XEROX

Xerox System Integration
Standard

Authentication Protocol

XEROX

Xerox System Integration Standard

Authentication Protocol

XSIS 098404
April 1984

Xerox Corporation
Stamford, Connecticut 06904

Preface

This document is one of a family of publications that describes the network protocols underlying Xerox Network Systems (XNS).

This version of the Authentication Protocol Standard replaces the previous version of the same document (XSIS 098210, October 1982). It represents a major extension of the facility defined by the earlier document, while retaining backward compatibility. In addition to the "Simple" form of authentication defined in the earlier version, this revision defines "Strong" Authentication by specifying the data types, remote procedures and other protocol facilities needed to provide this stronger encryption-based form of network security. It also defines additional protocol operations to support the use of the old Simple form of credentials, including a new operation to be used by servers to check the validity of Simple credentials.

Xerox Network Systems comprise a variety of digital processors interconnected by means of a variety of transmission media. System elements communicate both to transmit information between users and to economically share resources. For system elements to communicate with one another, certain standard protocols must be observed.

Comments and suggestions on this document and its use are encouraged. Please address communications to:

Xerox Corporation
Office Systems Division
Network Systems Administration Office
3450 Hillview Avenue
Palo Alto, California 94304

Table of contents

1 Introduction

Security is an important aspect of any information handling system. It is a particularly difficult problem in a distributed system where network resources are constantly changing. The Authentication Service solves part of the security problem, that of guaranteeing the identity of network users.

1.1 Purpose

This document defines the complete specification of the protocol employed for interactions between clients and the Authentication Service. This document also serves as a guide for using the Authentication Service. It does not describe any particular implementation of the Authentication Protocol.

The Authentication Service is an element in providing information security on the network. It provides a method of protecting user passwords, of verifying a user's identity, and of protecting interactions between clients and other services. It does not, however, require that either clients or other services make use of these facilities.

1.2 Relationship to other protocols

The protocol defined in this document is an application-level protocol. It employs several other protocols. Requests to an Authentication Service are communicated using the request-reply or transaction discipline defined by Courier [8]. Every request is modeled as a remote procedure as defined in Courier; every exceptional condition that may arise is modeled as a remote error. All parameters transferred between the client and the Authentication Service obey the conventions described in the Courier specification. This present specification, therefore, constitutes a Courier remote program.

The Authentication Protocol also depends upon the standard time format defined in the Time Protocol [11], and on the standard definition of Clearinghouse™ Directory Service names defined in the Clearinghouse Protocol [7].

1.3 Documentation conventions

Courier text and examples are depicted in special fonts, and generally conform to a certain style. The rules and style are set forth below.

1.3.1 Notation

Throughout this document, special fonts are used to depict Courier text instead of using quote marks or other delimiters. This convention also aids the eye in discriminating between Courier text and the exposition. Items in THIS FONT indicate elements of the Courier language and are almost always in upper case. **This font** indicates items that are defined using the Courier language.

Identifiers that are defined in this protocol (as opposed to being defined by Courier) will have their first letter capitalized if they are the name of a type, error, or procedure; identifiers with a lower case first letter are usually the names of variables, arguments, or results.

1.3.2 Notation for examples

In the examples that follow, a call to a remote procedure is denoted by the name of the procedure followed by the arguments supplied to it. A return from a remote procedure is denoted simply by the results, preceded—when confusion might otherwise result—by the keyword RETURNS. The argument or result list is modeled as a record; the arguments or results as the record's components. Accordingly, Courier's standard notation for record constants is used to specify argument and result lists.

For example, if the procedure **Add** is defined as:

```
Add: PROCEDURE [first, second: CARDINAL]
RETURNS [sum: CARDINAL] = 99;
```

then a call to that procedure would be denoted by:

```
Add [first: 7, second: 5]
```

and the call would yield the result:

[sum: 12] or **RETURNS [sum: 12]**

Note: The above notation for procedure calls should not be confused with the standard notation for a record constant selected by means of a CHOICE data type. The two are similar in appearance but otherwise unrelated.

Examples of remote errors are either just the name of the error, if it is defined without arguments:

Overflow

or the same as a procedure call if it is defined with arguments.

For example, if **Overflow** were defined as:

Overflow: ERROR [carry: CARDINAL] = 99;

then an example of its use might be:

Overflow [carry: 1]

indicating that **Overflow** was reported with argument **carry** having the value 1.

Courier requires values for a **SEQUENCE OF UNSPECIFIED** to be a sequence of numbers. To retain readability in examples, the content of a **SEQUENCE OF UNSPECIFIED** is described using Courier notation. The reader should understand that the numeric representation of these types is what should be used as the content of the sequence.

1.4 Document organization

Section 2 of this document gives an overview of the concepts defined in the standard and how the identity of a client should be verified. Section 3 defines the meaning and use of the various types of credentials and verifiers that are interpreted by the Authentication Service, and defines the remote procedures needed to interact with an Authentication Service. Section 4 defines the remote errors reported by the Authentication Service when exceptional conditions occur. Section 5 describes the different algorithms used by the Authentication Service for encryption, hashing, key transformation, and verifier calculation.

Appendix A lists other documents which supplement the specification. Appendix B lists the Authentication remote program in its entirety. Appendix C lists the well-known names of the distributed services. Appendix D gives an example of an interaction between a client, a service, and the Authentication Service using the Authentication Protocol.

2 Overview

The Xerox Network System is an open, distributed system. The devices attached to the internet are heterogeneous and autonomous and may be under the control and administration of widely separated groups of people. In such an environment it is important to provide protection from unauthorized use of services, and to provide privacy for data stored by clients. Two things are required before a service can provide this kind of security: the service must provide access control machinery, and the service must have some means of determining the identity of its clients. Access control mechanisms allow the group which controls a particular service to determine who is allowed to access that service and what those people are allowed to do. Determining the identity of a client allows the access control machinery to know with some degree of certainty exactly which person is trying to access the service. Neither proof of identity nor access control is any good without the other.

The Authentication Protocol addresses the problem of determining identities. It provides a means whereby a mutually suspicious client and service may prove their identities to each other. It is based upon a design by Needham and Schroeder [4].

2.1 Clients and services

A *service* is an entity that runs on a system element called a *server*. The term *Authentication Service* refers to both an implementation of the Authentication abstraction and to the tasks performed by that implementation. An Authentication Service is accessible to and used by software entities on other system elements called *clients*.

A client always interacts with an Authentication Service on behalf of a *user*. The user may be a human being, or it may be a software entity such as another service. All interaction between the client and the service is initiated by the client. The service never initiates activity with a client.

2.2 How authentication works

The fundamental problem of authentication is how to get the data that proves who one is to a recipient in a fashion that protects against the following two kinds of impersonation:

a) impersonation enabled by theft of identifying information (for example, a person's name and password);

b) impersonation by play-back of recorded information (for example, selected portions of a dialog between sender and recipient).

What makes the authentication process particularly elaborate is that the entire exchange of information between the various parties takes place over an unsecured communication medium. The exchange must be done in such a fashion that an intruder, if he were to monitor the entire conversation, could learn nothing that would compromise the process.

2.2.1 The authentication model

The solution provided by the Authentication Protocol postulates a secure Authentication Service which knows the specially-encrypted password, called a *key*, of every user and service in the internet. A sender, called the *initiator*, which wishes to authenticate itself to a receiver, called the *recipient*, contacts the Authentication Service, telling it the names of both parties and supplying a random number called the *nonce*. The Authentication Service sends back to the initiator several pieces of encrypted information that can be decoded only by the initiator (or anyone with the initiator's key). These pieces are: the *credentials*, the *nonce*, the recipient's name, and the *conversation key*.

The credentials are encrypted with the recipient's key. When decrypted, it contains the name of the initiator, the conversation key, and a date and time, called the *expiration time*, after which the credentials are no longer valid. (The expiration interval is fixed for an internet and is determined by the implementation of the Authentication Service on that internet. It should be on the order of many hours.) The nonce and recipient's name are included to protect against a replay of a previous conversation by a bogus Authentication Service. These two values should be checked by the initiator to make sure that they are the same as those submitted to the Authentication Service. Since only the recipient can decrypt the credentials, and since only the Authentication Service (and the recipient) can encrypt the credentials, the recipient knows the initiator is authentic. What the recipient isn't sure of is whether or not the credentials are a recording.

The decrypted conversation key is used by the initiator to encrypt a time stamp, called a *verifier*. When the initiator sends its credentials to the recipient, a verifier is included. The recipient decrypts the verifier and determines if the time stamp is reasonably close to the current time. This allows the recipient to detect a replay. Only the initiator could encrypt the verifier (the time stamp) and only the recipient could decrypt it, since only these two parties know the conversation key. To be effective a verifier may be used only once. Any other alternative would compromise the authentication process by allowing recorded verifiers to be replayed.

The logic of this process is as follows: the recipient receives, in a protected manner, a key (the conversation key) that only the initiator and recipient know; the reception of a proper verifier demonstrates that the sender knows the conversation key, and therefore must be the initiator; and the time stamp prevents an intruder from reusing the verifier.

2.2.2 Some assumptions

The authentication procedure is based on several assumptions. If any of these assumptions does not hold true, either the authentication process is compromised, or it cannot be implemented. The first is the assumption that there exists a secure Authentication Service with which all clients and services can entrust a copy of their keys. For each client and service, only the client or service and the Authentication Service know the key. Secondly, it is assumed that the encryption algorithm is secure even if an intruder can obtain known plain-text/cypher-text pairs. And lastly, it is assumed that the initiator's and recipient's clocks are reasonably well synchronized, and that there is a reasonable upper bound on communication delays.

2.3 Levels of security

Different system elements have different capabilities. Some can efficiently perform encryption while others cannot; still others must act on behalf of a client at the other end of an unsecured telephone line. To accommodate the security needs and capabilities of all, there are two different levels of security, called *strong* and *simple*. The goal of strong security is to make it, practically speaking, impossible for one user to impersonate another, whereas simple security makes it merely difficult. Neither strong nor simple passwords are ever transmitted in the clear. This limits the amount of damage done if the user accidentally types his strong password in a situation that expects the simple password. In such cases the datum transmitted is a value which is easily derived from the password but which does not contain enough information to reconstruct the original password.

The two levels of authentication are reflected throughout the Authentication Protocol. There are two levels of passwords, keys, credentials, and verifiers.

2.4 Passwords

Every user has two passwords, one for each level of security. A service has only one, a strong password.

2.4.1 Passwords for a user

Every user has two passwords, his *strong* and *simple* passwords. Which password the user must type depends on what sort of workstation he is using. If he is using a workstation powerful enough to perform encryption, then he should use his strong password. If he is using a workstation which is unable to perform encryption, or if he is using a remote terminal at the end of an unsecured telephone line, then he should use his simple password. It is up to the workstation software to make clear which password is required when the user logs on.

When a user enters his password into a system element, it is encoded according to a specific algorithm, one for each type of password (see 2.5). The result is a *key* called, respectively, a strong key or a simple key. The passwords are never transmitted on the internet unencoded. The strong key and simple password are also registered with the Authentication Service.

2.4.2 Passwords for a service

A service has only one password, a strong password. The authentication procedure is arranged so that the service never needs a simple password to verify the authenticity of a client (to the corresponding level of security, simple). Thus, simple passwords for a service are unnecessary.

2.5 Data protection

There are two principal methods employed in the Authentication Protocol to protect confidential data—hashing and the National Bureau of Standards' Data Encryption Standard (DES). All verifiers and the strong credentials utilize one of these two methods. The details of each are described in Section 5.

The transformation of a password into a key is also described in Section 5.

2.6 Credentials and verifiers

There are two levels of credentials and verifiers corresponding to the two levels of security. The structure and use of each are detailed below.

2.6.1 Strong credentials and verifiers

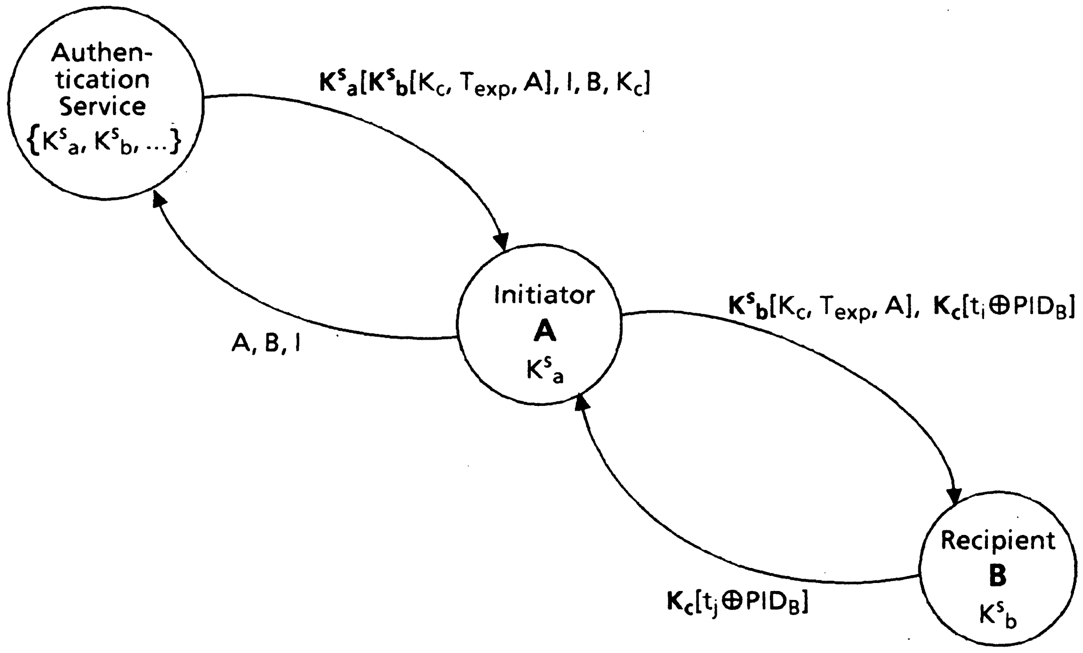
Strong credentials contain the fully qualified, distinguished name of the initiator (A), the expiration date and time of the credentials (T_{exp}), and a strong conversation key (K_C). A new strong conversation key is created by the Authentication Service for each request for credentials. It is a key in the same sense that the initiator and recipient have keys. The Authentication Service encrypts the concatenation of these three objects with the recipient's strong key (K^s_b). The verifier which accompanies a set of strong credentials is a time stamp (t_i) XORed with the recipient's processor ID, and encrypted by the initiator with the conversation key K_C. The process for strong authentication is shown in Figure 2.1.

A strong verifier may be used only once for each set of credentials; otherwise an eavesdropper could obtain a valid credentials-verifier pair. If the protocol specifies a verifier as part of a response, it would also be a strong verifier, but with a new time stamp. The computation of the value of the returned verifier is given in Section 5.

2.6.2 Simple credentials and verifiers

No interaction with an Authentication Service is required to manufacture a set of simple credentials. The credentials are simply A's name (A), either its distinguished name or an alias. (The distinguished name is recommended; a slight performance penalty may be associated with use of an alias in this context.) The verifier which accompanies a set of simple credentials is A's simple key, K^{si}_a. The simple authentication process is shown in Figure 2.2.

Unlike strong verifiers, simple verifiers are always the same for a given set of credentials. Since the credentials never have to be decoded by the initiator, this means that an eavesdropper may obtain a valid credentials-verifier pair and pose as the initiator indefinitely. Also, the returned verifier provides no assurance that B is not an imposter.


Authentication Service $\{K^s_a, K^s_b, \ldots\}$
$K^s_a[K^s_b[K_c, T_{exp}, A], I, B, K_c]$
A, B, I
Initiator **A** K^s_a
$K^s_b[K_c, T_{exp}, A], K_c[t_i \oplus PID_B]$
$K_c[t_j \oplus PID_B]$
Recipient **B** K^s_b

Legend

A, B	=	the fully qualified names of the initiator, A, and recipient, B
I	=	the nonce
K^s_a, K^s_b	=	the strong keys of A and B, respectively
K_c	=	the conversation key
T_{exp}	=	the expiration time for a set of credentials
$K_y[x]$	=	the value x encrypted with the key K_y
t_i	=	a time stamp obtained from the system clock at time i
PIDx	=	48-bit processor ID of X, padded after least significant bit with 16 bits of zero

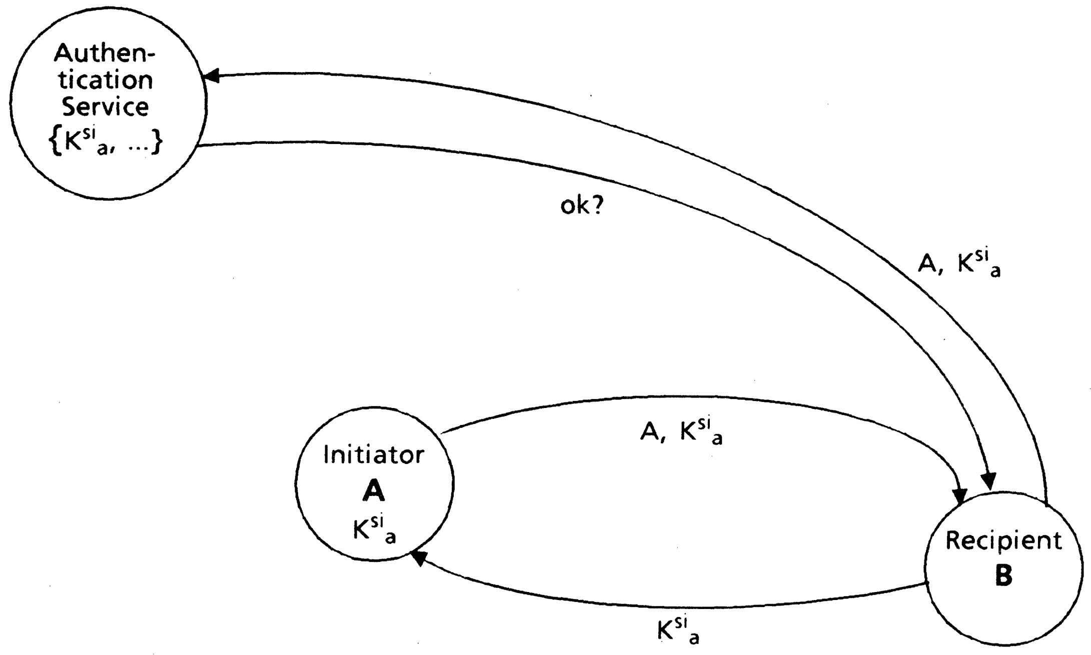
Figure 2.1 Strong authentication

2.6.3 Summary

Table 2.1 summarizes the credentials, verifiers, and conversation keys for the two levels of authentication.

2.7 Privileged users

The Authentication Protocol employs the notion of a *privileged user*. This user has special privileges with an Authentication Service. Such a user can act on behalf of any user in the same organization and domain (see the Clearinghouse Protocol [7]). After suitably authenticating itself a privileged user can register with or delete from the Authentication Service another user's strong key. The means by which privileged users are designated are dependent on the implementation and are not specified by this protocol.



Legend	
A	= the fully qualified name of the initiator, A
K^{si}_a	= A's simple key
ok?	= a boolean which indicates whether or not K^{si}_a is A's hashed password

Figure 2.2 Simple authentication

2.8 The Authentication database

Although it appears to clients as though there is only one Authentication Service per internet, the service is distributed over many servers. Each Authentication server has responsibility for some part of the entire Authentication database. The part of the database maintained by each Authentication server is called its local database. Parts of the Authentication database may be replicated on other servers. If credentials are required for A and B, and the keys for them are stored in different local databases, both local databases must be accessible to the Authentication Service before credentials can be generated.

The distributed nature of the database has two ramifications concerned with its modification. First, when a key is changed, there may be an appreciable delay before all copies of the new key are made consistent. Clients must be able to properly contend with the

Table 2.1 Summary of credentials, keys, and verifiers

	Credentials	Conversation Key	Verifier
Strong	$K^s_b[K_c, T_{exp}, A]$	K_c	$K_c[t_i \oplus PID_B]$
Simple	A	---	K^{si}_a

database being transiently incorrect. Second, if two copies of a key are modified "simultaneously" by two clients, the Authentication Service will see to it that, after the updating transient has passed, all copies of the key will be that which corresponds to the most recent change. This means that all prior updates will be rejected. To users, the update which prevails will appear to be unpredictable.

2.9 Guidelines for authenticating an initiator

To complete the process of authentication, the initiator and recipient must check the credentials and verifier. Below are some guidelines for doing this.

2.9.1 Checking strong credentials

Encrypted strong credentials must be a multiple of 64 bits in length (see 3.2.1). Before decryption, the length of the credentials should be checked to see if it is of the required multiple. When decrypted, the credentials should have the structure described in the protocol, the padding bits should be zero, and each field of the initiator's name should have meaningful sizes (zero or greater, and less than or equal to the maximum) and content.

The credentials should not have expired; i.e., the expiration time of the credentials should not be earlier than the current time. Also, the expiration time should be a reasonable time in the future; excessively large expiration times (more than several days) should be regarded with suspicion.

The parity bits in the strong conversation key should be correct (see 3.5.1).

When decrypted and XORed with the recipient's left-justified, zero-filled processor ID, the verifier should be within a reasonable interval—either way—of the current time (on the order of a small number of minutes).

The time stamps in the verifiers received should be strictly monotonically increasing for this set of credentials.

2.9.2 Checking simple credentials

The initiator should be a valid name; i.e., the lengths of the name fields should all be zero or greater, and less than or equal to the maximum.

The user's simple key, as stored in its Clearinghouse entry, should match the verifier.

2.10 Protection provided by the Authentication Service

The threats that can be brought against the Authentication Protocol are listed below along with how much protection the protocol offers against each one of them.

- Intruders may get information from client protocols by eavesdropping; that is, watching file or mail service interactions in order to steal files or mail.

- There is no protection against an intruder modifying the argument of a Courier call. The Authentication Service interactions are secure, but existing client protocols are not.
- A bogus Time Service could allow an expired credentials and its associated verifier sequence to be replayed. In order to do this the intruder needs to first build a bogus Time Service and then cause the recipient to reset his clock from this bogus Time Service.
- Intruders cannot pose as an Authentication Service because they don't have the initiator's strong key.
- Protocols which do not have reply verifiers in the results cannot prevent an intruder from posing as that kind of service.
- There is no protection against intruders who have the computational power to "break" the DES encryption algorithm. The protocol does make an effort to prevent an intruder from obtaining plain-text/cypher-text pairs.

3 Remote procedures

The Authentication Service implements the authentication operations as a Courier [8] remote program. Courier is especially well suited for these operations since Courier handles the problems of reliable delivery across networks.

Each procedure description includes a declaration of the procedure in Courier's standard notation, a description of the procedure's arguments and results, and occasionally an example of its use.

The following definition gives the program and version numbers of **Authentication** and lists all other Courier-based protocols which are referenced from this program.

```
Authentication: PROGRAM 14 VERSION 2 =
BEGIN
    DEPENDS UPON
        Clearinghouse (2) VERSION 2,
        Time (15) VERSION 2;
    ...
END.
```

This indicates that **Authentication** is program number 14. This document defines version 2. **Authentication** references some types and constants that are defined in other protocols as shown in the above declaration. These protocols are documented in the Clearinghouse Protocol [7] and the Time Protocol [11].

3.1 Encoded data

There are two basic methods employed in the Authentication Protocol for protecting sensitive data: the National Bureau of Standards' Data Encryption Standard, and a hashing algorithm. The National Bureau of Standards' Data Encryption Standard (DES) [1] is based on an invertible function which accepts as input parameters a 64-bit key and a 64-bit block of plain text, and produces a 64-bit block of cipher text. The Authentication Protocol, in support of DES, defines two types:

```
Key: TYPE = ARRAY 4 OF UNSPECIFIED;

Block: TYPE = ARRAY 4 OF UNSPECIFIED;
```

A **Key** must have a particular form. The least significant bit of each octet of the key acts as a parity bit (thus, the unconstrained data in a key consists of only 56 bits). Each bit is set so as to make the parity of the octet odd.

The hashing algorithm is employed where there is either reduced computational power available or reduced security requirements. The type **HashedPassword** is defined in support of this algorithm:

```
HashedPassword: TYPE = CARDINAL;
```

Details of the algorithms employed in the Authentication Protocol to encode sensitive data and how these data types are used can be found in Section 5.

3.2 Credentials

There are two forms of credentials: strong and simple. Strong credentials contain data which has been encrypted using DES. This section describes the formats of these data structures when they are not encrypted. How these structures are encrypted and decrypted is addressed in Section 5.

All credentials are defined as follows:

```
Credentials: TYPE = RECORD [
   type: CredentialsType,
   value: SEQUENCE OF UNSPECIFIED];

CredentialsType: TYPE = {simple(0), strong(1)};
```

To retain compatibility with the previous version of the Authentication Protocol the following variable is defined:

```
simpleCredentials: CredentialsType = simple;
```

3.2.1 Strong credentials

Strong credentials are associated with a **CredentialsType** of **strong**. When the **value** component of **Credentials** is decrypted with the recipient's strong key, the result is an object of the following form in its standard representation:

```
StrongCredentials: TYPE = RECORD [
   conversationKey: Key,
   expirationTime: Time.Time,
   initiator: Clearinghouse.Name];
```

To properly encrypt strong credentials, the length of the record must be a multiple of 64 bits. If a particular instance of **StrongCredentials** is not a multiple of 64 bits, it is understood that enough zero bits are appended to the instance to make it so.

conversationKey is used by the initiator to encrypt verifiers and by the recipient to decrypt them. **expirationTime** is the date and time after which the credentials will no longer be accepted. **initiator** is the name on which access control decisions will be based.

StrongCredentials is encrypted with the "Cipher Block Chaining with Checksum" mode of DES (see 5.2). The result is a series of **Blocks**. The **value** component of **Credentials** effectively encapsulates the **Block**. The sequence of 16-bit words that constitutes **value** is to be interpreted as a data object of type **Block** in its standard representation.

3.2.2 Simple credentials

Simple credentials are associated with a **CredentialsType** of **simple**. The **value** component of **Credentials** is unencrypted data of the following form:

```
SimpleCredentials: TYPE = Clearinghouse.Name;
```

The **value** component of **Credentials** effectively encapsulates **SimpleCredentials**. The sequence of 16-bit words that constitutes **value** is to be interpreted as a data object of type **SimpleCredentials** in its standard representation.

3.3 Verifiers

All types of verifiers are defined as:

```
Verifier: TYPE = SEQUENCE 12 OF UNSPECIFIED;
```

There are two forms of verifiers: strong and simple. There is no way to tell the type of a particular verifier in isolation. The type of a verifier is the same as the type of the credentials which it accompanies.

3.3.1 Strong verifiers

A **Verifier**, when associated with strong credentials, is data which has been encrypted with the "Electronic Code Book" mode of DES. When a strong verifier is decrypted with the conversation key from the accompanying credentials and XORed with the recipient's left-justified, zero-filled processor ID, the result is an object of the following form:

```
StrongVerifier: TYPE = RECORD [
    timeStamp: Time.Time,
    ticks: LONG CARDINAL];
```

The **timeStamp** field is the time the verifier was created, derived from the clock of the initiator's system element. For a verifier to be acceptable, the **timeStamp** it contains must be close to the clock of the recipient's system element. This prevents an intruder from replaying an earlier conversation. The **ticks** field subdivides each second (the resolution of a **Time.Time**) to allow multiple verifiers to be created per second. **ticks** increases monotonically within a given second, but does not necessarily start at zero.

When a strong verifier is specified in the protocol, the sequence of 16-bit words that constitutes a **Verifier** (that is, the encrypted **StrongVerifier**) is to be interpreted as a data object of type **Block** in its standard representation.

3.3.2 Simple verifiers

A **Verifier**, when associated with simple credentials, is data which has been encoded with the hashing algorithm defined in the Authentication Protocol (see 5.1). A simple verifier has the following form:

```
SimpleVerifier: TYPE = HashedPassword;
```

This value is derived from the initiator's simple password. This value is computed by the initiator and checked by the recipient. Since simple verifiers do not change from call to call, simple credentials are susceptible to being replayed by an intruder who can eavesdrop on the network.

When a simple verifier is specified, the 16-bit word that constitutes a **Verifier** is to be interpreted as a data object of type **SimpleVerifier** in its standard representation.

3.4 Obtaining credentials

To obtain a set of strong credentials, the initiator calls the remote procedure:

```
GetStrongCredentials: PROCEDURE [initiator, recipient: Clearinghouse.Name,
    nonce: LONG CARDINAL]
RETURNS [credentialsPackage: SEQUENCE OF UNSPECIFIED]
REPORTS [CallError] = 1;

CredentialsPackage: TYPE = RECORD[
    credentials: Credentials,
    nonce: LONG CARDINAL,
    recipient: Clearinghouse.Name,
    conversationKey: Key];
```

Arguments: **initiator** is the name of the initiator to be authenticated. **recipient** is the name of the recipient to whom the initiator wishes to prove his identity. **nonce** is a random number. Its inclusion protects against replays by a bogus Authentication service.

It is occasionally necessary (for instance, in order to modify a strong key) to authenticate oneself to the Authentication Service. To do this the name of the Authentication Service must be known. Since any instance of the Authentication Service will suffice, the service is given a "well-known" name that will work with all instances. The well-known name of the Authentication Service is:

```
Authentication Service:CHServers:CHServers
```

See Appendix C for the well-known names of other services.

Results: **credentialsPackage**, when decrypted using the initiator's strong key, is a record of the form **CredentialsPackage**. It contains: credentials encrypted with the recipient's strong key, the nonce value, the recipient's name, and the conversation key which is used to encrypt verifiers to go with these credentials. The client should verify that the nonce and recipient name are the same that were specified in the call.

CheckSimpleCredentials requests the Authentication Service to verify that the initiator identified by the credentials has submitted the correct password.

```
CheckSimpleCredentials: PROCEDURE [credentials: Credentials, verifier: Verifier]
RETURNS [ok: BOOLEAN]
REPORTS [AuthenticationError, CallError] = 2;
```

Arguments: **credentials** and **verifier** must be the simple credentials and verifier of the client.

Result: **ok**, if TRUE, indicates that the hashed password registered for the client is the same as that submitted as the verifier.

3.5 Modifying keys

In order to create credentials, the Authentication Service must know the keys of both of the parties involved. This section describes how the database of keys maintained by the Authentication Service may be added to, updated, and deleted from.

3.5.1 Strong keys

CreateStrongKey registers a strong key with the Authentication Service. Only a privileged user is allowed to create new strong keys.

```
CreateStrongKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name, key: Block]
REPORTS [AuthenticationError, CallError] = 3;
```

Arguments: **credentials** and **verifier** must be the strong credentials and verifier of a privileged user. **name** specifies the owner of the key. It must be registered in the Clearinghouse so that it can be looked up by the Authentication Service. **key** is the strong key to be registered, encrypted using the ECB mode of DES and the conversation key contained in the credentials.

ChangeStrongKey changes a user's strong key registered with an Authentication Service. Only the owner of a strong key may change it. The ability to change a strong key is determined by the way in which the internet is administered. Administrative rules may not allow strong keys to be changed by a remote procedure call in which case calling this procedure may cause an error to be reported.

```
ChangeStrongKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    newKey: Block]
REPORTS [AuthenticationError, CallError] = 4;
```

Arguments: **credentials** and **verifier** must be the strong credentials and verifier of the owner. **newKey** is the new strong key, encrypted using the ECB mode of DES, and the conversation key contained in the credentials.

DeleteStrongKey deletes a user's strong key registered with an Authentication Service. Only the owner of a strong key or a privileged user may delete it.

```
DeleteStrongKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name]
REPORTS [AuthenticationError, CallError] = 5;
```

Arguments: **credentials** and **verifier** must be the strong credentials and verifier of the owner or of a privileged user. **name** specifies the owner of the key to be deleted.

3.5.2 Simple keys

CreateSimpleKey registers a simple key with the Authentication Service. Only a privileged user is allowed to create new simple keys.

```
CreateSimpleKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name, key: HashedPassword]
REPORTS [AuthenticationError, CallError] = 6;
```

Arguments: **credentials** and **verifier** must be the strong credentials and verifier of a privileged user. **name** specifies the owner of the key. It must be registered in the Clearinghouse so that it can be looked up by the Authentication Service. **key** is the simple key to be registered, encoded using the hashing algorithm.

ChangeSimpleKey changes a user's simple key registered with an Authentication Service. Only the owner of a simple key may change it.

```
ChangeSimpleKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    newKey: HashedPassword]
REPORTS [AuthenticationError, CallError] = 7;
```

Arguments: **credentials** and **verifier** must be the strong credentials and verifier of the owner. **newKey** is the new simple key, encoded using the Authentication hashing algorithm. The ability to change a simple key is determined by the way in which the internet is administered. Administrative rules may not allow simple keys to be changed by a remote procedure call in which case calling this procedure may cause an error to be reported.

DeleteSimpleKey deletes a user's simple key registered with an Authentication Service. Only the owner of a simple key or a privileged user may delete it.

```
DeleteSimpleKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name]
REPORTS [AuthenticationError, CallError] = 8;
```

Arguments: **credentials** and **verifier** must be the strong credentials and verifier of the owner or of a privileged user. **name** specifies the owner of the key to be deleted.

3.6 Locating an Authentication server

There exists an Authentication operation that is not part of the Authentication Courier program. This operation, called **BroadcastForServers**, is used to locate an instance of the Authentication Service in the internet. It is invoked using the Packet Exchange Protocol

[9]. A broadcast is made on the designated network and each Authentication server on that network returns its network address.

The packet used is a standard Packet Exchange packet. The destination and source socket number is 21_{10}. The Packet Exchange Client Type has a value of 2. The format of the body of the packet (the level 3 part) is shown in Figure 3.1. The word numbers are relative to the beginning of the level 3 part of the Packet Exchange packet.

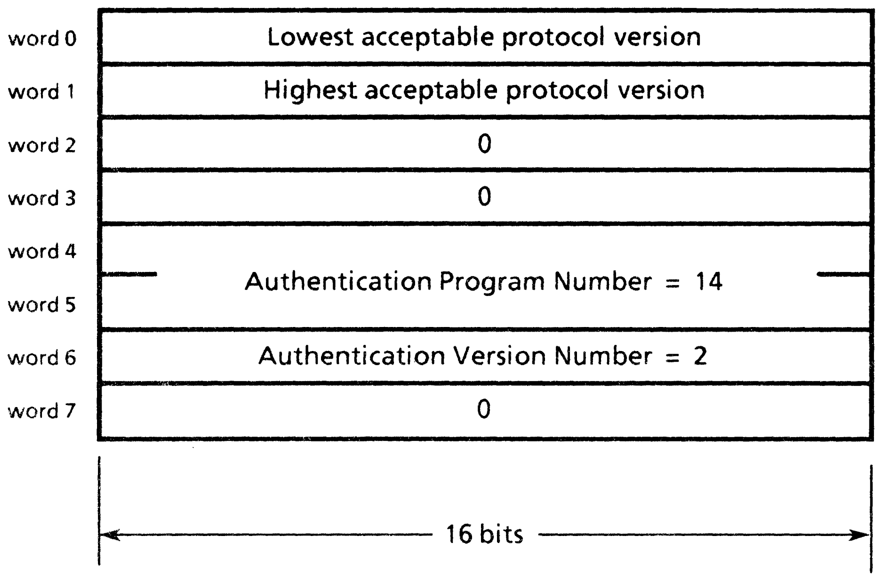

word 0
Lowest acceptable protocol version
word 1
Highest acceptable protocol version
word 2
0
word 3
0
word 4
word 5
Authentication Program Number = 14
word 6
Authentication Version Number = 2
word 7
0
16 bits

Figure 3.1 **BroadcastForServers** packet format

The lowest and highest acceptable version numbers indicate how the results are to be encoded. The values in these fields correspond to versions of Courier. The results will be encoded according to the version of Courier contained in the highest acceptable protocol version.

The result of the **BroadcastForServers** operation is contained in another Packet Exchange packet shown in Figure 3.2. This return packet is constructed so that the body of it is the same as Courier would return from a remote procedure call. (If the server cannot return results, because of an error or version mismatch, it will not respond to the broadcast.) The result is a Courier **ReturnMessageBody** which contains the network address of the responding Authentication server. This address is encoded the same as Courier would encode **Clearinghouse.NetworkAddress** (see the Clearinghouse Protocol [7]).

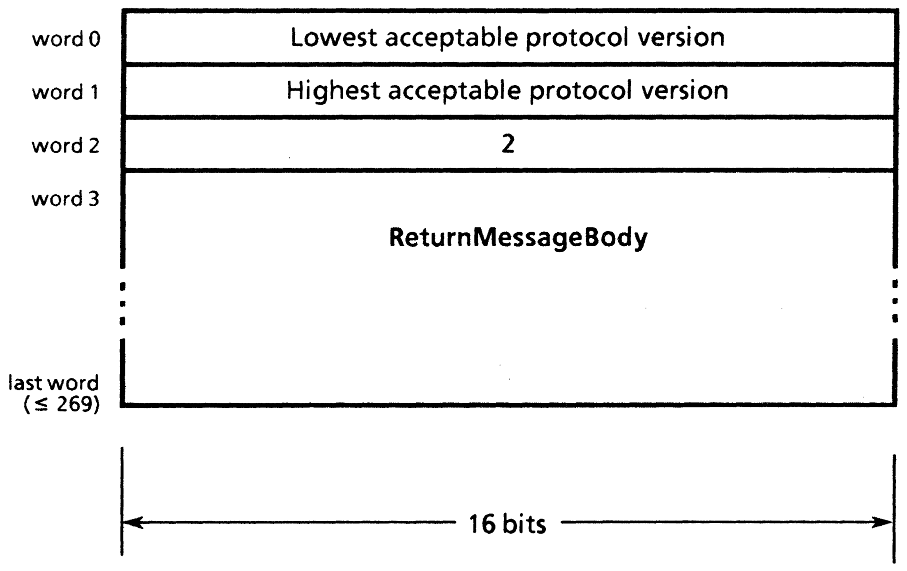

word 0
Lowest acceptable protocol version
word 1
Highest acceptable protocol version
word 2
2
word 3
ReturnMessageBody
last word (≤ 269)
16 bits

Figure 3.2 Response packet format

4 Remote errors

When a remote procedure completes successfully, it returns results as specified in the definition of the procedure. However, conditions can arise before or during execution of the procedure that make successful completion of the request impossible. For example, the client may have specified incorrect arguments in a remote procedure call, or some required resource may be unavailable.

When such conditions occur, an error is reported to communicate to the client the nature of the problem. Each error encompasses an entire class of possible conditions and the specific problem is further described by the arguments of the error. For example, **AuthenticationError** indicates that the Authentication server could not complete the requested operation. The particular problem is specified by the argument to **AuthenticationError**, which is of type **Problem**.

All remote errors are defined below. Each error definition includes a declaration of the error in Courier notation, and a description of its arguments.

4.1 Authentication errors

Authentication errors indicate that either the credentials or verifier was unacceptable. The exact reason for authentication failure is conveyed by the **Problem** parameter.

```
AuthenticationError: ERROR [problem: Problem] = 2;

Problem: TYPE = {
    credentialsInvalid(0),
    verifierInvalid(1),
    verifierExpired(2),
    verifierReused(3),
    credentialsExpired(4),
    inappropriateCredentials(5)};
```

credentialsInvalid indicates that the credentials were unacceptable. If the credentials were strong, this means the decryption failed. If the credentials were simple, it means that the name could not be found in the Clearinghouse.

verifierInvalid indicates that the verifier was unacceptable. If the verifier was strong, this means that the decryption failed; if simple, that the password (as looked up in the Clearinghouse) did not hash to the verifier.

verifierExpired indicates that the strong verifier is not recent enough. Verifiers more than a few minutes old are unacceptable. A bad or unsynchronized clock may cause good verifiers to appear to have expired.

verifierReused indicates that the recipient has seen either the same strong verifier before or one generated more recently. To prevent replays, a verifier is invalid once it has been exposed on the internet.

credentialsExpired indicates that the expiration date and time of the credentials has been exceeded. A bad or unsynchronized clock may cause good credentials to appear to have expired.

inappropriateCredentials means that the credentials accompanying a remote procedure call were not of the proper strength to perform the desired operation; strong credentials were submitted for a simple credentials operation or vice-versa. For example, simple credentials are too weak for the **CreateStrongKey** operation.

4.2 Calling errors

An operation may fail for the following reasons: a lack of resources, the service does not have the requested information, a communication failure, or a server crash. The rejection is reported as a **CallError**.

```
CallError: ERROR [problem: CallProblem, whichArg: Which] = 1;

Which: TYPE = {notApplicable(0), initiator(1), recipient(2), client(3)};
```

The argument **problem** describes the problem in greater detail. The argument **whichArg** indicates which argument caused the error.

```
CallProblem: TYPE = {
    tooBusy(0),
    accessRightsInsufficient(1),
    keysUnavailable(2),
    strongKeyDoesNotExist(3),
    simpleKeyDoesNotExist(4),
    strongKeyAlreadyRegistered(5),
    simpleKeyAlreadyRegistered(6),
    domainForNewKeyUnavailable(7),
    domainForNewKeyUnknown(8),
    badKey(9),
    badName(10),
    databaseFull(11),
    other(12)};
```

tooBusy means that the service is processing so many other client requests that it presently cannot handle the operation. The client should pause a short while and retry the operation, or try the operation on some other server. **accessRightsInsufficient** means that the client did not have sufficient access rights to complete the operation. For example, this error could be reported if a non-privileged user attempted to create a key. This error is also reported when, as the result of internet administrative rules, no user is allowed to change his strong key.

keysUnavailable indicates that the required Authentication server is down or not currently available via the internet. The **keysUnavailable** error may also occur when a **ChangeStrongKey** or **ChangeSimpleKey** operation is attempted, and the server in which the key is to be changed is down or currently unavailable. In both cases the name of the entity whose key is unavailable is indicated by **whichArg**.

The errors **strongKeyDoesNotExist** and **simpleKeyDoesNotExist** indicate that the key necessary to create a set of credentials for the initiator (strong key for strong credentials, simple key for simple) is not registered with the Authentication Service. These errors may also be returned by the **ChangeStrongKey** and **ChangeSimpleKey** operations. In all of these cases the name of the entity whose key is not registered is indicated by **whichArg**.

The errors **strongKeyAlreadyRegistered** and **simpleKeyAlreadyRegistered** may occur when a **CreateStrongKey** or **CreateSimpleKey** operation is attempted, and that key already exists. The errors **domainForNewKeyUnavailable** and **domainForNewKeyUnknown** may occur when a **CreateStrongKey** or **CreateSimpleKey** operation is attempted and the server on which the new key would be stored is down, unavailable, or unknown. In all of these cases the name of the entity whose key is to be registered is indicated by **whichArg**.

badKey is reported when the client attempts to add a key with bad parity bits to the database (via **CreateStrongKey** or **ChangeStrongKey**). **badName** implies a bad Clearinghouse name supplied to a create key operation. **databaseFull** indicates that the Authentication database has no more room to store data.

The error **other** exists so that new servers and procedures can be developed in a staged fashion; this value should never be reported by a correctly functioning server conforming to this protocol.

5 Algorithms

Certain algorithms are employed in the Authentication Protocol to protect data transferred over the internet. These algorithms, which specify both how data is encoded and how it is decrypted, are described in this chapter. Also, the algorithms for computing a conversation key and a returned verifier are defined.

5.1 Hashing algorithm

The source for a hashed password will normally be a user-supplied sequence of characters called a password. Such a password may contain any character in the Xerox Character Encoding Standard [5] except those in the Xerox Rendering Code Standard [10]. A **HashedPassword** must be computed from the password in the following fashion:

a) the following character substitution is made in the password: if the character is an upper-case ISO/ASCII character in the range "A" through "Z" in character set 0 of the Encoding Standard, it is replaced with the corresponding lower-case character; otherwise the character is unchanged;

b) every character is mapped to a 16-bit code as defined by the Encoding standard;

c) the resulting sequence of numbers is interpreted as a multiple-precision, unsigned, binary integer whose most significant bit is the left-most bit of the first character code, and whose least significant bit is the right-most bit of the last character code;

d) the value for a **HashedPassword** is the remainder when this integer is divided by 65,357.

<u>Example</u>:

If the password is "Temp," the corresponding value for a **HashedPassword** is:

$$\begin{aligned}\text{Hash["Temp"]} &= \text{Hash["temp"]}\\ &= (\text{["t"]}*2^{48} + \text{["e"]}*2^{32} + \text{["m"]}*2^{16} + \text{["p"]}) \bmod 65{,}357\\ &= (116*2^{48} + 101*2^{32} + 109*2^{16} + 112) \bmod 65{,}357\\ &= 18{,}335\end{aligned}$$

The reader should not anticipate that disclosure of a hashed password is a significantly less serious security concern than disclosure of the password itself. For any given value of

HashedPassword, it is computationally easy to find a string which hashes to that value. The hashing function is intended to prevent complete disclosure of the unhashed password in cases where the user accidentally enters his strong password by mistake.

5.2 DES encryption

The building block of the National Bureau of Standards' Data Encryption Standard (DES) [1] is an invertible function which accepts a 56-bit key and a 64-bit block of plain text and produces a 64-bit block of cipher text.

The output of this function depends equally on every bit of the key and every bit of the input. Changing one bit of the key or of the input will, on the average, cause half of the bits in the output to change. This function, in its simplest form, is called the *Electronic Code Book* (ECB) mode of DES. It is used in the Authentication Protocol to encrypt timestamps for strong verifiers, and to encrypt strong keys in the **CreateStrongKey** and **ChangeStrongKey** operations.

The basic Electronic Code Book mode of DES may be improved for pieces of plain text longer than 64 bits by feeding back the previous block's cipher text and XORing it with the plain text of the next block before encryption. This is a simple, reversible operation which makes duplicate plain text blocks produce different cipher text blocks. This mode of DES, called *cipher block chaining* (CBC) and described in the National Bureau of Standards' Data Encryption Standard [1], is shown in Figure 5.1. In the Authentication Protocol the initialization vector is always zero. The inverse of this function is easily derived.

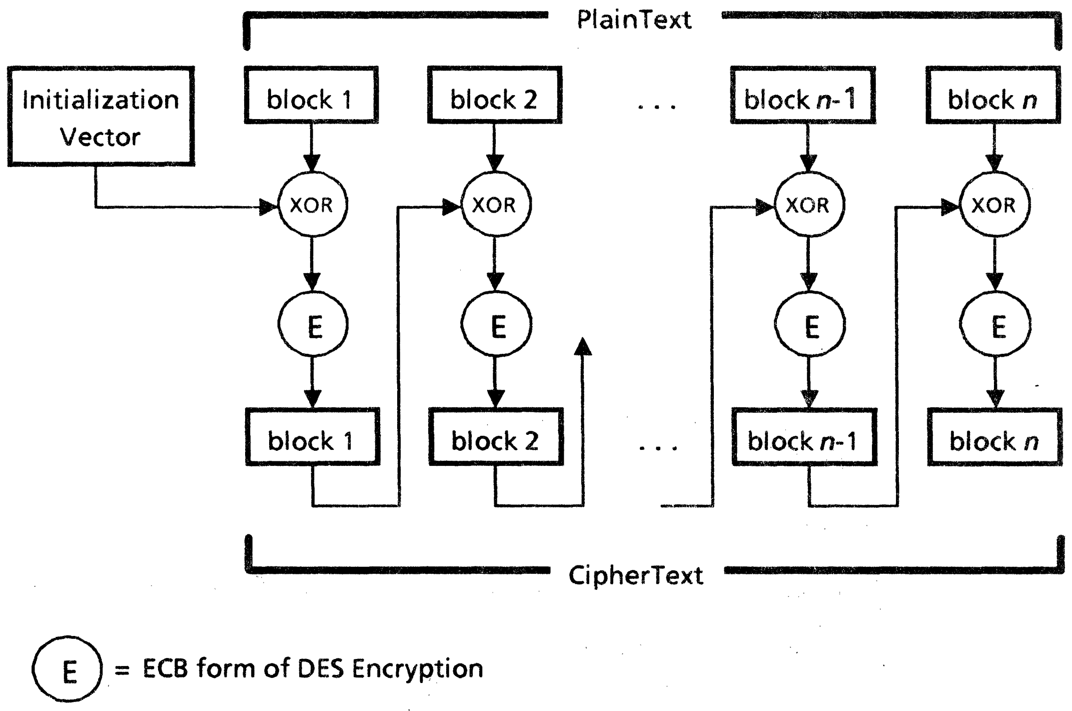


Figure 5.1 Standard cipher block chaining

The Authentication Protocol uses a variant of the standard cipher block chaining mode in which the XOR sum of plain text blocks 1 through n-1 is XORed with plain text block n

before encryption. This makes it possible to detect tampering with the cipher text. This mode of DES, called cipher block chaining with checksum (CBCC), is shown in Figure 5.2.

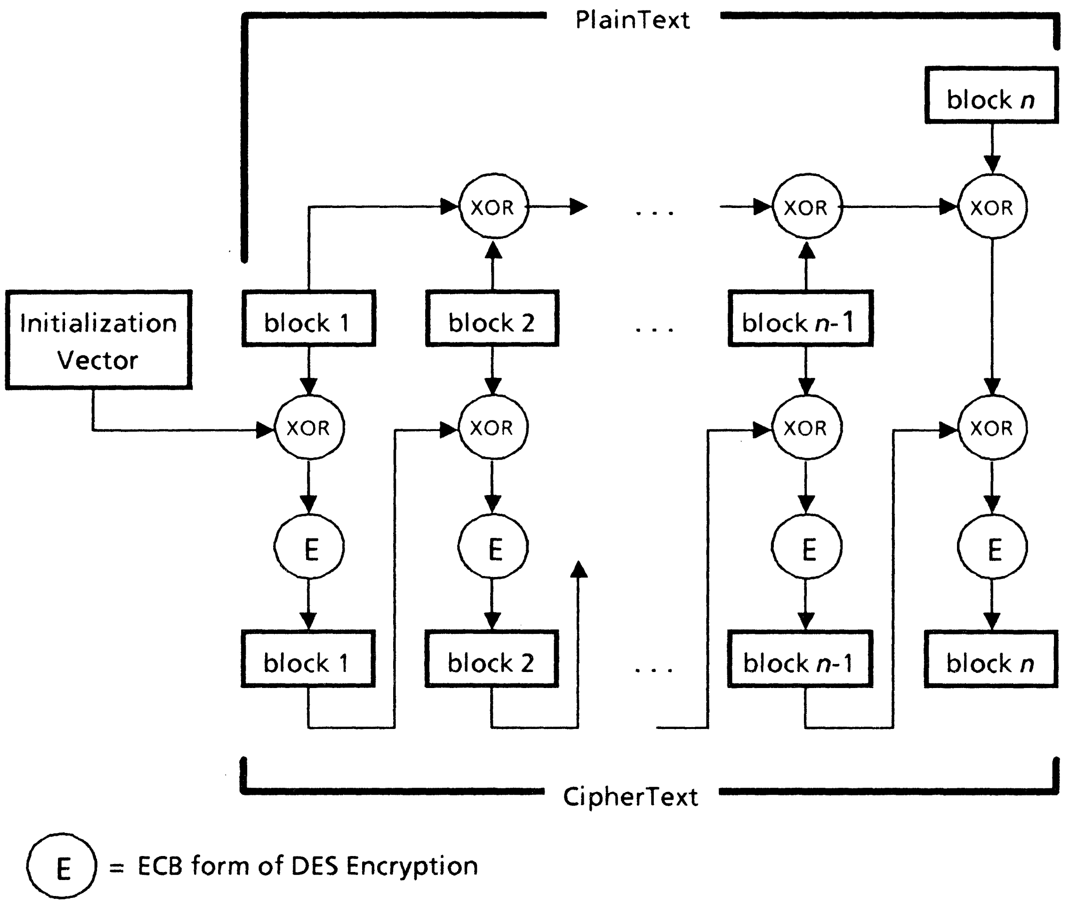


Figure 5.2 Cipher block chaining with checksum

5.3 Algorithm for computing a strong key

As mentioned in Chapter 2, a user's strong password is converted by his system element to a strong key. So that the user may use this same password in authenticated conversations initiated at various devices, it is convenient if all these devices use the same algorithm to manufacture the strong key. The algorithm is described below.

A strong password may contain any character in the Xerox Character Encoding Standard [5] except those in the Xerox Rendering Code Standard [10]. A DES encryption **Key** is computed from the password in the following fashion: [Note: steps a) and b) describe exactly the same transformation as steps a) and b) in the hashing algorithm defined in 5.1.]

a) the following character substitution is made in the password: if the character is an upper-case ISO/ASCII character in the range "A" through "Z" in character set 0 of the Encoding Standard, it is replaced with the corresponding lower-case character; otherwise the character is unchanged;

b) every character is mapped to a 16-bit code as defined by the Encoding standard;

c) the resulting sequence of numbers is assembled into a series of 64-bit blocks [b_1, b_2, ..., b_{n-1}, b_n] as shown in Figure 5.3. If the number of characters in the password is not a

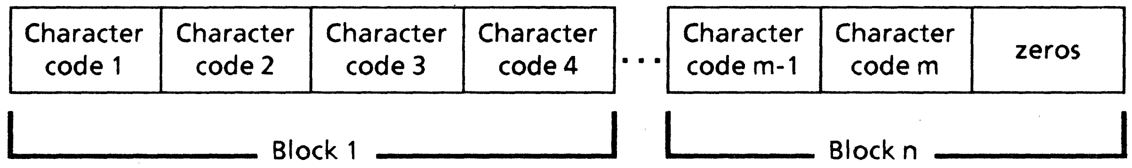


Figure 5.3 Decomposing a password into blocks

multiple of four, block b_n is filled with a sufficient number of zeros, placed to the right of the least-significant bit of the last character code, to bring the block length to 64 bits;

d) Let the function **E**[k, d] indicate encryption using the ECB mode of DES where k is the key and d is the data to be encrypted. Then the following algorithm is applied iteratively until $i = n$ (i.e., until all the blocks of the password have been processed):

$k_0 = 0$
$k_i = \mathbf{E}[k_{i-1}, b_i]$

e) The final output of this process, k_n, is parity adjusted by viewing it as a series of octets and setting the least-significant bit of each octet to the appropriate value to make each octet have odd parity.

It is recommended that passwords contain at least 4 characters.

Example:

If the password is "Cat," the corresponding value for a **Key** is (expressed in octets):

Key["Cat"] = Key["cat"]
= **SetOddParity**[**E**[0, (["c"]*2^{48} + ["a"]*2^{32} + ["t"]*2^{16} + 0)]]
= **SetOddParity**[**E**[0, (99*2^{48} + 97*2^{32} + 116*2^{16} + 0)]]
= **SetOddParity**[[142_8, 336_8, 332_8, 314_8, 173_8, 224_8, 231_8, 40_8]]
= [142_8, 337_8, 332_8, 315_8, 172_8, 224_8, 230_8, 40_8]

5.4 Algorithm for computing returned verifier

Some protocols require that a verifier be returned with the results of a remote operation. In order for the initiator to validate the results it is necessary that it know how the returned strong verifier is computed. Assume the decrypted value of the verifier sent to the recipient is as follows: **timeStamp** = t, and **ticks** = k. The value the recipient should send back is computed as follows:

if $k \geq 2^{32}-1$ then set **timeStamp** $\leftarrow$ t + 1, and **ticks** $\leftarrow$ 0
if $k < 2^{32}-1$ then set **timeStamp** $\leftarrow$ t, and **ticks** $\leftarrow$ k + 1

Then, 16 binary zeros are appended to the recipient's processor ID (after the least significant bit), and this number is XORed with the verifier record to produce a value that will be encrypted.

Appendix A References

The following documents supplement this protocol specification.

[1] National Bureau of Standards. The Data Encryption Standard. Federal Information Processing Standards Publication (FIPS PUB) 46; January 1977; National Technical Information Service, Springfield, VA.
This reference defines the basic DES encryption algorithm.

[2] National Bureau of Standards. DES Modes of Operation. Federal Information Processing Standards Publication (FIPS PUB) 81; 1980; National Technical Information Service, Springfield, VA.
This reference discusses the various modes of using the DES encryption algorithm.

[3] National Bureau of Standards. Guidelines for Implementing and Using the NBS Data Encryption Standard. Federal Information Processing Standards Publication (FIPS PUB) 74; 1981 April; National Technical Information Service, Springfield, VA.
This reference provides guidelines for using the DES encryption algorithm.

[4] Needham, R. A.; Schroeder, M. D. Using Encryption for Authenticating in Large Networks of Computers. Communications of the Association of Computing Machinery 21, 12; 1978 December; pp. 995-999.
This paper is the original design specification for the Authentication Protocol.

[5] Xerox Corporation. Character Encoding Standard. Xerox System Integration Standard. Stamford, Connecticut; 1983 May; XSIS 058305.
This reference defines the character set used for computing hashed passwords and strong keys.

[6] Xerox Corporation. Clearinghouse Entry Standard. Xerox System Integration Standard. Stamford, Connecticut. In preparation.
This reference defines the structure of the property values for names. In particular, it defines the property which determines whether a name represents a privileged person.

[7] Xerox Corporation. Clearinghouse Protocol. Xerox System Integration Standard. Stamford, Connecticut; 1983 August; XSIS 078308.
This reference defines the structure of user names which are used for initiator, recipient, and key owner names.

[8] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Xerox System Integration Standard. Stamford, Connecticut; 1981 December; XSIS 038112.
This reference defines the Courier language, in terms of which the Authentication Protocol is defined.

[9] Xerox Corporation. Internet Transport Protocols. Xerox System Integration Standard. Stamford, Connecticut; 1981 December; XSIS 028112.
This reference defines the Packet Exchange Protocol which is used to locate an Authentication server.

[10] Xerox Corporation. Rendering Code Standard. Xerox System Integration Standard. Stamford, Connecticut. In preparation.
Along with [5], this reference defines the character set used for passwords.

[11] Xerox Corporation. Time Protocol. Xerox System Integration Standard. Stamford, Connecticut; 1982 October; XSIS 088210.
This reference defines the Time Standard upon which the Authentication Protocol relies for the definition of the format of time quantities.

B

Appendix B
Program declaration

The complete declaration of the Authentication Protocol is given below.

```
Authentication: PROGRAM 14 VERSION 2 =
BEGIN
    DEPENDS UPON
        Clearinghouse (2) VERSION 2,
        Time (15) VERSION 2;

-- TYPES SUPPORTING ENCODING --

Key: TYPE = ARRAY 4 OF UNSPECIFIED; -- least significant bit of each octet is an odd parity bit --

Block: TYPE = ARRAY 4 OF UNSPECIFIED; -- cipher text or plain text block --

HashedPassword: TYPE = CARDINAL;

-- TYPES DESCRIBING CREDENTIALS AND VERIFIERS --

CredentialsType: TYPE = {simple(0), strong (1)};

 -- To retain compatibility with the previous version of the Authentication Protocol, the
following variable is defined. --
simpleCredentials: CredentialsType = simple;

Credentials: TYPE = RECORD [type: CredentialsType, value: SEQUENCE OF UNSPECIFIED];

CredentialsPackage: TYPE = RECORD[
    credentials: Credentials,
    nonce: LONG CARDINAL,
    recipient: Clearinghouse.Name,
    conversationKey: Key];

-- Instances of the following type must be a multiple of 64 bits, padded with zeros --
StrongCredentials: TYPE = RECORD [
    conversationKey: Key,
    expirationTime: Time.Time,
    initiator: Clearinghouse.Name];
```

```
SimpleCredentials: TYPE = Clearinghouse.Name;

Verifier: TYPE = SEQUENCE 12 OF UNSPECIFIED;

StrongVerifier: TYPE = RECORD [timeStamp: Time.Time, ticks: LONG CARDINAL];

SimpleVerifier: TYPE = HashedPassword;

-- PROCEDURES --

-- Strong Authentication --

GetStrongCredentials: PROCEDURE [initiator, recipient: Clearinghouse.Name,
    nonce: LONG CARDINAL]
RETURNS [credentialsPackage: SEQUENCE OF UNSPECIFIED]
REPORTS [CallError] = 1;

CreateStrongKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name, key: Block]
REPORTS [AuthenticationError, CallError] = 3;

ChangeStrongKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    newKey: Block]
REPORTS [AuthenticationError, CallError] = 4;

DeleteStrongKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name]
REPORTS [AuthenticationError, CallError] = 5;

-- Simple Authentication --

CheckSimpleCredentials: PROCEDURE [credentials: Credentials, verifier: Verifier]
RETURNS [ok: BOOLEAN]
REPORTS [AuthenticationError, CallError] = 2;

CreateSimpleKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name, key: HashedPassword]
REPORTS [AuthenticationError, CallError] = 6;

ChangeSimpleKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    newKey: HashedPassword]
REPORTS [AuthenticationError, CallError] = 7;

DeleteSimpleKey: PROCEDURE [credentials: Credentials, verifier: Verifier,
    name: Clearinghouse.Name]
REPORTS [AuthenticationError, CallError] = 8;

-- ERRORS --

AuthenticationError: ERROR [problem: Problem] = 2;
Problem: TYPE = {
    credentialsInvalid(0), -- decryption failed --
    verifierInvalid(1), -- decryption failed --
    verifierExpired(2), -- the verifier was too old --
    verifierReused(3), -- the verifier has been used before --
    credentialsExpired(4), -- the credentials have expired --
    inappropriateCredentials(5)}; -- need other kind of credentials to talk to recipient --

CallError: ERROR [problem: CallProblem, whichArg: Which] = 1;
Which: TYPE = {notApplicable(0), initiator(1), recipient(2), client(3)};
CallProblem: TYPE = {
    tooBusy(0), -- server is too busy to service this request --
    accessRightsInsufficient(1), -- operation prevented by access controls --
    keysUnavailable(2), -- the server which holds the required key was inaccessible --
    strongKeyDoesNotExist(3), -- a strong key critical to this operation has not been
        registered --
    simpleKeyDoesNotExist(4), -- a simple key critical to this operation has not been
        registered --
    strongKeyAlreadyRegistered(5), -- cannot create a strong key for an entity which
        already has one --
    simpleKeyAlreadyRegistered(6), -- cannot create a simple key for an entity which
        already has one --
    domainForNewKeyUnavailable(7), -- cannot create a new key because the domain to
        hold it is inaccessible --
    domainForNewKeyUnknown(8), -- cannot create a new key because the domain to
        hold it is unknown --
    badKey(9), -- bad key passed to CreateStrongKey or ChangeStrongKey --
    badName(10), -- bad name passed to CreateStrongKey or ChangeStrongKey --
    databaseFull(11), -- no more data can be added to the Authentication database --
    other(12)};

END. -- of Authentication
```

C

Appendix C
Well-known service names

It is occasionally necessary to authenticate oneself to a distributed service, such as the Clearinghouse or Authentication Service, where the specific instance of the service is not important. To avoid having to give each instance of the distributed service a different name (necessary for authentication), the distributed services are given "well-known" names that can be used to reference any instance of that service. The following well-known names are defined for the corresponding distributed services:

Authentication:	Authentication Service: CHServers:CHServers
Clearinghouse:	Clearinghouse Service: CHServers:CHServers
Mail:	Mail Service: CHServers:CHServers

D

Appendix D Example

To show how the Authentication Protocol may be employed, an extended example is given below.

Suppose a client whose user's name is *George:Xerox:Xerox*, wishes to authenticate himself to a File Service whose name is *Manila:Xerox:Xerox* in order to be granted access to a particular file. First, the client would locate an Authentication server. This is done exactly the same as is described for locating a Clearinghouse in the Clearinghouse Protocol (Appendix E) [7]. Next, the client would call the Authentication Service to get some strong credentials. He supplies a random number (63749) as the nonce parameter.

GetStrongCredentials[initiator: ["Xerox" "Xerox" "George"],
recipient: ["Xerox" "Xerox" "Manila"], nonce: 63749]

The Authentication Service responds with a set of credentials and a conversation key:

RETURNS[credentials:[type: strong, value: [*... encrypted StrongCredentials, nonce, Manila:Xerox:Xerox, and conversation key ...* **]]]**

The client can decrypt the **credentials.value** using George's strong key. Using the decrypted key, the client then encrypts a verifier. Suppose the current time is 479 and the client elects to set the ticks to 10. Then the record:

strongVerifier: [timeStamp: 479, ticks: 10]

would be XORed with the processor ID of *Manila:Xerox:Xerox*, padded with zeros, and encrypted using the decrypted conversation key. These credentials would then be passed on to the File Service (using the Filing Protocol):

Logon[credentials: [*... encrypted StrongCredentials ...* **],**
verifier: [*... encrypted [timeStamp: 479, ticks: 10] ...* **]]**

The File Service first checks to see that the credentials and verifier are well formed; that is, that they are a multiple of 64 bits in length. Then the File Service uses its strong key to decrypt the credentials which would produce a record containing the conversation key, the time the credentials expire, and the name of the initiator:

[conversationKey: ... *key* ...**,**
    expirationTime: ... *hours later than 479* ...**,**
    initiator: ["Xerox" "Xerox" "George"]]

The File Service checks that the credentials are well-formed: it is a multiple of 64 bits in length, the padding bits are zero, and the various fields are well-formed. Then it confirms that the current time does not exceed the expiration time, and that the conversation key is well-formed: the parity bits are correct. Then the verifier is decrypted using the conversation key. Since this is the first call from George, the time and ticks in the verifier are recorded for the next interaction with the client. At this point the File Service is fairly certain that it is communicating with *George:Xerox:Xerox*. It can use the name to check its access list to see if George is allowed to access this service.

The File Service computes the verifier to be returned; in this case it would be:

returnedVerifier: [timeStamp: 479, ticks: 11]

It then XORs it with its own processor ID padded with zeros and encrypts this record with the conversation key and returns the results of the logon, namely:

RETURNS[session: [token: [41B, 3B],
    verifier: [... *encrypted [timeStamp: 479, ticks: 11]* ...**]]]**

George now decrypts **verifier** using the conversation key and checks that **ticks** is one larger than was sent. George is now reasonably sure that he is communicating with the File Service *Manila:Xerox:Xerox*.

The next procedure call that George makes on the File Service, the File Service will check to make sure that the new verifier has a time or ticks larger than the previous one.

XEROX

XEROX

Authentication Protocol

Xerox System Integration Standard

XSIS 098404

610P72586