Chapter 20

Authentication for Distributed Systems[1]

Thomas Y. C. Woo and Simon S. Lam

A fundamental concern in building a secure distributed system is authentication of local and remote entities in the system. We survey authentication issues in distributed system design. Two basic paradigms underlying the design of authentication protocols are presented. We then propose an authentication framework that can be used for designing secure distributed systems, including specific protocols for secure bootstrapping, user-host authentication, and peer-peer authentication. We conclude with an overview of two existing authentication systems, namely, Kerberos and SPX.

INTRODUCTION

A *distributed system*—a collection of hosts interconnected by a network—poses some intricate security problems. A fundamental concern is authentication of local and remote entities in the system. In a distributed system, the hosts communicate by sending and receiving messages over the network. Various resources (like files and printers) distributed among the hosts are shared across the network in the form of network services provided by *servers.* Individual processes (*clients*) that desire access to resources direct service requests to the appropriate servers. Aside from such client-server computing, there are many other reasons for having a distributed system. For example, a task can be divided up into subtasks that are executed concurrently on different hosts.

A distributed system is susceptible to a variety of threats mounted by intruders as well as legitimate users of the system. Indeed, legitimate users are more powerful adversaries since they possess internal state information not

[1] *Based on a paper entitled "Authentication for Distributed Systems," Thomas Woo and Simon Lam,* IEEE Computer, *January 1992 (and errata, March 1992). Reprinted with permission.*

usually available to an intruder (except after a successful penetration of a host). We identify two general types of threats.

The first type, *host compromise,* refers to the subversion of individual hosts in a system. Various degrees of subversion are possible, ranging from the relatively benign case of corrupting process state information to the extreme case of assuming total control of a host. Host compromise threats can be countered by a combination of hardware techniques (like processor protection modes) and software techniques (like *security kernel/reference monitor*). These techniques are outside the scope of this chapter; we refer interested readers to [4] for an overview of the area of computer systems security. In this chapter, we assume that each host implements a reference monitor that can be trusted to properly segregate processes.

The second type, *communication compromise,* includes threats associated with message communications. We subdivide these into:

(T1) eavesdropping of messages transmitted over network links to extract information on private conversations;

(T2) arbitrary modification, insertion, and deletion of messages transmitted over network links to confound a receiver into accepting fabricated messages; and

(T3) replay of old messages; this can be considered a combination of (T1) and (T2).

(T1) is a *passive* threat, while (T2) and (T3) are *active* threats. A passive threat does not affect the system being threatened, whereas an active threat does. Therefore, passive threats are inherently undetectable by the system, and can only be dealt with by using preventive measures. Active threats, on the other hand, are combated by a combination of prevention, detection, and recovery techniques.

Additionally, there are threats of "traffic analysis" and "denial of service"; we will not consider them here because they are more relevant to the general security of a distributed system than to our restricted setting of authentication.

Corresponding to these threats, some basic security requirements can be formulated. For example, *secrecy* and *integrity* are two common requirements for secure communication. Secrecy specifies that a message can be read only by its intended recipients, while integrity specifies that every message is received exactly as it was sent, or a discrepancy is detected.

A strong cryptosystem can provide a high level of assurance of both the secrecy and integrity (see "Basic Cryptography" sidebar). More precisely, an encrypted message provides no information regarding the original message, hence guaranteeing secrecy, and if tampered with, would not decrypt into an understandable message, hence guaranteeing integrity.

Replay of old messages can be countered by using *nonces* or *timestamps* [4, 11]. A nonce is information that is guaranteed *fresh,* that is, it has not

Basic Cryptography

A cryptosystem comes with two procedures, one for *encryption* and one for *decryption*. A formal description of a cryptosystem includes specifications for its *message, key, ciphertext spaces,* and encryption and decryption functions.

There are two broad classes of cryptosystems, *symmetric* and *asymmetric* [4, 5]. In the former, the encryption and decryption keys are the same and hence must be kept secret. In the latter, the encryption key differs from the decryption key, and only the decryption key must be kept secret. The encryption key, however, can be made public. Consequently, it is important that no one be able to determine the decryption key from the encryption key. Symmetric and asymmetric cryptosystems are also referred to as *shared key* and *public key* cryptosystems, respectively.

Knowledge of the encryption key allows one to encrypt arbitrary messages in the message space, while knowledge of the decryption key allows one to recover a message from its encrypted form. Thus, the encryption and decryption functions satisfy the following relation: $\mathcal{M}$ is the message space, $K_E \times K_D$ is the set of encryption/decryption key pairs:

$$\forall m \in \mathcal{M} : \forall(k, k^{-1}) \in K_E \times K_D : \{\{m\}_k\}_{k^{-1}} = m \qquad \text{(C1)}$$

where $\{x\}_y$ denotes the encryption operation on message x if y is an encryption key, and the decryption operation on x if y is a decryption key. (In the case of a symmetric cryptosystem with identical encryption and decryption keys, the operation should be clear from the context.)

Two widely used cryptosystems are the Data Encryption Standard (DES) [2], a symmetric system, and RSA [3], an asymmetric system. In RSA, encryption-decryption key pairs satisfy the following commutative property [1]:

$$\forall m \in \mathcal{M} : \forall(k, k^{-1}) \in K_E \times K_D : \{\{m\}_{k^{-1}}\}_k = m \qquad \text{(C2)}$$

hence yielding a *signature* capability. That is, suppose k and k^{-1} are P's asymmetric keys, then $\{m\}_{k^{-1}}$ can be used as P's signature on m since it could only have been produced by P, the only principal that knows k^{-1}. By (C2), P's signature is verifiable by any principal with knowledge of k, P's public key. Note that in (C2), the roles of k and k^{-1} are reversed; specifically, k^{-1} is used as an encryption key while k functions as a decryption key. To avoid confusion with the more typical roles for k and k^{-1} as exemplified in (C1), we refer to encryption by k^{-1} as a *signing* operation. In this chapter, asymmetric cryptosystems are assumed to be commutative.

Since, in practice, symmetric cryptosystems can operate much faster than asymmetric ones, asymmetric cryptosystems are often used only for initialization/control functions, while symmetric cryptosystems can be used for both initializations and actual data transfer.

References

1. W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of IEEE,* 67(3):397–427, March 1979.
2. National Bureau of Standards, US Department of Commerce, Washington, D.C. *Data Encryption Standard FIPS Pub 46,* January 15, 1977.
3. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM,* 21(2):120–126, February 1978.
4. B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, 2nd edition, 1996.
5. G. J. Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys,* 11(4):305–330, December 1979.

appeared or been used before. Therefore, a reply that contains some function of a recently sent nonce should be believed timely because the reply could have been generated only after the nonce was sent. Perfect random numbers are good nonce candidates; however, their effectiveness is dependent upon the randomness that is practically achievable. Timestamps are values of a local clock. Their use requires at least some loose synchronization of all local clocks, and hence their effectiveness is also somewhat restricted.

The balance of this chapter is organized as follows. In the next section, we discuss what authentication means as well as the various authentication needs in a distributed system. Then we describe the different types of authentication exchanges in a distributed system. Following this, we present two paradigms of authentication protocol design and discuss why realistic authentication protocols are difficult to design. Next, we propose an authentication framework for distributed systems, and present specific authentication protocols that can be used within the framework. Finally, we describe authentication protocols in two existing systems: Kerberos and SPX, and then present some conclusions.

WHAT NEEDS AUTHENTICATION?

In simple terms, authentication is identification plus verification. *Identification* is the procedure whereby an entity claims a certain identity, while *verification* is the procedure whereby that claim is checked. Thus the *correctness* of an authentication relies heavily on the verification procedure employed.

The entities in a distributed system that can be distinctly identified are collectively referred to as *principals*. There are three main types of authentication of interest in a distributed system:

(A1) *message content authentication*—verifying that the content of a message received is the same as when it was sent;

(A2) *message origin authentication*—verifying that the sender of a received message is the same one recorded in the sender field of the message; and

(A3) *general identity authentication*—verifying that the principal's identity is as claimed.

(A1) is commonly handled by tagging a key-dependent *message authentication code* (MAC) onto a message before it is sent. Message integrity can be confirmed upon receipt by recomputing the MAC and comparing it with the one attached. (A2) is a subcase of (A3). A successful general identity authentication results in a belief held by the authenticating principal (the *verifier*) that the authenticated principal (the *claimant*) possesses the claimed identity. Hence, subsequent claimant actions are attributable to the claimed identity. General identity authentication is needed for both authorization and accounting functions. In the balance of this chapter, we restrict our attention to general identity authentication only.

In an environment where both host and communication compromises can occur, principals must adopt a mutually suspicious attitude toward one another. Therefore, *mutual* authentication, whereby both communicating principals verify each other's identity, rather than *one-way* authentication, whereby only one principal verifies the identity of the other principal, is usually required.

In a distributed system environment, authentication is carried out using a protocol involving message exchanges. We refer to these protocols as *authentication protocols.*

Most existing systems use only very primitive authentication measures or none at all. For example:

- The prevalent login procedure requires users to enter their passwords in response to a system prompt. Users are then one-way authenticated by verifying the (possibly transformed) password against an internally stored table. However, no mechanism lets users authenticate a system. Such a design is acceptable only when the system is trustworthy, or the probability of compromise is low.
- In a typical client-server interaction, the server—on accepting a client's request—has to trust that (1) the resident host of the client has correctly authenticated the client, and (2) the identity supplied in the request actually corresponds to the client. Such trust is valid only if the system's hosts are trustworthy and its communication channels are secure.

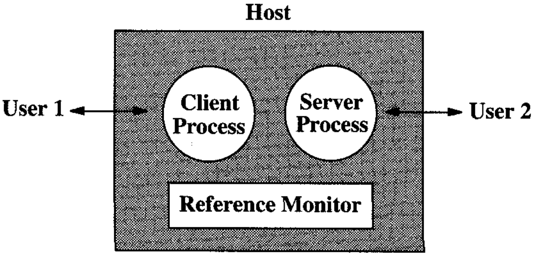


FIGURE 1 Principals in a distributed system.

These measures are seriously inadequate because the notion of trust in distributed systems is poorly understood. A satisfactory formal explication of trust has yet to be proposed. Second, the proliferation of large-scale distributed systems spanning multiple administrative domains has produced extremely complex trust relationships.

In a distributed system, the entities that require identification are hosts, users, and processes [10]. They thus constitute the principals involved in an authentication, which we describe (see also Figure 1).

Hosts. These are addressable entities at the network level. A host is distinguished from its underlying supporting hardware. For example, a host H running on workstation W can be moved to run on workstation W' by performing on W' the bootstrap sequence for H. A host is usually identified by its name (for example, a fully qualified domain name) or its network address (for example, an IP address), whereas a particular host hardware is usually identified by its factory assigned serial number (for example, an ID burned into its boot PROM).

Users. These entities are ultimately responsible for all system activities. In other words, users initiate and are accountable for all system activities. Most access control and accounting functions are based on users. (For completeness, a special user called *root* can be postulated, who is accountable for system-level activities like process scheduling.) Typical users include humans, as well as accounts maintained in the user database. Note that users are considered to be outside the system boundary.

Processes. The system creates processes within the system boundary to represent users. A process requests and consumes resources on behalf of its unique associated user. Processes fall into two classes: client and server. Client processes are consumers who obtain services from server processes, who are service providers. A particular process can act as both a client and a server. For example, print servers are usually created by (and hence associated with) the user *root,* and act as servers for printing requests by other processes. However, they act as clients when requesting files from file servers.

AUTHENTICATION EXCHANGES

For the various principals introduced in the above section, we identify the following major types of authentication exchanges in a distributed system.

Host-host. Host-level activities often require cooperation between hosts. For example, individual hosts exchange link information for updating their internal topology maps. In remote bootstrapping, a host, upon reinitialization, must identify a trustworthy *bootstrap server* to supply the information (for example, a copy of the operating system) required for correct initialization.

User-host. A user gains access to a distributed system by logging into a host in the system. In an open access environment where hosts are scattered across unrestricted areas, a host can be arbitrarily compromised, necessitating mutual authentication between the user and host.

Process-process. Two main subclasses exist:

- Peer-peer communication. Peer processes must be satisfied with each other's identity before private communication can begin.
- Client-server communication. An access decision concerning a client's request can be made only when the client's identity is affirmed. A client is willing to surrender valuable information to a server only after it has verified the server's identity.

As shown later, these two classes of authentication are closely related, and can be handled by similar protocols.

From now on, we use authentication to refer to general identity authentication.

AUTHENTICATION PROTOCOL PARADIGMS

Authentication in distributed systems is invariably carried out with protocols. A *protocol* is a precisely defined sequence of *communication* and *computation* steps. A communication step transfers messages from one principal (the sender) to another (the receiver), while a computation step updates a principal's internal state. Two distinct states can be identified upon termination of the protocol, one signifying successful authentication and the other failure.

Although the goal of any authentication is to verify the claimed identity of a principal, specific success and failure states are highly protocol dependent. For example, the success of an authentication during the connection establishment phase of a communication protocol is usually indicated by the distribution of a fresh *session key* between two mutually authenticated peer processes. On the other hand, in a user login authentication, success usually results in the creation of a login process on behalf of the user.

We present protocols in the following format. A communication step whereby P sends a message M to Q is represented as $P \rightarrow Q : M$ whereas a computation step of P is written as $P : \ldots$ where "$\ldots$" is a specification of the computation step. For example, following is the typical login protocol between a host H and a user U (f denotes a *one-way* function, that is, given y it is computationally infeasible to find an x such that $f(x) = y$):

$$\begin{array}{lll} U \rightarrow H & : & U \\ H \rightarrow U & : & \text{“Please enter password”} \\ U \rightarrow H & : & p \\ H & : & \text{compute } y = f(p) \\ & : & \text{retrieve user record } (U, f(\textit{password}_U)) \text{ from user database} \\ & : & \text{if } y = f(\textit{password}_U) \text{ then accept; otherwise reject} \end{array}$$

We next examine the key ideas that underlie authentication protocol design by presenting several protocol paradigms.

Since authentication protocols directly use cryptosystems, their basic design principles also follow closely the type of cryptosystem used. Specifically, we identify two basic paradigms for authentication, one based on symmetric cryptosystems and the other on asymmetric cryptosystems.

Note that protocols presented in this section are intended to illustrate basic design principles only. A realistic protocol is necessarily a refinement of these basic protocols and addresses weaker environment assumptions, stronger postconditions, or both. Also, a realistic protocol may use both symmetric and asymmetric cryptosystems.

The protocols presented in the balance of this chapter have been slightly revised from the ones published in [16]. The revisions ensure that they follow a design principle for authentication protocols called the Principle of Full Information as expounded in [18]. According to the principle, a principal should, in an authentication exchange, include in each outgoing encrypted message all of the information it has gathered so far in the exchange. In particular, each message should contain the names of the authenticating principals. A conclusion of [18] is that to optimize an authentication protocol, a designer should focus on reducing the number of messages (or rounds) in the protocol, rather than simplifying encrypted messages.

Protocols Based upon Symmetric Cryptosystems

In a symmetric cryptosystem, knowing the shared key lets a principal encrypt and decrypt arbitrary messages. Without such knowledge, a principal cannot create the encrypted version of a message, or decrypt an encrypted message. Hence, authentication protocols can be designed according to the following principle called SYM:

> *If a principal can correctly encrypt a message using a key that the verifier believes is known only to a principal with the claimed identity (outside of the verifier), this act constitutes sufficient proof of identity.*

Thus SYM embodies the proof-by-knowledge principle for authentication, that is, a principal's knowledge is indirectly demonstrated through its ability to encrypt (see "Approaches to Authentication" sidebar). Using SYM, we immediately obtain the following basic protocol: (k denotes a symmetric key shared between P and Q)

$$\begin{array}{lll}
P & : & \text{create } m = \text{“I am } P\text{.”} \\
 & : & \text{compute } m' = \{m, Q\}_k \\
P \rightarrow Q & : & m, m' \\
Q & : & \text{verify } \{m, Q\}_k \stackrel{?}{=} m' \\
 & : & \text{if equal then accept; otherwise reject}
\end{array}$$

Clearly, the SYM design principle is sound only if the underlying cryptosystem is strong (one cannot create the encrypted version of a message without knowing the key) and the key is secret (it is shared only between the real principal and the verifier). Note that this protocol performs only one-way authentication; mutual authentication can be achieved by reversing the roles of P and Q.

One major weakness of the protocol is its vulnerability to replays. More precisely, an adversary could masquerade as P by recording the message m' and later replaying it to Q. As mentioned, replay attacks can be countered by using nonces or timestamps. We modify the protocol by adding a *challenge-and-response* step using nonces:

$$\begin{array}{lll}
P \rightarrow Q & : & \text{“I am } P\text{.”} \\
 & : & \text{generate nonce } n \\
Q \rightarrow P & : & n \\
P & : & \text{compute } m = \{P, Q, n\}_k \\
P \rightarrow Q & : & m \\
Q & : & \text{verify } \{P, Q, n\}_k \stackrel{?}{=} m \\
 & : & \text{if equal then accept; otherwise reject}
\end{array}$$

Replay is foiled by the freshness of n. Thus, even if an eavesdropper has monitored all previous authentication conversations between P and Q, it

Approaches to Authentication

All authentication procedures involve checking known information about a claimed identity against information supplied by the claimant during the identity verification procedure. Such checking can be based on the following three approaches [2]:

Proof by Knowledge. The claimant demonstrates knowledge of some information regarding the claimed identity that can only be known or produced by a principal with the claimed identity. For example, password knowledge is used in most login procedures. A proof by knowledge can be conducted by a direct demonstration, like typing in a password, or by an indirect demonstration, such as correctly computing replies to challenges by a verifier. Direct demonstration is not preferable from a security viewpoint, since a compromised verifier can record the submitted knowledge and later impersonate the claimant by presenting the recorded knowledge. Indirect demonstration can be designed to induce high confidence in the verifier, without leaving any clue to how the claimant's replies are computed. For example, Feige, Fiat, and Shamir proposed a *zero-knowledge* protocol for proof of identity [1]. This protocol allows a claimant C to prove to a verifier V that C knows how to compute replies to challenges posed by V without revealing what the replies are. These protocols are provably secure (under complexity assumptions). However, additional refinements are needed before they can be applied in practical systems.

Proof by Possession. The claimant produces an item that can only be possessed by a principal with the claimed identity, for example, an ID badge. For this to work, the item has to be unforgeable and safely guarded.

Proof by Property. The verifier directly measures certain claimant properties. For example, various biometric techniques can be used: fingerprint, retina print, and so on. The measured property has to be distinguishable, that is, unique among all possible principals.

Proof by knowledge and possession (and combinations thereof) can be applied to all types of authentication needs in a secure distributed system, while proof by property is generally limited to the authentication of human users by a host equipped with specialized measuring instruments.

References

1. U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *ACM Symposium on Theory of Computing*, pages 210–217, 1987.
2. K. Shankar. The total computer security problem. *Computer*, 10(6):50–73, June 1977.

still could not produce the correct n'. (This also points out the need for the cryptosystem to withstand a *known plaintext* attack. That is, the cryptosystem must be unbreakable given the knowledge of plaintext-ciphertext pairs.) The challenge-and-response step can be repeated any number of times until the desired level of confidence is reached by Q.

This protocol is impractical as a general large-scale solution because each principal must store in memory the secret key for every other principal it would ever want to authenticate. This presents major initialization (the predistribution of secret keys) and storage problems. Moreover, the compromise of one principal can potentially compromise the entire system. These problems can be significantly reduced by postulating a centralized *authentication server* A that shares a secret key k_{XA} with every principal X in the system [11]. The basic authentication protocol then becomes:

$P \rightarrow Q$	:	"I am P."
Q	:	generate nonce n
$Q \rightarrow P$	:	n
P	:	compute $x = \{P, Q, n\}_{k_{PA}}$
$P \rightarrow Q$	:	x
Q	:	compute $y = \{P, Q, x\}_{k_{QA}}$
$Q \rightarrow A$	:	y
A	:	recover P, Q, x from y by decrypting with k_{QA}
	:	recover P, Q, n from x by decrypting with k_{PA}
	:	compute $m = \{P, Q, n\}_{k_{QA}}$
$A \rightarrow Q$	:	m
Q	:	verify $\{P, Q, n\}_{k_{QA}} \overset{?}{=} m$
	:	if equal then accept; otherwise reject

Thus Q's verification step is preceded by a *key translation* step by A. The protocol correctness now also rests on A's trustworthiness—that A will properly decrypt using P's key and reencrypt using Q's key. The initialization and storage problems are greatly alleviated because each principal needs to keep only one key. The risk of compromise is mostly shifted to A, whose security can be guaranteed by various measures, such as encrypting stored keys using a master key and putting A in a physically secure room.

Protocols Based upon Asymmetric Cryptosystems

In an asymmetric cryptosystem, each principal P publishes his public key k_P and keeps secret his private key k_P^{-1}. Thus only P can generate $\{m\}_{k_P^{-1}}$ for any message m by signing it using k_P^{-1}. The signed message $\{m\}_{k_P^{-1}}$ can be verified by any principal with knowledge of k_P (assuming a commutative asymmetric cryptosystem). The ASYM design principle is:

If a principal can correctly sign a message using the private key of the claimed identity, this act constitutes a sufficient proof of identity.

This ASYM principle follows the proof-by-knowledge principle for authentication, in that a principal's knowledge is indirectly demonstrated through its signing capability. Using ASYM, we obtain a basic protocol as follows:

$$\begin{array}{lll}
P \rightarrow Q & : & \text{“I am } P\text{.”} \\
Q & : & \text{generate nonce } n \\
Q \rightarrow P & : & n \\
P & : & \text{compute } m = \{P, Q, n\}_{k_P^{-1}} \\
P \rightarrow Q & : & m \\
Q & : & \text{verify } (P, Q, n) \overset{?}{=} \{m\}_{k_P} \\
 & : & \text{if equal then accept; otherwise reject}
\end{array}$$

This protocol depends on the guarantee that $\{n\}_{k_P^{-1}}$ cannot be produced without the knowledge of k_P^{-1} and the correctness of k_P as published by P and kept by Q.

As in the protocols that use symmetric keys, the initialization and storage problems can be alleviated by postulating a centralized *certification authority* A that maintains a database of all published public keys. The protocol can then be modified as follows:

$$\begin{array}{lll}
P \rightarrow Q & : & \text{“I am } P\text{.”} \\
Q & : & \text{generate nonce } n \\
Q \rightarrow P & : & n \\
P & : & \text{compute } m = \{P, Q, n\}_{k_P^{-1}} \\
P \rightarrow Q & : & m \\
Q \rightarrow A & : & \text{“I need } P\text{'s public key.”} \\
A & : & \text{retrieve public key } k_P \text{ of } P \text{ from key database} \\
 & : & \text{create certificate } c = \{P, k_P\}_{k_A^{-1}} \\
A \rightarrow Q & : & P, c \\
Q & : & \text{recover } (P, k_P) \text{ from } c \text{ by decrypting with } k_A \\
 & : & \text{verify } (P, Q, n) \overset{?}{=} \{m\}_{k_P} \\
 & : & \text{if equal then accept; otherwise reject}
\end{array}$$

Thus c, called a *public-key certificate,* represents a certified statement by A that P's public key is k_P. Other information such as an expiration date and the classification of principal P (for mandatory access control) can also be included in the certificate (such information is omitted here). Each principal in the system need only keep a copy of the public key k_A of A.

In this protocol, A is an example of an *on-line* certification authority. It supports interactive queries and is actively involved in authentication exchanges.

A certification authority can also operate *off-line*, in which case, a public-key certificate is issued to a principal when it first registers. The certificate is kept by the principal and is forwarded during an authentication exchange, thus eliminating the need to make a separate query. Forgery is impossible, since a certificate is signed by the certification authority.

Notion of Trust

Correctness of both the symmetric and asymmetric protocols presented above requires more than the existence of secure communication channels between principals and the appropriate authentication servers and certification authorities. In fact, such correctness is critically dependent on the ability of the servers and authorities to faithfully follow the protocols. Each principal bases its judgment on its own observations (messages sent and received) and its trust of the server's judgment.

An authentication server in a symmetric protocol is trusted not to divulge the secret keys of principals and to apply the correct secret key as specified by the protocol. An on-line certification authority is trusted not to divulge its own private key and to have the correct public keys of principals. An off-line certification authority is trusted not to divulge its own private key and to properly verify the identity of a principal before issuing a public-key certificate for the principal.

A formal understanding of authentication would require both a formal specification of trust and a rigorous reasoning method wherein trust is a basic element. Presently, our formal understanding of trust in distributed systems is at best inadequate.

AUTHENTICATION PROTOCOL FAILURES

Despite the apparent simplicity of their basic design principles, realistic authentication protocols are notoriously difficult to design. Various published protocols have exhibited subtle security problems [3, 4, 11].

There are several reasons for such difficulty. First, most realistic cryptosystems satisfy algebraic identities additional to those in (C1) and (C2). These extra properties may generate undesirable effects when combined with protocol logic. Second, even assuming that the underlying cryptosystem is perfect, unexpected interaction among the protocol steps can lead to subtle logical flaws. Third, assumptions regarding the environment and the capabilities of an adversary are not explicitly specified, making it extremely difficult to determine when a protocol is applicable and what final states have been achieved.

We illustrate the difficulty by showing an authentication protocol proposed in [11] that contains a subtle weakness [4] (k_P and k_Q are symmetric keys shared between P and A, and Q and A, respectively, where A is an authentication server and k is a session key).

$$\begin{array}{llll}
(1) & P \rightarrow A & : & P, Q, n_P \\
(2) & A \rightarrow P & : & \{n_P, Q, k, \{k, P\}_{k_Q}\}_{k_P} \\
(3) & P \rightarrow Q & : & \{k, P\}_{k_Q} \\
(4) & Q \rightarrow P & : & \{n_Q\}_k \\
(5) & P \rightarrow Q & : & \{n_Q + 1\}_k
\end{array}$$

The message $\{k, P\}_{k_Q}$ in step (3) can only be decrypted and hence understood by Q. Step (4) reflects Q's knowledge of k, while step (5) assures Q of P's knowledge of k; hence the authentication handshake is based entirely on knowledge of k. The subtle weakness in the protocol arises from the fact that the message $\{k, P\}_{k_Q}$ sent in step (3) contains no information for Q to verify its freshness.[2] In fact, this is the first message sent to Q about P's intention to establish a secure connection. An adversary who has compromised an old session key k' can impersonate P by replaying the recorded message $\{k', P\}_{k_Q}$ in step (3) and subsequently executing the steps (4) and (5) using k'.

To avoid protocol failures, formal methods may be employed in the design and verification of authentication protocols. A formal design method should embody the basic design principles as illustrated in the previous section. Informal reasoning such as, "If you believe that only you and Bob know k, then you should believe any message you receive encrypted with k was originally sent by Bob," should be formalized by a verification method.

Early attempts at formal verification of security protocols mainly followed an algebraic approach [5]. Messages exchanged in a protocol were viewed as terms in an algebra. Various identities involving the encryption and decryption operators (for example, (C1) and (C2)) were taken to be term-rewriting rules. A protocol was *secure* if it was impossible to derive certain terms (for example, the term containing the key) from the terms obtainable by an adversary. The algebraic approach was limited, since it had been used mainly to deal with one aspect of security, namely secrecy. Recently, various logical approaches have been proposed to study authentication protocols [3]. Most of these logics adopt a modal basis, with *belief* and *knowledge* as central notions. The logical approaches appear to be more general than the algebraic ones, but they lack the rigorous foundations of more well-established logics like first-order logic and temporal logic. In particular, a satisfactory semantic model for these logical systems has not been developed. Much research is needed to obtain sound design methods and to formally understand authentication issues.

AN AUTHENTICATION FRAMEWORK

We have so far presented various basic concepts of authentication. In this section, we synthesize these concepts into an authentication framework that can be incorporated into the design of secure distributed systems. In particular, we identify and describe five aspects of secure distributed system design and the associated authentication needs. This section is not exhaustive in scope; other issues may have to be addressed in an actual distributed system security framework.

Host initializations. All process executions take place inside hosts. Some hosts (like workstations) also act as system entry points by allowing user logins. The overall security of a distributed system is highly dependent on the security of each of the hosts. However, in an open network environment, not all hosts can be physically protected. Thus resistance to compromise must be built into a host's software to ensure secure operation. This suggests the importance of host software integrity. In particular, for a host that employs remote initialization, loading it with the correct host software is essential to its proper functioning. In fact, one way to compromise a public host is to reboot the host with incorrect initialization information. Authentication can be used to implement secure bootstrapping.

User logins. User identity is established at login, and all subsequent user activities are attributed to this identity. All access control decisions and accounting functions are based on this identity. Correct user identification is thus crucial to the functioning of a secure system. Also, any host in an open environment is susceptible to compromise. Therefore a user should not engage in any activity with a host without first ascertaining the host's identity. A mutual user-host authentication can achieve the required guarantees.

Peer communications. Distributed systems can distribute a task over multiple hosts to achieve a higher throughput or more balanced utilizations than centralized systems. Correctness of such a distributed task depends on whether peer processes participating in the task can correctly identify each other. Authentication can be used here to identify friend or foe.

Client-server interactions. The client-server model provides an attractive paradigm for constructing distributed systems. Servers are willing to provide service only to authorized clients while clients are interested in dealing only with legitimate servers. Authentication can be used to establish a verified consumer-supplier relationship.

Inter-domain communications. Most distributed systems are not centrally owned or administered; for example, a campus-wide distributed system often interconnects individually administered departmental subsystems. Identifying principals across subsystems requires additional authentication mechanisms across domains.

Assumptions

In the kind of malicious environments postulated in our threats model, some basic assumptions about the system must be satisfied to achieve some level of security. We offer a set of assumptions below (for other possible assumptions, see [1, 10]). These assumptions are also depicted in Figure 2.

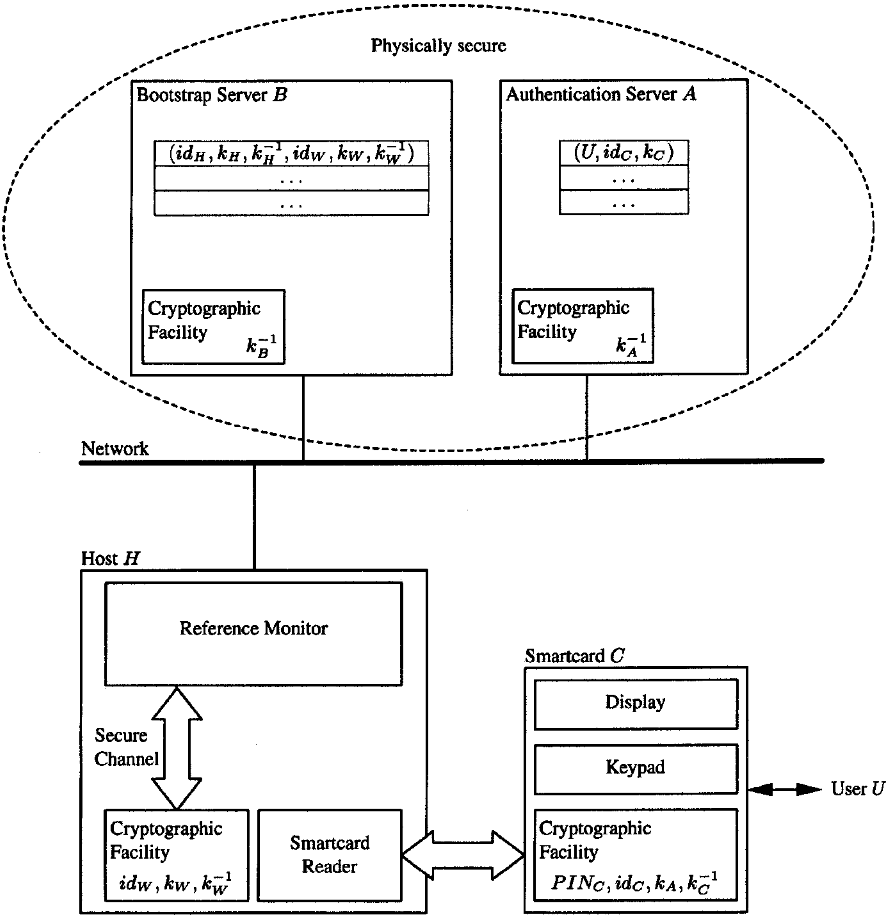

Physically secure
Bootstrap Server B
$(id_H, k_H, k_H^{-1}, id_W, k_W, k_W^{-1})$
...
...
Cryptographic Facility k_B^{-1}
Authentication Server A
(U, id_C, k_C)
...
...
Cryptographic Facility k_A^{-1}
Network
Host H
Reference Monitor
Secure Channel
Cryptographic Facility id_W, k_W, k_W^{-1}
Smartcard Reader
Smartcard C
Display
Keypad
Cryptographic Facility $PIN_C, id_C, k_A, k_C^{-1}$
User U

FIGURE 2 Authentication architecture.

- **Cryptographic Facility.** Each host hardware W has a unique built-in immutable identity id_W, and contains a tamper-proof cryptographic facility (CF) that encapsulates the public key k_W and the private key k_W^{-1} of W. That is, the keys are permanently sealed inside a CF and cannot be directly read from the outside, even by the host itself. The second function of a CF is to act as a black box for all cryptographic computation. A CF accepts commands and data from the host reference monitor and carries out any requested computation using both the supplied data and its internal information. A CF can communicate with the host reference monitor via a secure channel.

  Ideally, a CF is implemented in hardware either as an add-on card or directly on the motherboard. In this case, the tamper-proof property can be enforced by engineering design and tremendous computational advantages can be gained. Alternatively, a CF can be implemented in software. In this case, explicit trust assumptions (for example, the root file system is secure) will be needed.

- **Smartcard.** Our framework makes use of smartcards for user logins. The main function of a smartcard is to serve as an aid for a human user to carry out (mostly cryptographic) computation required by the user-host authentication protocol. A *smartcard* is a calculator-like device that has a display and a keypad, and contains a CF and a clock.

  Each legitimate user U is issued a smartcard C that has a unique built-in immutable identity id_C. Each smartcard C encapsulates in its CF its private key k_C^{-1}, the public key of the authentication server k_A, and a pin number PIN_C for its legitimate holder. (The pin number is chosen in a card-issuing procedure.) Each host that supports user logins using smartcards is equipped with a smartcard reader.

  The smartcards are assumed to be customizable. That is, the authority issuing a smartcard can initialize its contents with specific chosen values. In particular, the value of PIN_C is chosen by a user, while the value of k_A is fixed for a particular security domain.

- **Physical Security.** Certain assumptions regarding physical security are also needed for our framework. These assumptions are typical of most security frameworks. In fact, it can be informally argued that some minimal physical security is always required for "bootstrapping" security. In other words, a security framework should be thought of as an "amplifier" for security.

  The bootstrap and authentication servers in our framework are assumed to be secure. Typically, this is achieved by running these servers in a dedicated fashion on physically secure machines. No regular user accounts are allowed on these machines and they are locked in physically secure rooms.

The bootstrap server B is used in secure bootstrapping. It maintains a database of host information. In our framework, we make a distinction between *host* and *host hardware.* A host hardware refers to a bare machine, for example, a Sun SPARC 10 workstation with a particular serial number. A host refers to a specific instance of an operating system on some host hardware. A host typically has a high-level (for example, DNS) host name and an IP address.

The host database contains, for each host H, a record of the form

$$(id_H, k_H, k_H^{-1}, id_W, k_W, k_W^{-1})$$

specifying the unique host hardware W that can be initialized to run H. For added security, all records in the database can be encrypted under a secret master key.

The authentication server A maintains a database on principals. More precisely, for each user U, A keeps a record (U, id_C, k_C), binding U to its smartcard C. Also, for each "end" server S, A keeps a record of its public key k_S.

Each of the above assumptions is achievable with current technology. In particular, the technology of a battery-powered credit-card-sized smartcard with a built-in LCD display and keypad that can perform specialized computations has steadily progressed in recent years. Also, some vendors are starting to include specialized cryptographic facilities and smartcard readers for hosts as options. The use of a smartcard or other form of computation aid is essential to realizing mutual authentication between a host and a user. Unaided human users simply cannot carry out the intensive computations required by an authentication protocol.

To simplify our presentation, the bootstrap server and the authentication server are assumed to be centralized. Decentralized servers can be supported by adding authentication between them (see the section on inter-domain authentication). Such authentication can be carried out in a hierarchical manner as suggested in the protocol standard CCITT X.509 [19].

Protocol Overview

In the following subsections, we present protocol solutions to address the authentication needs outlined above. Specifically, we present concrete protocols, namely, a *secure bootstrap* protocol, a *user-host authentication* protocol and a *peer-peer authentication* protocol, to address respectively the authentication needs of host initializations, user logins, and peer communications. Client-server authentication is a special case of peer-peer authentication, and can be achieved with a similar protocol.

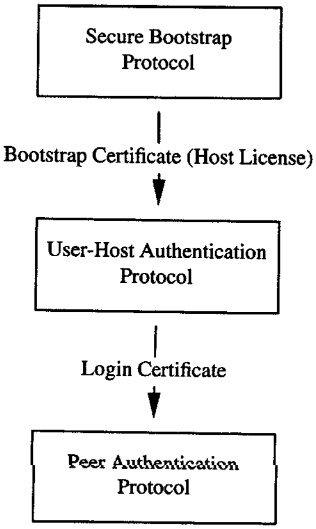

Secure Bootstrap Protocol
Bootstrap Certificate (Host License)
User-Host Authentication Protocol
Login Certificate
Peer Authentication Protocol

FIGURE 3 Relationship between protocols.

The secure bootstrap protocol is used to initialize a host into a "safe" and well-defined initial state prior to resuming normal operation. In particular, a correctly loaded reference monitor is ready to assume control of the host in this state. The user-host authentication protocol is responsible for user logins; it allows mutual authentication between a user and a host. The peer authentication protocol mutually authenticates two peer processes.

These protocols are inter-related to one another in that the information acquired in one protocol is used in another protocol (see Figure 3). For example, a *bootstrap certificate* or *host license* is generated upon successful termination of the secure bootstrap protocol. This host license is in turn used in the user-host authentication protocol to generate a *login certificate.* Similarly, the login certificate can be used in the authentication exchange of the peer authentication protocol.

Our protocols should not be considered definitive or optimal. They are presented in this chapter to illustrate possible solution approaches and, together, they demonstrate a coherent and consistent solution for authentication in distributed systems. In the last subsection, we briefly discuss the issues of inter-domain authentication.

Secure Bootstrapping

The secure bootstrap protocol is initiated when a host hardware attempts a remote initialization. This could occur after a voluntary shutdown, a system crash, or a malicious attack by an adversary attempting to subvert the host.

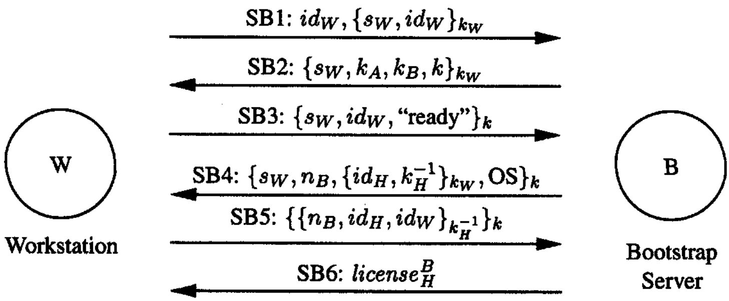

SB1: $id_W, \{s_W, id_W\}_{k_W}$
SB2: $\{s_W, k_A, k_B, k\}_{k_W}$
SB3: $\{s_W, id_W, \text{"ready"}\}_k$
W
SB4: $\{s_W, n_B, \{id_H, k_H^{-1}\}_{k_W}, \text{OS}\}_k$
B
SB5: $\{\{n_B, id_H, id_W\}_{k_H^{-1}}\}_k$
Workstation
SB6: $license_H^B$
Bootstrap Server

FIGURE 4 Secure bootstrap protocol.

The secure bootstrap protocol specification is shown in Figure 4. A step by step specification including some computation steps is given below. OS denotes the operating system to be bootstrapped.

	W	:	generate new secret s_W
(SB1)	$W \rightarrow$ all	:	$id_W, \{s_W, id_W\}_{k_W}$
	B	:	retrieve record $(id_H, k_H, k_H^{-1}, id_W, k_W, k_W^{-1})$ for W from database
		:	generate new session key k
(SB2)	$B \rightarrow W$	:	$\{s_W, k_A, k_B, k\}_{k_W}$
	W	:	if s_W present, proceed; otherwise abort
(SB3)	$W \rightarrow B$	:	$\{s_W, id_W, \text{"ready"}\}_k$
	B	:	generate nonce n_B
(SB4)	$B \rightarrow W$	:	$\{s_W, n_B, \{id_H, k_H^{-1}\}_{k_W}, \text{OS}\}_k$
(SB5)	$W \rightarrow B$	:	$\{\{n_B, id_H, id_W\}_{k_H^{-1}}\}_k$
(SB6)	$B \rightarrow W$	:	$license_H^B$

The basic idea of the protocol is as follows: Upon resetting, W generates a new secret s_W for use as a challenge. A secret is like a nonce but with the additional property that it is not predictable. In step (SB1), W announces its intention to reboot by broadcasting a boot request. We assume that W and the bootstrap server B are on the same broadcast network, thus allowing B to receive the boot request. The boot request is encrypted using k_W. Therefore, only B, which has knowledge of k_W^{-1}, can recover the secret s_W. On receiving the boot request, B retrieves the record for id_W, and uses k_W^{-1} in the record to recover s_W from the boot request. B then generates a fresh key k to be used for loading OS. In step (SB2), the new key k, together with the public keys of B and authentication server A, are sent to W. W ascertains that $\{s_W, k_A, k_B, k\}_{k_W}$ came from B by checking the presence of s_W, since only B could have composed the message. The nonce property of s_W demonstrates that the message is not a replay. Thus, k_A, k_B, and k in the message can be safely taken to be respectively the public keys of A and B, and the session key to be used for loading OS. At this point, W has authenticated B. It proceeds by sending the "ready" message in step (SB3).

When the "ready" message is received, B is certain that the original boot request actually came from W, because only W could have decrypted $\{s_W, k_A, k_B, k\}_{k_W}$ to retrieve k. The boot request is timely because the session key k also serves as a nonce. At this point, B and W have mutually authenticated each other.

Step (SB4) is the actual loading of OS and the transferring of host H's private key k_H^{-1}. OS includes its checksum, which should be recomputed by W to detect any tampering in transit. W acknowledges the receipt of k_H^{-1} and OS by returning the nonce n_B, and id_H and id_W signed with k_H^{-1} in step (SB5). B then verifies that the correct n_B and IDs are returned. In step (SB6), a host license

$$license_H^B = \{id_H, id_W, k_H, T_h, L_h\}_{k_B^{-1}}$$

signed by B affirming the binding of host id_H with public key k_H and host hardware id_W is sent to W. The fields T_h and L_h within the license denote the creation time and expiration date of the license, respectively.

After receiving the license, W officially "becomes" H, which retains the license as proof of successful bootstrapping and of its own identity. Observe that if secrecy is not required, OS can be transferred unencrypted. However, the checksum of OS must be sent in encrypted form.

Discussion

The design of the secure bootstrap protocol violates one common principle for using asymmetric cryptosystems, namely, the private key of a principal is not shared so that trust requirements are reduced. In our design, the private key k_W^{-1} of W is shared between W and B, and it is used essentially as a shared secret key (as in a symmetric cryptosystem) in the initial authentication steps ((SB1) and (SB2)). The rationale behind this is to avoid the need to customize the cryptographic facilities of hosts (for example, preloading each host's CF with k_B).

Another approach is to have a host's CF pre-certify (that is, sign in the form of a certificate) the public keys it will need. For example, W's CF can pre-certify both A and B's public keys by creating two certificates, one each for A and B, and storing them in some on-line certificate depository D. On receiving a boot request from W, D sends these certificates to W, which recognizes its own signature and recovers the public keys it needs to continue bootstrapping.

User-Host Authentication

User-host authentication occurs when a human user U walks up to a host H and attempts to log in. Our authentication protocol requires a smartcard C. A successful authentication guarantees host H that U is the legitimate holder of

C and guarantees user U that H is a "safe" host to use. That is, host H holds a valid license (obtained through secure bootstrapping) and possesses knowledge of the private key k_H^{-1}.

In most systems, the end result of a successful user authentication is the creation of a login process by the host's reference monitor on the user's behalf. The login process is a proxy for the user, and all requests generated by this process are taken as if they are directly made by the user. However, a remote host/server has no way of confirming such proxy status, except to trust the authentication capability and integrity of the local host. Such trust is unacceptable in a potentially malicious environment because a compromised host can simply claim the existence of user login processes to obtain unauthorized services.

This trust requirement can be alleviated if a user explicitly *delegates* its authority to the login host [1, 10]. The delegation is carried out by having the user's smartcard sign a *login certificate* to the login host upon the successful termination of a user-host authentication protocol. The login certificate asserts the host's proxy status with respect to the user, and can be presented by the host in future authentication exchanges with others.

Because of the possibility of forgery, the possession of a login certificate should not be taken as sufficient proof of delegation. The host must also demonstrate knowledge of a private delegation key k_d^{-1} whose public counterpart k_d is named in the certificate. In addition, to reduce the potential impact of a host compromise, an expiration timestamp is included so that the login certificate is given only a finite lifetime.

We present such a user-host authentication protocol in Figure 5. A specification with computation steps is given below. We assume that the host

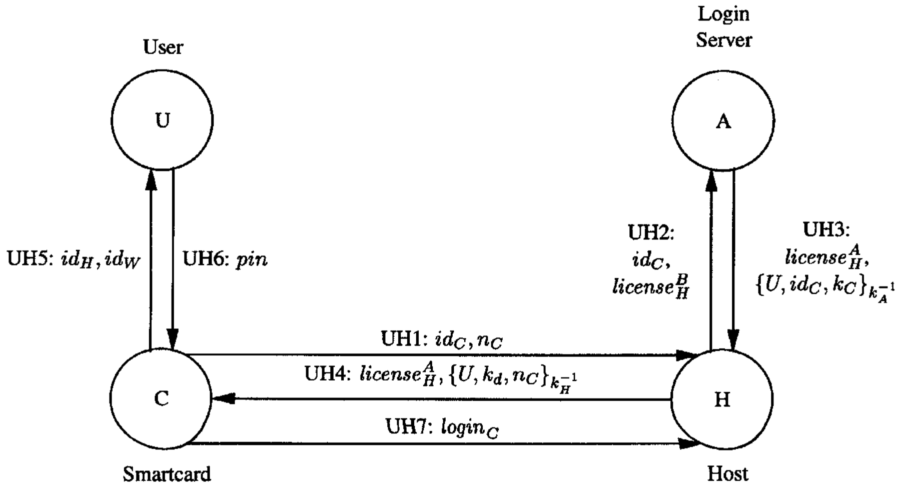

Login Server
User
U
A
UH5: id_H, id_W
UH6: pin
UH2: id_C, $license_H^B$
UH3: $license_H^A$, $\{U, id_C, k_C\}_{k_A^{-1}}$
UH1: id_C, n_C
UH4: $license_H^A, \{U, k_d, n_C\}_{k_H^{-1}}$
UH7: $login_C$
C
H
Smartcard
Host

FIGURE 5 User-Host authentication protocol.

holds a valid license $license_H^B$ as would be the case if the host has executed the secure bootstrap protocol.

	C	:	generate nonce n_C
(UH1)	$C \rightarrow H$	:	id_C, n_C
(UH2)	$H \rightarrow A$	:	$id_C, license_H^B$
	A	:	check host license lifetime; if expired, abort
(UH3)	$A \rightarrow H$	:	$license_H^A, \{U, id_C, k_C\}_{k_A^{-1}}$
	H	:	generate new delegation key pair (k_d, k_d^{-1})
(UH4)	$H \rightarrow C$	:	$license_H^A, \{U, k_d, n_C\}_{k_H^{-1}}$
	C	:	check license lifetime; if expired, abort
(UH5)	$C \rightarrow U$	:	id_H, id_W
	U	:	verify if id_H/id_W is the host desired; if not, abort
(UH6)	$U \rightarrow C$	:	pin
	C	:	verify $pin \stackrel{?}{=} PIN_C$; if not equal, abort
(UH7)	$C \rightarrow H$	:	$login_C$

The protocol proceeds as follows: A user inserts his/her smartcard into the host's card reader. This activates the card and it generates a nonce n_C. In step (UH1), the card's identity id_C together with n_C are sent through the card reader to the host. In step (UH2), H requests user information associated with id_C from the authentication server A. Since the license held by H was signed by B and hence is not decipherable by C, a key translation is requested by H in the same step.

Upon receiving the request from H, A first checks that the host license submitted has not expired. Then it retrieves the user record for id_C and forwards that along with the translated license $license_H^A = \{id_H, id_W, k_H, T_l, L_l\}_{k_A^{-1}}$ to H in step (UH3). (Note that this license can be cached by H and need not be requested for every user authentication.)

H now knows both the legitimate holder U of the smartcard C and the public key k_C associated with C. Knowledge of U can be used to enforce local discretionary control to provide service (or not), while k_C is needed to verify the authenticity of C. H proceeds to generate a new delegation key pair (k_d, k_d^{-1}). H keeps k_d^{-1} private, to be used in the future for demonstrating its delegation from U.

In step (UH4), H returns the nonce n_C with the public delegation key k_d, and a copy of its translated license to C. C retrieves (id_H, id_W), the identity of H, from the translated license by decrypting it with k_A. A check is made to ensure that the license has not expired. Then in step (UH5), the identity (id_H, id_W) is displayed on the card's own screen. In step (UH6), if the user decides to proceed, he/she enters on the card's keypad his/her pin number (pin) assigned when the card was issued. The pin number entered is compared with the one stored in the card, PIN_C. If they are equal, C signs a login certificate

$$login_C = \{U, id_H, k_d, T_c, L_c\}_{k_C^{-1}}$$

binding the user U with the host id_H and the public delegation key k_d; this is sent to H in step (UH7), completing the delegation. The fields T_c and L_c within the login certificate denotes the time of creation and expiration date of the login certificate, respectively. Host H (and others) can verify the validity of the login certificate using k_C.

When user U logs out, the host erases its copy of the private delegation key k_d^{-1} to void the delegation from U. If H is compromised after the delegation, the validity of the login certificate is limited by its lifetime, L_c.

Discussion

When smartcard C is issued, its CF is loaded with the public key of a particular server. For C to verify a host license, the license must be signed with a private key whose public counterpart is known to C. Thus, each card must be mapped to a particular authentication server A. Typically, a card is mapped to the authentication server associated with the authority that issues the card. If a user and a host belong to different domains (see the section on client-server authentication), multiple key translations may be needed before the license of the host can be presented to the user.

To reduce the smartcard's complexity, various implementation techniques can be used to eliminate the need for a clock on the card. Also, the keypad of the smartcard can be a simple one with just a few keys for making changes. Eliminating the keypad altogether requires more ingenuity, but can be done [1].

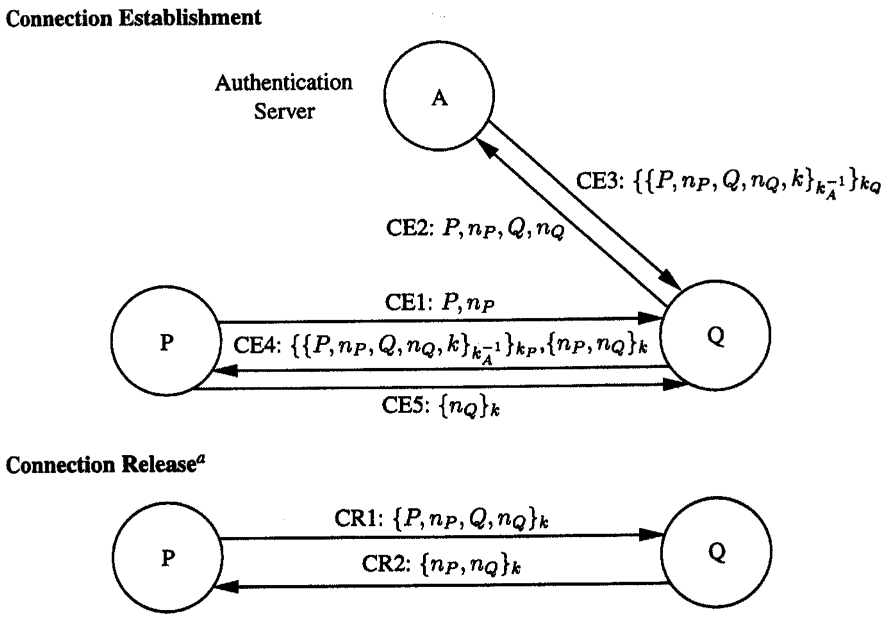
The display on a smartcard is crucial to many of its functionalities, and hence should not be eliminated. Indeed, the cost of an LCD display is insignificant compared to the extra trust required if it were eliminated.

Peer-Peer Authentication

The primary goal of peer authentication is to establish the identities of two peer principals. Most peer authentication protocols, however, also accomplish a secondary goal, namely, the negotiation of cryptographic parameters (for example, a new session key) for future communication between the peers. These cryptographic parameters are collectively referred to as a *security association.*

In connection-oriented communication schemes, peer authentication and the associated cryptographic parameters negotiation are performed in the connection establishment phase. In connectionless communication schemes, both authentication and cryptographic parameters negotiation can be performed the first time a principal is contacted.

The peer authentication protocol in our framework is shown in Figure 6. It actually consists of two separate protocols, one for connection establishment



[a] Either P or Q can initiate connection release. P is shown here as the initiator for illustration purpose.

FIGURE 6 Peer-Peer authentication protocol.

and one for connection release.[3] This protocol was first introduced in [16]. An implementation of the protocol to provide a secure socket service was reported in [15]. Its design principles and correctness proof were presented in [17].

The protocol assumes that the public key of each principal is known by all other principals. For example, Q knows k_P and k_A, the public keys of P and A, respectively. If P and Q are processes started from login shells, their public keys are the public delegation keys in their login certificates.

	P	:	generate nonce n_P
(CE1)	$P \rightarrow Q$	:	P, n_P
	Q	:	generate nonce n_Q
(CE2)	$Q \rightarrow A$	:	P, n_P, Q, n_Q
	A	:	generate new session key k
(CE3)	$A \rightarrow Q$	:	$\{\{P, n_P, Q, n_Q, k\}_{k_A^{-1}}\}_{k_Q}$
(CE4)	$Q \rightarrow P$	:	$\{\{P, n_P, Q, n_Q, k\}_{k_A^{-1}}\}_{k_P}, \{n_P, n_Q\}_k$
		:	create security association to Q with k as session key
(CE5)	$P \rightarrow Q$	:	$\{n_Q\}_k$
		:	create security association to P with k as session key

Consider the connection establishment protocol. Before P initiates a connection establishment request, it generates a nonce n_P for use as a challenge. In step (CE1), P initiates the authentication exchange by informing Q of its identity and nonce. Upon receiving P's connection establishment request, Q generates its own nonce n_Q. In step (CE2), Q forwards both the identity of P and P's nonce together with its own identity and nonce to A.

A serves the authentication request from Q by generating a new session key k to be used for future communication between P and Q. In step (CE3), A sends k in a signed and encrypted message to Q. After verifying that the nonces n_P and n_Q are returned, Q recovers k and forwards in step (CE4) the signed component to P, together with an *authenticator* $\{n_P, n_Q\}_k$. The authenticator allows P to infer that the correspondent principal must be Q, as only Q (aside from A) knows k. At this point, P has authenticated Q and is willing to accept k as the session key for its security association to Q. In step (CE5), P returns an acknowledgment $\{n_Q\}_k$ to Q. This authenticates P to Q, which proceeds to install a security association to P with k as the session key.

For key distribution only, the authenticator in step (CE4) and the subsequent acknowledgment in step (CE5) are not necessary. They are included for *key handshake,* that is, to assure each other that the correct session key has been properly established by both principals.[4]

Consider the connection release protocol with P as the initiator. (Either P or Q can initiate connection release.)

(CR1)	$P \rightarrow Q$	:	$\{P, n_P, Q, n_Q\}_k$
	Q	:	verify presence of same nonces as used in connection establishment
		:	destroy security association with P
(CR2)	$Q \rightarrow P$	:	$\{n_P, n_Q\}_k$
	P	:	destroy security association with Q

In step (CR1), P sends a request for termination in the form $\{P, n_P, Q, n_Q\}_k$ to Q. The nonces n_P and n_Q are the same ones used in their connection establishment, and are stored as part of the security association. On receiving the termination request, Q checks that the nonces contained therein match those in its own security association. If so, Q acknowledges the request by returning $\{n_P, n_Q\}_k$ in step (CR2) and destroys the security association. P, on receiving the termination acknowledgment, destroys the corresponding security association, thus completing the connection release exchange. The scenario when Q initiates the termination request is symmetric.

Discussion

The connection establishment protocol was actually obtained by "composing" two subprotocols, one for key distribution and the other for mutual authentication. Our design of these two subprotocols and their composition are described in [17].

The connection establishment protocol is interesting in another regard: it can be viewed as a secure extension of the three-way handshake used in TCP connection establishment. Specifically, steps (CE1), (CE4), and (CE5) correspond to the three-way handshake. In (CE1), *P* communicates its sequence number (nonce n_P) to *Q*. In (CE4), *Q* acknowledges *P*'s sequence number as well as forwarding its own (nonce n_Q). Finally, *P* acknowledges *Q*'s sequence number in (CE5). The encryption required for (CE1), (CE4), and (CE5) together with the extra messages to *A* can be considered the cost of adding security to a three-way handshake.

The connection establishment protocol uses a trusted server *A* in its authentication exchange. This is not strictly necessary in an asymmetric encryption-based protocol (see the section on SPX). Whether or not an on-line trusted server should be used is a controversial topic. We believe that judicious use of on-line trusted servers can enhance security by providing on-line supervisory functions (for example, management, audit, and revocation) that cannot be achieved off-line. The key is achieving a balance between the desire for on-line functionalities and the degree of security risks one is willing to accept.

In our protocol, a trusted server *A* provides the following functionalities.

1. *A* provides a source of high-quality unbiased session keys for use between authenticating principals. This is especially important in an environment where the authenticating principals do not have a reliable local source of randomness. Moreover, it is generally agreed that an on-line random number service is essential to a distributed systems security infrastructure [8].
2. *A* provides an on-line audit service for tracking authentication exchanges. *P*, *Q*, and *A* can periodically reconcile their authentication records to reveal potential attacks.
3. *A* facilitates on-line management of principals. For example, *A* can be used to track where a principal is currently logged on.
4. *A* provides a simple revocation mechanism. It invalidates expired certificates and aborts authentications involving principals whose privileges have been revoked.

Client-Server Authentication

Since both clients and servers are implemented as processes, the basic protocol for peer-peer authentication can be applied here as well. However, several issues peculiar to client-server interactions need to be addressed.

In a general-purpose distributed system environment, new services (hence servers) are made available dynamically. Thus, instead of informing clients of every service available, most implementations use a *service broker* to keep track of and direct clients to appropriate service providers. A client first contacts the service broker by using a *purchase protocol* that performs the necessary mutual authentication prior to the granting of a *ticket*. The client later uses the ticket to redeem services from the actual server using a *redemption protocol*.

Authentication performed by the purchase protocol proceeds in the same way as the protocol for peer-to-peer authentication, while in the redemption protocol authentication is based upon possession of a ticket and knowledge of some information recorded in the ticket. Such a ticket contains the names of the client and the server, a key, and a timestamp to indicate lifetime (similar to a login certificate). A ticket can be used only between the specified client and server. A prime example of this approach is the Kerberos authentication system, which we discuss in a following section.

Another special issue of client-server authentication is *proxy authentication* [7]. To satisfy a client's request, a server often needs to access other servers on behalf of the client. For example, a database server, upon accepting a query from a client, may need to access the file server to retrieve certain information on the client's behalf. A straightforward solution would require the file server to directly authenticate the client. However, this may not be feasible. In a long chain of service requests, the client may not be aware of a request made by a server in the chain, and hence may not be in a position to perform the required authentication. An alternative is to extend the concept of delegation [7] previously used in user-host authentication. Specifically, a client can forward a signed *delegation certificate* affirming the delegation of its rights to a server along with its service request. The server is allowed to delegate to another server by signing its own delegation certificate as well as relaying the client's certificate. In general, for a service request involving a sequence of servers, delegation can be propagated to the final server through intermediate servers, forming a *delegation chain*.

Various refinements are possible to extend the delegation scheme described. For example, restricted delegation can be carried out by explicitly specifying a set of rights and/or objects in a delegation certificate.

Inter-Domain Authentication

Up to now, we have assumed a centralized certification authority trusted by all principals. However, a realistic distributed system is often composed of subsystems independently administered by different authorities. We use the term *domain* to refer to such a subsystem. Each domain D maintains its own certification authority A_D that has jurisdiction over all principals within the domain. *Intra-domain* authentication refers to an authentication exchange between two principals belonging to the same domain, whereas *inter-domain* authentication refers to an authentication exchange that involves two principals belonging to different domains.

Using the previously described protocols, A_D is sufficient for all intra-domain authentications for each domain D. However, a certification authority has no way of verifying a request from a remote principal, even if the request is certified by a remote certification authority. Hence, additional mechanisms are required for inter-domain authentication.

To allow inter-domain authentication, two issues need to be addressed: *naming* and *trust*. Naming is concerned with ensuring that principals are uniquely identifiable across domains, so that each authentication request can be attributed to a unique principal. A global naming system spanning all domains can be used to provide globally unique names to all principals. A good example of this is the Domain Name System used in the Internet.

Trust refers to the willingness of a local certification authority to accept a certification made by a remote authority regarding a remote principal. Such trust relationships must be explicitly established between domains, which can be achieved by:

- sharing an inter-domain key between certification authorities that are willing to trust each other,
- installing the public keys of all trusted remote authorities in a local certification authority's database, and
- introducing an inter-domain certification authority for authenticating domain-level authorities.

A hierarchical organization corresponding to that of the naming system can generally be imposed on the certification authorities. In this case, an authentication exchange between two principals P and Q involves multiple certification authorities on a path in the hierarchical organization between P and Q [6]. The path is referred to as a *certification path.*

CASE STUDIES

We study two authentication services: Kerberos and SPX. Both primarily address client-server authentication needs. Their services are generally available to an application program through a programming interface. While Kerberos uses only a symmetric cryptosystem, SPX uses an asymmetric cryptosystem as well.

Kerberos

Kerberos is an authentication system designed for MIT's Project Athena [12, 13]. The goal of Project Athena is to create an educational computing environment based on high-performance workstations, high-speed networking, and servers of various types. Researchers envisioned a large-scale (10,000 workstations to 1,000 servers) open network computing environment in which individual workstations can be privately owned and operated. Therefore, a workstation cannot be trusted to identify its users correctly to network services. Kerberos is not a complete authentication framework required for secure distributed computing in general; it only addresses issues of client-server interactions.

We limit our discussion to the Kerberos authentication protocols and omit various administrative issues.

Kerberos's design is based on the use of a symmetric cryptosystem together with trusted third-party authentication servers. It is a refinement of ideas presented in [11]. The basic components include authentication servers (*Kerberos servers*) and *ticket-granting servers* (TGSs). A database is maintained that contains information on each principal. It stores a copy of each principal's key that is shared with Kerberos. For a user principal U, its shared key k_U is computed from its password $password_U$; specifically $k_U = f(password_U)$ for some one-way function f. The database is read by Kerberos servers and TGSs in the course of authentication.

Kerberos uses two main protocols. The *credential initialization protocol* authenticates user logins and installs initial tickets at the login host. A client uses the client-server authentication protocol to request services from a server.

The credential initialization protocol uses Kerberos servers. Let U be a user who attempts to log into a host H. The protocol is specified in Figure 7.[5]

(1)	$U \rightarrow H$	:	U
(2)	$H \rightarrow$ Kerberos	:	U, TGS
(3)	Kerberos	:	retrieve k_U and k_{TGS} from database
		:	generate new session key k
		:	create ticket-granting ticket $tick_{\mathrm{TGS}} = \{U, \mathrm{TGS}, k, T, L\}_{k_{\mathrm{TGS}}}$
(4)	Kerberos $\rightarrow H$	:	$\{\mathrm{TGS}, k, T, L, tick_{\mathrm{TGS}}\}_{k_U}$
(5)	$H \rightarrow U$	:	"Password?"
(6)	$U \rightarrow H$	:	*passwd*
(7)	H	:	compute $p = f(passwd)$
		:	recover k, $tick_{\mathrm{TGS}}$ by decrypting $\{\mathrm{TGS}, k, T, L, tick_{\mathrm{TGS}}\}_{k_U}$ with p
		:	if decryption fails, abort login; otherwise retain $tick_{\mathrm{TGS}}$ and k
		:	erase *passwd* from memory

FIGURE 7 Kerberos credential initialization protocol.

In step (1), user U initiates login by entering his/her user name. In step (2), the login host H forwards the login request to a Kerberos server. In steps (3) and (4), the Kerberos server retrieves the user record of U and returns a *ticket-granting* ticket $tick_{\mathrm{TGS}} = \{U, \mathrm{TGS}, k, T, L\}_{k_{\mathrm{TGS}}}$ to H, where T is a timestamp and L is the ticket's lifetime. In steps (5) and (6), U enters his/her password in response to H's prompt. In step (7), If *passwd* is not the valid password of U, p would not be identical to k_U, and decryption in the last step would fail.[6] Upon successful authentication, the host obtains a new session key k and a copy of $tick_{\mathrm{TGS}}$. The ticket-granting ticket is used to request server tickets from a TGS. Note that $tick_{\mathrm{TGS}}$ is encrypted with k_{TGS}, the shared key between TGS and Kerberos.

Because a ticket is susceptible to interception or copying, it does not by itself constitute sufficient proof of identity. Therefore, a principal presenting a ticket must also demonstrate knowledge of the session key k named in the ticket. An *authenticator* (to be described) provides the demonstration. Figure 8 shows the protocol for a client C to request network service from a server S. T_1 and T_2 are timestamps.

In step (1), client C presents its ticket-granting ticket $tick_{\mathrm{TGS}}$ to TGS to request a ticket for server S.[7] C's knowledge of k is demonstrated using the authenticator $\{C, T_1\}_k$. In step (2), TGS decrypts $tick_{\mathrm{TGS}}$, recovers k, and uses it to verify the authenticator. If both step (2) decryptions are successful and T_1 is timely, TGS creates a ticket $tick_S$ for server S and returns it to C. Holding $tick_S$, C repeats the authentication sequence with S. Thus, in step (5), C presents S with $tick_S$ and a new authenticator. In step (6), S performs verifications similar to those performed by TGS in step (2). Finally, step (7) assures C of the server's

(1)	$C \rightarrow \mathrm{TGS}$	:	S, $tick_{\mathrm{TGS}}, \{C, T_1\}_k$
(2)	TGS	:	recover k from $tick_{\mathrm{TGS}}$ by decrypting with k_{TGS}
		:	recover T_1 from $\{C, T_1\}_k$ by decrypting with k
		:	check timeliness of T_1 with respect to local clock
		:	generate new session key k'
		:	create server ticket $tick_S = \{C, S, k', T', L'\}_{k_S}$
(3)	$\mathrm{TGS} \rightarrow C$	:	$\{S, k', T', L', tick_S\}_k$
(4)	C	:	recover k', $tick_S$ by decrypting with k
(5)	$C \rightarrow S$	:	$tick_S, \{C, T_2\}_{k'}$
(6)	S	:	recover k' from $tick_S$ by decrypting with k_S
		:	recover T_2 from $\{C, T_2\}_{k'}$ by decrypting with k'
		:	check timeliness of T_2 with respect to local clock
(7)	$S \rightarrow C$	:	$\{T_2 + 1\}_{k'}$

FIGURE 8 Kerberos client-server authentication protocol.

identity. Note that this protocol requires "loosely synchronized" local clocks for the verification of timestamps.

Kerberos can also be used for authentication across administrative or organizational domains. Each domain is called a *realm*. Each user belongs to a realm identified by a field in the user's ID. Services registered in a realm will accept only tickets issued by an authentication server for that realm.

To support cross-realm authentication, an *inter-realm key* is shared between two realms. The TGS of one realm can be registered as a principal in another realm by using the shared inter-realm key. A user can thus obtain a ticket-granting ticket for contacting a remote TGS from its local TGS. When the ticket-granting ticket is presented to the remote TGS, it can be decrypted by the remote TGS, which uses the appropriate inter-realm key to ascertain that it was issued by the user's local TGS. In general, an *authentication path* spanning multiple intermediate realms is possible.

Kerberos is an evolving system on its fifth version (V5). Various limitations of previous versions of Kerberos were discussed in [2, 9], some of which have been remedied.

SPX

SPX is another authentication service intended for open network environments [14]. It is a major component of the Digital Distributed System Security Architecture [6] and its functionalities resemble those of Kerberos. SPX has a credential initialization and a client-server authentication protocol. In addition, it has an *enrollment protocol* that registers new principals. In this subsection, we focus only on the first two protocols and omit the last, along with most other administrative issues.

SPX has a Login Enrollment Agent Facility (LEAF) and a Certificate Distribution Center (CDC) that corresponds to Kerberos servers and TGSs. LEAF, similar to a Kerberos server, is used in the credential initialization protocol. CDC is an on-line depository of public-key certificates (for principals and certification authorities) and the encrypted private keys of principals. Note that CDC need not be trustworthy as everything stored in it is encrypted and can be verified independently by principals.

SPX also contains hierarchically organized certification authorities (CAs) which operate off-line and are selectively trusted by principals. Their function is to issue public-key certificates (binding names and public keys of principals). Global trust is not needed in SPX. Each CA typically has jurisdiction over just one subset of all principals, while each principal P trusts only a subset of all CAs, referred to as the *trusted authorities of P*. System scalability is greatly enhanced by the absence of global trust and on-line trusted components.

(1)	$U \rightarrow H$	:	$U, passwd$
(2)	$H \rightarrow$ LEAF	:	$U, \{T, n, h_1(passwd)\}_{k_{\mathrm{LEAF}}}$
(3)	LEAF $\rightarrow$ CDC	:	U
(4)	CDC $\rightarrow$ LEAF	:	$\{\{k_U^{-1}\}_{h_2(password_U)}, h_1(password_U)\}_k, \{k\}_{k_{\mathrm{LEAF}}}$
(5)	LEAF	:	recover k by decrypting with k_{LEAF}^{-1}
		:	recover $\{k_U^{-1}\}_{h_2(password_U)}$ and $h_1(password_U)$ by decrypting with k
		:	verify $h_1(passwd) \overset{?}{=} h_1(password_U)$
		:	if not equal, abort
(6)	LEAF $\rightarrow H$	:	$\{\{k_U^{-1}\}_{h_2(password_U)}\}_n$
(7)	H	:	recover k_U^{-1} by decrypting first with n and then with $h_2(passwd)$
		:	generate (RSA) delegation key pair (k_d, k_d^{-1})
		:	create ticket $tick_U = \{L, U, k_d\}_{k_U^{-1}}$
(8)	$H \rightarrow$ CDC	:	U
(9)	CDC $\rightarrow H$	:	$\{A, k_A\}_{k_U^{-1}}$

FIGURE 9 SPX credential initialization protocol.

The credential initialization protocol is performed when a user logs in (see Figure 9). It installs a ticket and a set of trusted-authority certificates for the user upon successful login. In the protocol, U is a user who attempts to log in a host H; *passwd* is the password entered by U; T is a timestamp; L is the lifetime of a ticket; n is a nonce; h_1 and h_2 are publicly known one-way functions; k is a (DES) session key; k_U, k_{LEAF}, and k_A are respectively the public keys of U, the LEAF server, and a trusted authority A of U; and k_U^{-1} and k_{LEAF}^{-1} are the private keys of U and LEAF, respectively.

In step (1), user U enters its ID and password. In step (2), H applies the one-way function h_1 to the password U entered and sends the result, along with a timestamp T and a nonce n, in a message to LEAF. Upon receiving the message from H, LEAF forwards a request to CDC for U's private key. This key is stored as a record $(\{k_U^{-1}\}_{h_2(password_U)}, h_1(password_U))$ in CDC. Note that a compromise of CDC would not reveal these private keys. In step (4), CDC sends the requested private-key record to LEAF using a temporary session key k. In step (5), LEAF recovers both $\{k_U^{-1}\}_{h_2(password_U)}$ and $h_1(password_U)$ from CDC's reply. LEAF then verifies *passwd* by checking $h_1(passwd)$ against $h_1(password_U)$. If they are not equal, the login session is aborted and the abortion logged. Because $h_1(password_U)$ is not revealed to any principal except LEAF, password guessing attacks would require contacting LEAF for each guess or compromising LEAF's private key.

Having determined the password to be valid, LEAF sends the first part of the private-key record encrypted by n to H in step (6). (The nonce n sent in step (2) is used as a symmetric key for encryption.) In step (7), H recovers k_U^{-1} by decrypting the reply from LEAF first with n and then with $h_2(passwd)$. H then generates a pair of delegation keys and creates a ticket $tick_U$. In step (8), H requests the public-key certificate for a trusted authority of U from CDC. CDC replies with the certificate in step (9). In fact, multiple certificates can be returned in step (9) if U trusts more than one CA. These trusted authorities' certificates were previously deposited in the CDC by U using the enrollment protocol.

The authentication exchange protocol between a client C and a server S is shown is Figure 10. To simplify the protocol specification so that a single public-key certificate is sent in step (2) and in step (5), we made the following assumption: Let C's public-key certificate be signed by A_C where A_C denotes a trusted authority of S. Similarly, let S's public-key certificate be signed by A_S where A_S denotes a trusted authority of C. T is a timestamp and k is a (DES) session key.

In step (1), C requests S's public-key certificate from CDC. In step (2), CDC returns the requested certificate. C can verify the public-key certificate by decrypting it with k_{A_S}, which is the public key of A_S obtained by C when it executed the credential initialization protocol. In step (3), $tick_C$ (referred to as $tick_U$ in the credential initialization protocol) and the private delegation key k_d^{-1} (generated in step (7) of the credential initialization protocol), along with a new session key k, are sent to S. Only S can recover k from $\{k\}_{k_S}$ and subsequently decrypt $\{k_d^{-1}\}_k$ to recover k_d^{-1}. Possession of $tick_C$ and knowledge of the private delegation key constitute sufficient proof of delegation from C to S. However, if such delegation from C to S is not needed, $\{\{k\}_{k_S}\}_{k_d^{-1}}$ is sent in step (3) instead of $\{k_d^{-1}\}_k$; this acts as an authenticator for proving C's knowledge of k_d^{-1} without

$$
\begin{array}{lll}
(1) & C \rightarrow \mathrm{CDC} & : \; S \\
(2) & \mathrm{CDC} \rightarrow C & : \; \{S, k_S\}_{k_{A_S}^{-1}} \\
(3) & C \rightarrow S & : \; T, \{k\}_{k_S}, tick_C, \{k_d^{-1}\}_k \\
(4) & S \rightarrow \mathrm{CDC} & : \; C \\
(5) & \mathrm{CDC} \rightarrow S & : \; \{C, k_C\}_{k_{A_C}^{-1}} \\
(6) & S & : \; \text{recover } k \text{ from } \{k\}_{k_S} \\
 & & : \; \text{recover } k_d^{-1} \text{ from } \{k_d^{-1}\}_k \\
 & & : \; \text{recover } k_d \text{ from } tick_C \\
 & & : \; \text{verify that } k_d \text{ and } k_d^{-1} \text{ form a delegation key pair} \\
(7) & S \rightarrow C & : \; \{T + 1\}_k
\end{array}
$$

FIGURE 10 SPX client-server authentication protocol.

revealing it. In steps (4) and (5), S requests C's public-key certificate, which is used to verify $tick_C$ in step (6). In step (7), S returns $\{T + 1\}_k$ to C to complete mutual authentication between C and S.

Since SPX is a relatively recent proposal, its security properties have not been studied extensively. Such study would be necessary before it could be generally adopted.

Although SPX offers services similar to those of Kerberos, its elimination of on-line trusted authentication servers and the extensive use of hierarchical trust relationships are intended to make SPX scalable for very large distributed systems.

CONCLUSION

With the growth in scale of distributed systems, security has become a major concern—and a limiting factor—in their design. For example, security has been strongly advocated as one of the major design constraints in both the Athena and Andrew projects. Most existing distributed systems, however, do not have a well-defined security framework and use authentication only for their most critical applications, if at all.

Various authentication needs for distributed systems have been described in this chapter, and some specific protocols are presented. Most of them are practically feasible with today's technology and their adoption and use should be just a matter of need.

ACKNOWLEDGEMENTS

We thank Clifford Neuman of the University of Washington and John Kohl of the Massachusetts Institute of Technology for reviewing the section on Kerberos, and Joseph Tardo and Kannan Alagappan of Digital Equipment Corporation for reviewing the section on SPX. We are also grateful to the anonymous referees for their constructive comments.

ENDNOTES

1. This work was sponsored by grants from the Texas Advanced Research Program, National Science Foundation, and the NSA INFOSEC University Research Program.
2. Note that only P and A know k to be fresh.

3. Secure connection release is seldom addressed in the literature. Although a premature release (for example, one forced by a saboteur) may not cause problems with respect to confidentiality or integrity, it is a potential denial of service attack.
4. An implicit key handshake is performed when the session key is first used, for example, in the first transmission of user data.
5. Kerberos in the protocol refers to a Kerberos server.
6. In practice, f may not be one-to-one. It suffices to require that given two distinct elements x and y, the probability of $f(x)$ being equal to $f(y)$ is negligible.
7. Note that each client process is associated with a unique user who created the process. It inherits the user ID and the ticket-granting ticket issued to the user during login.

REFERENCES

1. M. Abadi, M. Burrows, C. Kaufman, and B.W. Lampson. Authentication and delegation with smart-cards. *Science of Computer Programming,* 21(2):93–113, October 1993.
2. S.M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *Proceedings of Usenix Winter Conference,* pages 253–267, Dallas, TX, January 1991.
3. M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *ACM Transactions on Computer Systems,* 8(1):18–36, February 1990.
4. D.E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.
5. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory,* IT-29(2):198–208, March 1983.
6. M. Gasser, A. Goldstein, C. Kaufman, and B.W. Lampson. The Digital distributed system security architecture. In *Proceedings of 12th National Computer Security Conference,* pages 305–319, Baltimore, Maryland, October 1989.
7. M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of 11th IEEE Symposium on Research in Security and Privacy,* pages 20–30, Oakland, California, May 7–9 1990.
8. C. Kaufman. *DASS Distributed Authentication Security Service,* RFC 1507. September 1993.
9. J.T. Kohl, B.C. Neuman, and T.Y. Ts'o. The evolution of the Kerberos authentication system. In F. Brazier and D. Johansen, editors, *Distributed Open Systems,* pages 78–94. IEEE Computer Society Press, 1994.

10. J. Linn. Practical authentication for distributed computing. In *Proceedings of 11th IEEE Symposium on Research in Security and Privacy,* pages 31–40, Oakland, California, May 7–9 1990.

11. R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM,* 21(12):993–999, December 1978.

12. B.C. Neuman and T.Y. Ts'o. An authentication service for computer networks. *IEEE Communications Magazine,* 32(9):33–38, September 1994.

13. J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of Usenix Winter Conference,* pages 191–202, Dallas, TX, February 1988.

14. J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of 12th IEEE Symposium on Research in Security and Privacy,* pages 232–244, Oakland, California, May 20–22 1991.

15. T.Y.C. Woo, R. Bindignavle, S. Su, and S.S. Lam. SNP: An interface for secure network programming. In *Proceedings of Usenix Summer Technical Conference,* Boston, Massachusetts, June 6–10 1994. Available from http://www.cs.utexas.edu/users/lam/NRL/.

16. T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer,* 25(1):39–52, January 1992. See also "Authentication" revisited. *Computer,* 25(3):10, March 1992.

17. T.Y.C. Woo and S.S. Lam. Design, verification, and implementation of an authentication protocol. In *Proceedings of International Conference on Network Protocols,* Boston, Massachusetts, October 25–28 1994. Available from http://www.cs.utexas.edu/users/lam/NRL/.

18. T.Y.C. Woo and S.S. Lam. A lesson on authentication protocol design. *ACM Operating Systems Review,* 28(3):24–37, July 1994.

19. CCITT Recommendation X.509. The Directory—Authentication framework, 1988. See also ISO/IEC 9594-8, 1989.