticket and session key cache. Again, multi-user 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. Kerberos also suffers from a cascading failure problem. Namely, if the KDC is compromised, all traffic keys are compromised.

The most serious problems, though, result from the way in which 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 username. The answering message (18.6) 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 offline. 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 [Wu, 1999]. There is also ongoing work on using public key cryptography for the initial authentication.

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, as there are no standards for such extensions, there is no interoperability.

Microsoft has extended Kerberos in a different fashion. They use the vendor extension field to carry Windows-specific authorization data. This is nominally standards-compliant, but it made it impossible to use the free versions of Kerberos as KDCs in a Windows environment. Worse yet, initially Microsoft refused to release documentation on the format of the extensions. When they did, they said it was "informational," and declined to license the technology. To date, there are no open-source Kerberos implementations that can talk to Microsoft Kerberos. For more details on compatibility issues, see [Hill, 2000].

18.2 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 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 either strictly local traffic (i.e., on one shared coaxial cable) or 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.

The best-known link encryption scheme is *Wired Equivalent Privacy (WEP)* (see Section 2.5); its failures are independent of the general problems of link encryption.

18.3 Network-Level Encryption

Network-level encryption is, in some sense, the most useful way to protect conversations. Like application-level encryptors, it allow systems to converse over existing insecure Internets; like link-level encryptors, it is 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.

The network-layer encryption mechanism for the Internet is known as IPsec [Kent and Atkinson, 1998c; Thayer *et al.*, 1998]. IPsec includes an encryption mechanism (*Encapsulating Security Protocol (ESP)*) [Kent and Atkinson, 1998b]; an authentication mechanism (*Authentication Header (AH)*) [Kent and Atkinson, 1998a]; and a key management protocol (*Internet Key Exchange (IKE)*) [Harkins and Carrel, 1998].

18.3.1 ESP and AH

ESP and AH rely on the concept of a *key-id*. The key-id (known in the spec as a *Security Parameter Index (SPI)*), 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, and so on. The choices made for any particular packet depend on the two sites' security policies, and often on the application as well.

The original version of ESP did encryption only. If authentication was desired, it was used in conjunction with AH. However, a number of subtle yet devastating attacks were found [Bellovin,

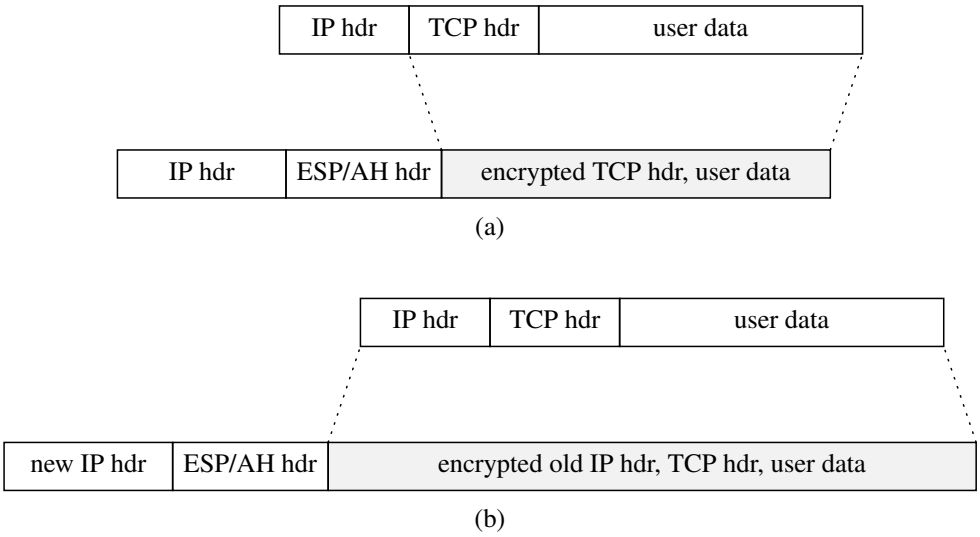

Figure 18.2: Network-level encryption.

1996]. Accordingly, ESP now includes an authentication field and an anti-replay counter, though both are optional. (Unless you *really* know what you're doing, and have a *really* good reason, we strongly suggest keeping these enabled.) The anti-replay counter is an integer that starts at zero and counts up. It is not allowed to wrap around; if it hits 2^{32}, the systems must rekey (see below).

AH can be used if only the authenticity of the packet is in question. A telecommuter who is not working with confidential data could, for example, use AH to connect through the firewall to an internal host. On output from the telecommuter's machine, each packet has an AH header prepended; the firewall will examine and validate this, strip off the AH header, and reinject the validated packet on the inside.

Packets that fail the integrity or replay checks are discarded. 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 operates above TCP, where an additional retransmission mechanism might be needed.

The ESP design includes a "null cipher" option. This provides the other features of ESP—authentication and replay protection—while not encrypting the payload. The null cipher variant is thus quite similar to AH. The latter, however, protects portions of the *preceding* IP header. The need for such protection is quite debatable (and we don't think it's particularly useful); if it doesn't matter to you, stick with ESP.

IPsec offers many choices for placement. Depending on the exact needs of the organization, it 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.

IPsec can operate by encapsulation or tunneling. A packet to be protected is encrypted; following that, a new IP header is attached (see Figure 18.2a). 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 IPsec 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.

The granularity of protection provided by IPsec depends on where it is placed. A host-resident IPsec can, of course, guarantee the actual source host, though often 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 18.3. Encryptors (labeled "E") can protect hosts on a LAN (A1 and A2), on a WAN (C), or on 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.

Protection can be even finer-grained than that. A *Security Policy Database (SPD)* can specify the destination addresses and port numbers that should be protected by IPsec. Outbound packets matching an SPD entry are diverted for suitable encapsulation in ESP and/or AH. Inbound packets are checked against the SPD to ensure that they are protected if the SPD claims they should be; furthermore, they must be protected with the proper SPI (and hence key). Thus, if host A has an encrypted connection to hosts B and C, C cannot send a forged packet claiming to be from B but encrypted under C's key.

One further caveat should be mentioned. Nothing in Figure 18.3 implies that any of the protected hosts actually can talk to one another, 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.

Currently, each vendor implements its own scheme for describing the SPD. A standardized mechanism, called *IP Security Policy (IPSP)*, is under development.

Details about using IPsec in a VPN are discussed in Section 12.2.

18.3.2 Key Management for IPsec

A number of possible key management strategies can be used with IPsec. The simplest is static keying: The administrator specifies the key and protocols to be used, and both sides just use them, without further ado. Apart from the cryptanalytic weaknesses, if you use static keying, you can't use replay protection.

Most people use a key management protocol. The usual one is *Internet Key Exchange (IKE)* [Harkins and Carrel, 1998], though a Kerberos-based protocol (*Kerberized Internet Negotiation of Keys (KINK)*) is under development [Thomas and Vilhuber, 2002]. IKE can operate with either certificates or a shared secret. Note that this shared secret is not used directly as a key; rather, it is used to authenticate the key agreement protocol. As such, features like anti-replay are available.

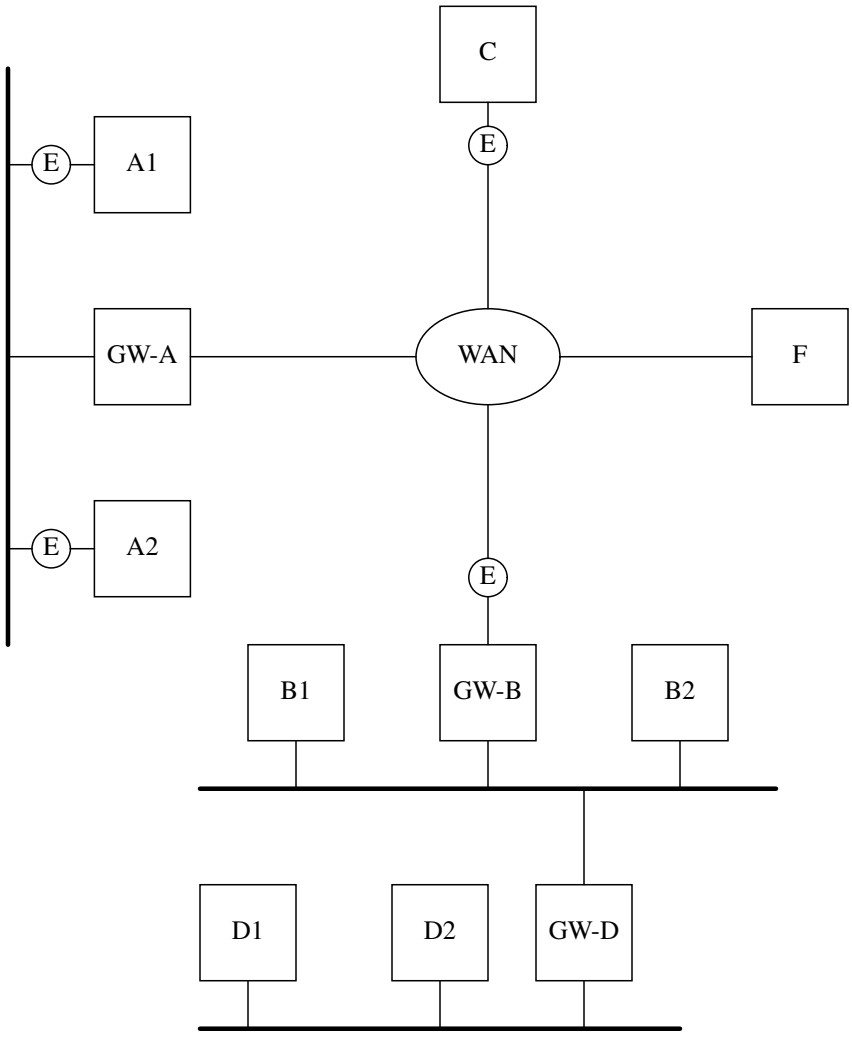

Figure 18.3: Possible configurations with IPsec.

Certificate-based IKE is stronger still, as one end doesn't need to know the other end's secret. Unfortunately, differences in certificate contents and interpretation between different vendors has made interoperability difficult. The complexity of IKE itself—in addition to key agreement, it can negotiate *security associations (SAs)*, add security associations to existing SAs, probe for dead peers, delete SAs, and so on—has also contributed to this problem.

Work is proceeding on several fronts to address these issues. The IETF's *Public Key Infrastructure (X.509) (PKIX)* working group is trying to standardize certificates; see [Adams and Farrell, 1999; Myers *et al.*, 1999] and the group's Web page (`http://www.ietf.org/html.charters/pkix-charter.html`) for a further list. There is also work to produce a so-called "IKEv2" key management protocol; while at press time the design is still in flux, there is little doubt it will be significantly simpler and (we hope) more interoperable.

18.4 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.

18.4.1 Remote Login: *Ssh*

Ssh, the Secure Shell [Ylönen, 1996], has become an extremely popular mechanism for secure remote login. Apart from its intrinsic merits, *ssh* was developed in (and is available from) Finland, a country with no restrictions on the export of cryptography. At its simplest, *ssh* is a more or less plug-compatible replacement for *rlogin*, *rsh*, and *rcp*, save that its authentication is cryptographic and the contents of the conversation are protected from eavesdropping or active attacks. It can do far more.

The most important extra ability of *ssh* is port-forwarding. That is, either the client or the server can bind a socket to a set of specified ports; when someone connects to those ports, the request is relayed to the other end of the *ssh* call, where a call is made to some other predefined host and port. In other words, *ssh* has a built-in tunnel mechanism.

As with all tunnels (see Section 12.1), this can be both good and bad. We sometimes use *ssh* to connect in through our firewall; by forwarding the strictly local instances of the SMTP, POP3, and WWW proxy ports, we can upload and download mail securely, and browse internal Web sites. Conversely, someone who wanted to could just as easily set up an open connection to an internal *telnet* server—or worse.

When *ssh* grants access based on public keys, certificates are not used; rather, the public key stands alone in the authorization files. Depending on how it is configured (and there are far too many configuration options), authentication can be host-to-host, as with the *r* commands, or user-to-host. In fact, *ssh* can even be used with conventional passwords, albeit over an encrypted connection. If user-to-host authentication is used, the user's private key is used to sign the con-

nection request. This key is stored in encrypted form on the client host; a typed passphrase is used to decrypt it.

Ssh can also forward the X11 port and the "authentication channel." These abilities are potentially even more dangerous than the usual port-forwarding.

The former permits remote windows to be relayed over a protected channel. It uses X11's magic cookie authentication technique to ward off evildoers on the remote machine. If the destination machine itself has been subverted, the Bad Guys can set up an X11 connection back to your server, with all that implies—see Section 3.11 for the gory details. In other words, you should never use this capability unless you trust the remote machine.

The same is true for the authentication channel. The authentication channel is *ssh*'s mechanism for avoiding the necessity of constantly typing your passphrase. The user runs *ssh-agent*, which sets up a file descriptor that is intended to be available only to that user's processes. Any new invocations of *ssh* can use this file descriptor to gain access to the private key. The ability to forward this channel implies that after a login to a remote machine, *ssh* commands on it can gain similar access. Again, if the remote machine has been subverted, you're in trouble—your cryptographically secure login mechanism has been compromised by someone who can go around the cipher and use your own facilities to impersonate you to any other machines that trust that key. The remedy is the same as with X11 forwarding, of course: Don't forward the authentication channel to any machines that you don't fully trust.

There is a mechanism whereby *ssh* keeps track of host public keys of remote *ssh* servers. The first time a user connects to a remote machine over *ssh*, he or she is shown the public key fingerprint of the server and asked if the connection should be continued. If the user responds in the affirmative, then the public key is stored in a file called `known-hosts`. Then, if the public key ever changes, either maliciously or by legitimate administration, the user is prompted again. The hope is that security-conscious users might hesitate and investigate if the public key changes.

Ssh uses a variety of different symmetric ciphers, including triple DES and IDEA, for session encryption. Your choice will generally depend on patent status, performance, and your paranoia level.

An IETF working group is developing a new version of *ssh*. Due to limitations of the current protocol, the new one will not be backwards-compatible.

18.4.2 SSL—The Secure Socket Layer

SSL is the standard for securing transactions on the Web. The IETF adopted the protocol and named its version the *Transport Layer Security (TLS)* protocol [Dierks and Allen, 1999]. We refer to the protocol as SSL, but all of our comments apply to both protocols. For an excellent introduction to both protocols, see [Rescorla, 2000b].

There are two purposes for the protocol. The first is to provide a confidentiality pipe between a browser and a Web server. The second is to authenticate the server, and possibly the client. Right now, client authentication is not very common, but that should change in the near future, in particular for intranet applications.

Protocol Overview

Servers supporting SSL must generate a public/private RSA key pair and obtain a certificate for the public key. The certificate must be issued by one of the root authorities that has its public signing key in the standard browsers. Popular browsers have hundreds of such keys, begging the question of whom exactly does everybody trust?

The certification authorities with root public keys in the browsers charge money for the service of verifying someone's identity and signing his or her public key. In return for this payment, they issue a certificate needed to support SSL. The certificate is simply a signed statement containing the public key and the identity of the merchant, in a special format specified in the protocol.

When a user connects to a secure server, the browser recognizes SSL from the URL, which starts with `https://` instead of `http://`, and initiates the SSL protocol on port 443 of the server, instead of the default port 80. The client initiates SSL by sending a message called the `SSL ClientHello` message to the server. This message contains information about the parameters that the client supports. In particular, it lists the cryptographic algorithms and parameters (called CipherSuites), compression algorithms, and SSL version number that it is running. Note that of all the major implementations of SSL, only OpenSSL implements compression.

The server examines the CipherSuites and compression algorithms from the client and compares them with its own list. Of the CipherSuites that they have in common, it then selects the most secure. The server informs the client of the chosen CipherSuite and compression algorithm and assigns a unique *session ID* to link future messages to this session. (In version 2, the client suggested a CipherSuite, the server pruned, and the client chose.) The purpose of the session ID is to allow the reuse of these keys for some time, rather than generating new ones for every communication. This reduces the computational load on the client and the server. The next step involves picking the keys that protect the communication.

Once the CipherSuite is set, the server sends its certificate to the client. The client uses the corresponding root public key in the browser to perform a digital signature verification on the certificate. If the verification succeeds, the client extracts the public key from the certificate and checks the DNS name against the certificate [Rescorla, 2000a]. If they do not match, the user is presented with a pop-up warning. Next, the client generates symmetric key material (random bits), based on the CipherSuite that was chosen by the server. This key material is used to derive encryption and authentication keys to protect the payload between the browser and the server. The client encrypts the symmetric key material with the public key of the server using RSA, and sends it to the server.

The server then uses its private key to decrypt the symmetric key material and derives the encryption and authentication keys. Next, the client and the server exchange messages that contain the MAC of the entire dialogue up to this point. This ensures that the messages were not tampered with and that both parties have the correct key. After the MACs are received and verified, application data is sent, and all future communication during the SSL session is encrypted and MACed. If a client reconnects to a server running SSL after communicating with a different server, and if the original SSL session has not expired, the client sends the previous session ID to indicate it

wants to resume using it. In that case, the messages in the SSL protocol will be skipped, and the keys derived earlier can be used again.

Security

There is more to security than strong cryptographic algorithms and well-designed protocols. Researchers have looked at the design of SSL and the consensus is that it is very good, as cryptographic protocols go [Wagner and Schneier, 1996]. Once you get beyond broken algorithms and protocols and buggy software, the weakest link in the chain often involves the user. SSL provides feedback to the user in the form of a lock icon at the bottom of the browser window. All this means is that the browser is engaging the SSL protocol with *some server*. It does not say anything about which server. The burden is on the user to check the security information on the page to discover who holds the certificate. In fact, all that the user can verify is that a certifying authority, that has a public key in the browser, issued a certificate for some entity, and that there is a certification path from that entity to the entity in the certificate. There is no guarantee that the server who serves a certificate is the entity in the certificate. If the two entities do not match, the browser typically issues a warning, but users often ignore such warnings. In fact, it is rare that users verify certificate information at all.

All sorts of threats can compromise the security of SSL. Attacks against the Domain Name Service (DNS) are very effective against SSL. If someone can map the host name in a URL to an IP address under his control, and if that person can obtain a certificate from any one of the root CAs, then he can provide *secure* service from that site and users have no way of knowing what happened.

To illustrate that it is not enough to assume that everything is secure just because SSL is used, let's look at an example. In early 2000, somebody created a site called PAYPAI.COM—with an I instead of an l—and sent out e-mail linking to the site. The attacker then obtained a certificate for PAYPAI.COM, and sent a message to many addresses indicating that someone had deposited $827 for the recipient, along with a URL to claim the money. As soon as the user logged in to this fake Web site—but with a real username and password—the attacker had captured the login and password of the person's Paypal account. Although the connection was over SSL, people were fooled because the attacker was using a legitimate certificate.

SSL provides a confidential pipe from a client to a server, but the user is responsible for verifying the identity of the server. This is not always possible. Besides the network-level threat, keep in mind that SSL is not a Web panacea. Sensitive data still sits on back-end Web servers, which may be vulnerable to attack, and in client caches. A well-designed virus could traverse client machines, farming the caches for sensitive information.

In summary, SSL is not a magical solution for security on the Web. It is very effective at reducing the ability of eavesdroppers to collect information about Web transactions, and it is the best thing that we have. It is not perfect because it runs in an imperfect world, full of buggy computers and human users.

Though originally designed for the Web, SSL is being used with other protocols. There are, for example, standards for POP3 and IMAP [Newman, 1999] over SSL. Expect to see more of this; it's reasonably easy to plug SSL into most protocols that run over TCP.

18.4.3 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. SNMPv3 has a suitable security mechanism [Blumenthal and Wijnen, 1999].

Authentication is done via HMAC [Krawczyk *et al.*, 1997] with either MD5 [Rivest, 1992b] or SHA-1 [NIST, 1993; Eastlake *et al.*, 2001]. Both parties share a secret key; there is no key management.

Secrecy is provided by using DES in CBC mode. The "key" actually consists of two 8-byte quantities: the actual DES key and a "pre-IV" used to generate the IV used for CBC mode. An AES specification is under development [Blumenthal *et al.*, 2002].

To prevent replay attacks—situations in which an enemy records and replays an old, but valid, message—secure SNMP messages include a timestamp and a message-id field. Messages that appear to be stale must be discarded.

18.4.4 Secure Electronic Mail

The previous two sections 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 was slow in coming, so various unofficial solutions appeared. This, of course, has led to interoperability problems.

The two main contenders are *Secure Multipurpose Internet Mail Extensions (S/MIME)*, developed by RSA Security, and *Pretty Good Privacy (PGP)*. Both 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 either 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.

S/MIME

S/MIME is a mail encryption standard originally developed by RSA Security. However, many different vendors have implemented it under license, especially for Windows platforms. Most notably, it exists in the mailers used by Microsoft IE and Netscape Navigator.

S/MIME uses an X.509-based certificate infrastructure. Each user can decide for himself or herself which certifying authorities should be trusted.

The actual security provided by S/MIME depends heavily on the symmetric cipher used. The so-called "export versions"—rarely shipped these days, given the changes in U.S. export rules—use 40-bit RC4, which is grossly inadequate against even casual attackers.

An IETF working group has been producing new versions of the S/MIME specification, including adding modern ciphers like AES.

PGP

Several different versions of PGP exist. The older versions use IDEA to encrypt messages, MD5 for message hashing, and RSA for message key encryption and signatures. To avoid some patent complications (not all of which matter anymore), some versions can use triple DES or CAST as well as IDEA for encryption, Diffie-Hellman for message key encryption, and the Digital Signature Standard for signing. Additionally, SHA has replaced MD5, as the latter appears to be weaker than previously believed. Recently, the IETF has standardized *OpenPGP* [Callas *et al.*, 1998], which is not bound to any particular implementation.

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.

The freedom of the web of trust notwithstanding, much of the world is moving toward X.509 certificates. This is a probable direction for PGP as well.

With either style of certificate, distribution remains a major problem. There are a number of PGP key servers around the world; keys can be uploaded and downloaded both manually and automatically. Sometimes, a private protocol is used; some use LDAP (see Section 3.8.3.)

18.4.5 Transmission Security vs. Object Security

It's important to make a distinction between securing the transmission of a message and securing the message itself. An e-mail message is an "object" that is likely to be stored in intermediate locations on its way from source to destination. As such, securing its transmission with SSL is of comparatively limited benefit. However, PGP and S/MIME are well-suited to the task, as a digital signature protects the object's authenticity, regardless of how it travels through the network.

By contrast, IPsec and SSL protect a transmission channel and are appropriate for protecting IP packets between two machines, regardless of the contents of the traffic. For point-to-point communication, transmission security is more appropriate. For store-and-forward applications, it is more appropriate to secure the objects themselves.

18.4.6 Generic Security Service Application Program Interface

A common interface to a variety of security mechanisms is the *Generic Security Service Application Program Interface (GSS-API)* [Linn, 2000; Wray, 2000]. 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.