Internet Draft

Josh Benaloh
Butler Lampson
Daniel Simon
Terence Spies
Bennet Yee
Microsoft Corp.
October 1995

The Private Communication Technology Protocol

<draft-benaloh-pct-00.txt>

1. Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ds.internic.net (US East Coast), nic.nordu.net (Europe), ftp.isi.edu (US West Coast), or munnari.oz.au (Pacific Rim).

This Internet-Draft expires 27 March 1996.

2. Abstract

This document specifies Version 1 of the Private Communication Technology (PCT) protocol, a security protocol that provides privacy over the Internet. Like SSL (see [1]), the protocol is intended to prevent eavesdropping on communications in client/server applications, with servers always being authenticated and clients being authenticated at the server's option. However, this protocol corrects or improves on several weaknesses of SSL.

3. Introduction

The Private Communication Technology (PCT) Protocol is designed to provide privacy between two communicating applications (a client and a server), and to authenticate the server and (optionally) the client. PCT assumes a reliable transport protocol (e.g. TCP) for data transmission and reception.

The PCT Protocol is application protocol-independent. A "higher level" application protocol (e.g. HTTP, FTP, TELNET, etc.) can layer on top of the PCT Protocol transparently. The PCT Protocol begins with a handshake phase that negotiates an encryption algorithm and (symmetric) session key as well as authenticating a server to the client (and, optionally, vice versa), based on certified asymmetric public keys. Once transmission of application protocol data begins, all data is encrypted using the session key negotiated during the handshake.

It should be noted that the PCT protocol does not specify any details about verification of certificates with respect to certification authorities, revocation lists, and so on. Rather, it is assumed that protocol implementations have access to a "black box" which is capable of ruling on the validity of received certificates in a manner satisfactory to the implementation's user. Such a ruling may, for instance, involve remote consultation with a trusted service, or even with the actual user through a text or graphic interface.

In addition to encryption and authentication, the PCT protocol verifies the integrity of messages using a hash function-based message authentication code (MAC).

The PCT protocol's record format is compatible with that of SSL (see [1]). Servers implementing both protocols can distinguish between PCT and SSL clients because the version number field occurs in the same position in the first handshake message in both protocols. In PCT, the most significant bit of the protocol version number is set to one.

The PCT protocol differs from SSL chiefly in the design of its handshake phase, which differs from SSL's in a number of respects:

- The round and message structures are considerably shorter and simpler: a reconnected session without client authentication requires only one message in each direction, and no other type of connection requires more than two messages in each direction.

- Negotiation for the choice of cryptographic algorithms and formats to use in a session has been extended to cover more protocol characteristics and to allow different characteristics to be negotiated independently. The PCT client and server negotiate, in addition to a cipher type and server certificate type, a hash function type and a key exchange type. If client authentication is requested, a client certificate type and signature type are also negotiated.

- Message authentication has been revamped so that it now uses different keys from the encryption keys. Thus, message authentication keys may be much longer (and message authentication therefore much more secure) than the encryption keys, which may be weak or even non-existent.

- A security hole in SSL's client authentication has been repaired; the PCT client's authentication challenge response now depends on the type of cipher negotiated for the session. SSL's client authentication is independent of the cipher strength used in the session and also of whether the authentication is being performed for a reconnection of an old session or for a new one. As a result, a "man-in-the-middle" attacker who has obtained the session key for a session using weak cryptography can use this broken session to authenticate as the client in a session using strong cryptography. If, for example, the server normally restricts certain sensitive functions to high-security sessions, then this security hole allows intruders to circumvent the restriction.

- A "verify-prelude" field has been added to the handshake phase, with which the client and server can check that the cipher type (and other) negotiations carried out in the clear have not been tampered with. (SSL version 3 uses a similar mechanism, but its complete version 2 compatibility negates its security function, by allowing adversaries simply to alter version numbers as well as cipher types.)

4. PCT Record Protocol Specification

4.1 PCT Record Header Format

For compatibility with SSL, the PCT protocol uses the same basic record format as SSL. In PCT, all data sent is encapsulated in a record, an object which is composed of a header and some non-zero amount of data. Each record header contains a two- or three-byte length code. If the most significant bit is set in the first byte of the record length code then the record has no padding and the total header length is two bytes; otherwise the record has padding and the total header length is three bytes. The record header is transmitted before the data portion of the record.

Note that in the long header case (three bytes total), the second most significant bit in the first byte has special meaning. When zero, the record being sent is a data record. When one, the record being sent is a security escape, and the first byte of the record is an ESCAPE_TYPE_CODE that indicates the type of escape. (Currently, the two examples of security escapes are the "out-of-band data" escape and the "redo handshake" escape.) In either case, the length code describes how much data is in the record as transmitted; this may be greater than the amount of data after decryption, particularly if padding was used.

The record length code does not include the number of bytes consumed by the record header (two or three). For the two-byte header, the record length is computed by (using a "C"-like notation):

```
RECORD_LENGTH = ((byte[0] & 0x7f) << 8)) | byte[1],
```

where byte[0] represents the first byte received and byte[1] the second byte received. When the three-byte header is used, the record length is computed as follows (using a "C"-like notation):

```
RECORD_LENGTH = ((byte[0] & 0x3f) << 8)) | byte[1];
IS_ESCAPE = (byte[0] & 0x40) != 0;
PADDING_LENGTH = byte[2];
```

The record header defines a value called PADDING_LENGTH. The PADDING_LENGTH value specifies how many bytes of data were appended to the original record by the sender. The padding data is used to make the record length a multiple of the block cipher's block size when a block cipher is used for encryption.

The sender of a "padded" record appends the padding data to the end of its normal data and then encrypts the total amount (which is now a multiple of the block cipher's block size). The actual value of the padding data is unimportant, but the encrypted form of it must be transmitted for the receiver to decrypt the record properly. Once the total amount being transmitted is known the header can be properly constructed with the PADDING value set appropriately.

The receiver of a padded record uses the PADDING_LENGTH value from the header when determining the length of the ACTUAL_DATA in the data record (see section 4.2).

4.2 PCT Record Data Format

The format of the data portion of an encrypted PCT record is slightly different from that of SSL. However, not only is the record header format identical in both protocols, but the first handshake message sent in each direction is also sent in the clear, and contains a version number field in the same location in both protocols. (In PCT protocol handshake messages, the most significant bit of this field is 1.) Hence, PCT is still compatible with SSL, in the sense that PCT handshake messages can be recognized and distinguished from SSL handshake messages by examination of the version number.

The PCT record is composed of two fields (transmitted and received in the order shown):

```
ENCRYPTED_DATA[N+PADDING_LENGTH]
MAC_DATA[MAC_LENGTH]
```

The ENCRYPTED_DATA field contains an encryption of the concatenation of ACTUAL_DATA (the N bytes of actual data being transmitted) and PADDING_DATA (the use of which is explained below). The MAC_DATA field contains the "Message Authentication Code" (MAC).

PCT handshake records are sent in the clear, and no MAC is used. Consequently PADDING_LENGTH will be zero and MAC_LENGTH will be zero. For non-handshake data records, the sender appends a PADDING_DATA field containing arbitrary data, so that N + PADDING_LENGTH is the appropriate length for the cipher being used to encrypt the data. (In the case where a block cipher is used, PADDING_LENGTH must be the minimum value such that the length of the concatenation of ACTUAL_DATA and PADDING_DATA is an exact multiple of the cipher's block size. Otherwise, PADDING_LENGTH is zero.)

MAC_DATA is computed as follows:

MAC_DATA := Hash(MAC_KEY, Hash(ACTUAL_DATA, PADDING_DATA, SEQUENCE_NUMBER))

If the client is sending the record, then the MAC_KEY is the CLIENT_MAC_KEY; if the server is sending the record, then the MAC_KEY is the SERVER_MAC_KEY. (The details of the derivation of these keys are given in section 5.3.1.) The selection of the hash function used to compute MAC_DATA is discussed in sections 5.2.1 and 5.2.2. The parameters of the inner invocation of the hash function are input into the hash function in the order shown above, with SEQUENCE_NUMBER represented in network byte order or "big endian" order. If the length of MAC_KEY is not an exact multiple of eight bits, then MAC_KEY is considered, for the purposes of MAC computation, to have (fewer than eight) zero bits appended to it, to create a string of an integral number of bytes for input into the MAC hash function.

The SEQUENCE_NUMBER is a counter which is incremented by both the sender and the receiver. For each transmission direction, a pair of counters is kept (one by the sender, one by the receiver). Before the first (handshake) record is sent or received in a PCT connection all sequence numbers are initialized to zero (except in the case of a restarting connection with a token-based exchange type, in which case the entire cipher state is preserved; see section 5.2.2). The sender's sender-to-receiver sequence number is incremented after every record sent, and the receiver's sender-to-receiver sequence number is incremented after every record received. Sequence numbers are 32-bit unsigned quantities, and may not increment past 0xFFFFFFFF. (See section 4.3.2.)

The receiver of a record (whether PCT client or server) first uses the sender's WRITE_KEY to decrypt the concatenated ACTUAL_DATA and PADDING_DATA fields, then uses the sender's MAC_KEY, the ACTUAL_DATA, the PADDING_DATA, and the expected value of the sequence number as input into the MAC_DATA function described above (the hash function algorithm used is determined the same way for the receiver as for the sender). The computed MAC_DATA must agree bit for bit with the transmitted MAC_DATA. If the two are not identical, then an INTEGRITY_CHECK_FAILED error occurs, and it is recommended that the record be treated as though it contained no data. (See section 5.4.) The same error occurs if N + PADDING_LENGTH is not correct for the block cipher used.

The PCT Record Layer is used for all PCT communications, including handshake messages, security escapes and application data transfers. The PCT Record Layer is used by both the client and the server at all times.

For a two-byte header, the maximum record length is 32767 bytes. For a three-byte header, the maximum record length is 16383 bytes. The PCT Handshake Protocol messages are constrained to fit in a single PCT Record Protocol record. Application protocol messages are allowed to consume multiple PCT Record Protocol records.

4.3 Security Escapes

4.3.1 Out-Of-Band Data

PCT, like SSL, supports the transmission and reception of "out-of-band data". Out of band data is normally defined at the TCP/IP protocol level, but because of PCT's privacy enhancements and support for block ciphers, this becomes difficult to support.

To send out-of-band data, the sender sends an escape record whose body contains a single byte of data which is the ESCAPE_TYPE_CODE value PCT_ET_OOB_DATA. The record following the escape record will be interpreted as "out-of-band" data and will only be made available to the receiver through an unspecified mechanism that is different from the receiver's normal data reception method. The escape record and the transmitted data record are transmitted normally (i.e. encryption, MAC computations, and block cipher padding remain in effect).

Note that the escape record and the associated data record are sent using normal TCP sending mechanisms, not using the "out of band" mechanisms. Note also that a "Redo Handshake" escape record (see below) and its associated handshake messages may be interposed between an "Out-of-Band Data" escape record and its associated data record. In such a case, the first non-escape, non-handshake record following the "Out-of-Band Data" escape record is treated as out-of-band data.

4.3.2 Redo Handshake

PCT allows either the client or the server to request, at any time after the handshake phase has been completed for a connection, that another handshake phase be performed for that connection. For example, either party is required to request another handshake phase rather than allow its sequence number to "wrap" beyond 0xFFFFFFFF. In addition, it is recommended that implementations enforce limits on the duration of both connections and sessions, with respect to the total number of bytes sent, the number of records sent, the actual time elapsed since the beginning of the connection or session, and, in the case of sessions, the number of reconnections made. These limits serve to ensure that keys are not used more or longer than it is safe to do so; hence the limits may depend on the type and strength of cipher, key exchange and authentication used, and may, at the implementer's discretion, include indications from the application as to the sensitivity of the data being transmitted or received.

To request a new handshake phase for this connection, the sender (client or server) sends an escape record whose body contains a single byte of data which is the ESCAPE_TYPE_CODE value PCT_ET_REDO_CONN. The escape record is transmitted normally (i.e. encryption, MAC computations, and block cipher padding remain in effect).

There are several cases to consider to ensure that the message pipeline gets flushed and to enable handshake messages to be distinguished from data records. The following rules ensure that the first messages in the redone handshake are always immediately preceded by a "Redo Handshake" escape message.

If the client initiates the "Redo Handshake", it sends the "Redo Handshake" escape message immediately followed by a normal CLIENT_HELLO message; the server, on receiving the "Redo Handshake" escape message, may be in one of two states. If the last message it sent was a "Redo Handshake" escape message, then it simply waits for the CLIENT_HELLO message; otherwise, it sends a "Redo Handshake" escape message in response, and then waits for the CLIENT_HELLO message.

If the server initiates the "Redo Handshake", then the server sends the "Redo Handshake" escape message and simply waits for a "Redo Handshake" escape message in response; this "Redo Handshake" should be immediately followed by a normal CLIENT_HELLO message. The client, on receiving the server's "Redo Handshake" escape message, may be in one of two states. If the last two messages it sent were a "Redo Handshake" escape message followed by a CLIENT_HELLO message, then it simply waits for a SERVER_HELLO message; otherwise, it sends a "Redo Handshake" escape message in response, followed by a CLIENT_HELLO message, and then waits for a SERVER_HELLO message.

In all cases, the sender of the "Redo Handshake" escape message continues to process incoming messages, but may not send any non-handshake messages until the new handshake completes.

The handshake phase that follows the "Redo Handshake" escape message is a normal one in most respects; the client may request the reconnection of an old session or request that a new session be initiated, and the server, on receiving a reconnection request, can accept the reconnection or demand that a new session be initiated instead. If a new session is being established, then the server must request client authentication if and only if client authentication was requested during the previous session. Otherwise, client authentication is optional. Both parties must verify that the specifications negotiated previously in the session (cipher type, key exchange type, certificate type, hash function type, client certificate type, and signature type), as well as any certificates exchanged, are identical to those found in the new handshake phase.

A mismatch results in a SPECS_MISMATCH or BAD_CERTIFICATE error (see section 5.4.) This ensures that the security properties of the communication channel do not change.

5. PCT Handshake Protocol Specification

5.1 PCT Handshake Protocol Flow

The PCT Handshake Protocol is used to negotiate security enhancements to data sent using the PCT Record Protocol. The security enhancements consist of authentication, symmetric encryption, and message integrity. Symmetric encryption is facilitated using a "Key Exchange Algorithm". PCT version 1 supports RSA {TM} -based key exchange (see [13]), Diffie-Hellman key exchange, and FORTEZZA token key exchange.

The PCT Handshake Protocol consists of four messages, sent respectively by the client, then server, then client, then server, in that order. (Moreover, under certain circumstances, the last two messages are omitted.) The messages are named, in order, CLIENT_HELLO, SERVER_HELLO, CLIENT_MASTER_KEY, and SERVER_VERIFY.

The general contents of the messages depend upon two criteria: whether the connection being made is a reconnection (a continuation of a previous session) or a new session and whether the client is to be authenticated. (The server is always authenticated.) The first criterion is determined by the client and server together; the CLIENT_HELLO will have different contents depending on whether a new session is being initiated or an old one continued, and the SERVER_HELLO message will either confirm a requested continuation of an old session, or require that a new session be initiated. The second criterion is determined by the server, whose SERVER_HELLO may contain a demand for authentication of the client. If the server does not require client authentication, and the reconnection of an old session is requested by the client and accepted by the server, then the CLIENT_MASTER_KEY and SERVER_VERIFY messages are unnecessary, and are omitted.

The CLIENT_HELLO message contains a random authentication challenge to the server and a request for the type and level of cryptography and certification to be used for the session. If the client is attempting to continue an old session, then it also supplies that session's ID.

In the case of a new session, the SERVER_HELLO message contains a certificate and a random connection identifier; this identifier doubles as an authentication challenge to the client if the server desires client authentication. The CLIENT_MASTER_KEY message sent by the client in response includes the master key for the session (from which session keys are derived), encrypted using the public key from the server's certificate, as well as a certificate and response to the server's authentication challenge, if requested. To ensure that previous unencrypted handshake messages were not tampered with, their keyed hash is included with the CLIENT_MASTER_KEY message. Finally, the server sends a SERVER_VERIFY message which includes a response to the client's challenge and a random session id for the session.

If the server accepts the old session id, then the SERVER_HELLO message contains a response to the client's challenge, and a random connection identifier which again doubles as a random challenge to the client, if the server requires client authentication. If no client authentication is requested, the handshake is finished (although an authentication of the client is implicit in the MAC included with the client's first data message). Otherwise, the subsequent CLIENT_MASTER_KEY message contains the client's response, and the SERVER_VERIFY message simply signals to the client to continue.

For a new session, the handshake phase has the following form (items in square brackets are included only if client authentication is required):

```
Client                                  Server
------                                  ------

CLIENT_HELLO:
client challenge;
client's cipher, hash,
    server-certificate,
    and key-exchange
    specification lists

                                        SERVER_HELLO:
                                        connection id/server challenge;
                                        server's cipher, hash,
                                            server-certificate,
                                            and key-exchange
                                            specification choices;
                                        server certificate
                                        [; server's signature-type
                                            and client-certificate
                                            specification lists]

CLIENT_MASTER_KEY:
master key, encrypted with
    server's public key;
authentication of previous two
    messages
[; client's signature-type and
    client-certificate
    specification choices;
client's certificate;
client's challenge response]

                                        SERVER_VERIFY:
                                        session id;
```

server's challenge response

For a reconnection of an old session, the handshake phase has the following form (items in square brackets are included if client authentication is required):

```
Client                                  Server
------                                  ------

CLIENT_HELLO:
client challenge;
session id;
client's cipher, hash,
    server-certificate,
    and key-exchange
    specification lists

                                        SERVER_HELLO:
                                        connection id/server challenge;
                                        old session's cipher, hash,
                                            server-certificate,
                                            and key-exchange
                                            specification choices;
                                        server's challenge response
                                        [; server's signature-type
                                            and client-certificate
                                            specification lists]

[CLIENT_MASTER_KEY:
client's certificate;
client's challenge response]


                                        [SERVER_VERIFY]
```

Note that the protocol is asymmetric between client and server. The client authenticates the server because only the server can decrypt the master key which is encrypted with the server's public key, and the server's challenge response depends on knowing that master key. The server authenticates the client because the client signs its challenge response with its public key. The reason for the asymmetry is that when there is no client authentication there is no client public key, so the client must choose the master key and encrypt it with the server public key to hide it from everyone except the server.

Usually the client can safely send data on the underlying transport immediately following the CLIENT_MASTER_KEY message, without waiting for the SERVER_VERIFY; we call this "initial data". Sending initial data is good because it means that PCT adds only one round trip; it is not possible to do better without exposing the server to a replay attack. However, it is unwise to send initial data if for some reason it is important for the client to be sure of being in contact with the correct server before sending any data.

5.2 PCT Handshake Protocol Messages

The PCT Handshake Protocol messages are sent using the PCT Record Protocol and consist of a single byte message type code, followed by some data. The client and server exchange messages as described above, sending either one or two messages each. Once the handshake has been completed successfully, the client sends its first actual data.

Handshake protocol messages are sent in the clear, with the exception of the key-exchange-related fields in the CLIENT_MASTER_KEY message, some of which involve (public-key) encryption.

The following notation is used for PCT messages:

```
char MSG_EXAMPLE
char FIELD1
char FIELD2
char THING_LENGTH_MSB
char THING_LENGTH_LSB
char THING_DATA[(MSB << 8)|LSB];
...
```

This notation defines the data in the protocol message, including the message type code. The order is presented top to bottom, with the topmost element being transmitted first.

For the "THING_DATA" entry, the MSB and LSB values are actually THING_LENGTH_MSB and THING_LENGTH_LSB (respectively) and define the number of bytes of data actually present in the message. For example, if THING_LENGTH_MSB were one and THING_LENGTH_LSB were four then the THING_DATA array would be exactly 260 bytes long. This shorthand is used below. Occasionally, a "THING_DATA" field is referred to as "THING", with the word "DATA" omitted.

The names of message elements have prefixes that identify the messages in which they appear; these prefixes are sometimes omitted in the text when the containing messages are obvious.

Length codes are unsigned values, and when the MSB and LSB are combined the result is an unsigned value. Length values are in bytes.

5.2.1 CLIENT_HELLO (Sent in the clear)

```
char CH_MSG_CLIENT_HELLO
char CH_CLIENT_VERSION_MSB
char CH_CLIENT_VERSION_LSB
char CH_PAD
```

```
char CH_SESSION_ID_DATA[32]
char CH_CHALLENGE_DATA[32]
char CH_OFFSET_MSB
```

```
char CH_OFFSET_LSB
char CH_CIPHER_SPECS_LENGTH_MSB
char CH_CIPHER_SPECS_LENGTH_LSB
char CH_HASH_SPECS_LENGTH_MSB
char CH_HASH_SPECS_LENGTH_LSB
char CH_CERT_SPECS_LENGTH_MSB
char CH_CERT_SPECS_LENGTH_LSB
char CH_EXCH_SPECS_LENGTH_MSB
char CH_EXCH_SPECS_LENGTH_LSB
char CH_KEY_ARG_LENGTH_MSB
char CH_KEY_ARG_LENGTH_LSB
char CH_CIPHER_SPECS_DATA[(MSB << 8)|LSB]
char CH_HASH_SPECS_DATA[(MSB << 8)|LSB]
char CH_CERT_SPECS_DATA[(MSB << 8)|LSB]
char CH_EXCH_SPECS_DATA[(MSB << 8)|LSB]
char CH_KEY_ARG_DATA[(MSB << 8)|LSB]
```

When a client first connects to a server it is required to send the CLIENT_HELLO message. The server is expecting this message from the client as its first message. It is an ILLEGAL_MESSAGE error for a client to send anything else as its first message. The CLIENT_HELLO message begins with the PCT version number, and two fixed-length fields followed by an offset to the variable length data. The CH_OFFSET field contains the number of bytes used by the various fields (currently only length fields) that follow the offset field and precede the variable-length fields. For PCT version 1, this offset value is always PCT_CH_OFFSET_V1, i.e., ten. However, inclusion of this field will allow future versions to be compatible with version 1, even if the number of these fields changes: a version 1 server should be able to find all the PCT version 1 fields in a higher-version CLIENT_HELLO message. The CH_PAD field may contain any value.

The CLIENT_HELLO message includes a string of random bytes used as challenge data from the client. Also, if the client finds a session identifier in its cache for the server, then that session-identifier data is sent. Otherwise, the special PCT_SESSION_ID_NONE value is used. In either case, the client specifies in CIPHER_SPECS_DATA, HASH_SPECS_DATA, CERT_SPECS_DATA, and EXCH_SPECS_DATA its preferred choices of symmetric cipher, key lengths, hash function, certificate, and asymmetric key exchange algorithm. However, if a session identifier is sent, then these choices are only relevant in the case where the server cannot recognize the session identifier, and a new session must therefore be initiated. If the server recognizes the session, then these fields are ignored by the server.

The CHALLENGE_DATA field contains 32 bytes of random bits, to be used to authenticate the server. The CHALLENGE_DATA should be cryptographically random, in the same sense as the MASTER_KEY (see section 5.3.1).

The CIPHER_SPECS_DATA field contains a list of possible symmetric ciphers supported by the client, in order of (the client's) preference. Each element in the list is a four-byte field, of which the first two bytes contain a code representing a cipher type, the third byte contains the encryption key length in bits (0-255), and the fourth byte contains the MAC key length in bits, minus 64 (values 0-255, representing lengths 64-319; this encoding enforces the requirement that the MAC key length be at least 64 bits). The entire list's length in bytes (four times the number of elements) is placed in CIPHER_SPECS_LENGTH.

The HASH_SPECS_DATA field contains a list of possible hash functions supported by the client, in order of (the client's) preference. The server will choose one of these to be used for computing MACs and deriving keys. Each element in the list is a two-byte field containing a code representing a hash function choice. The entire length of the list (twice the number of elements) is placed in HASH_SPECS_LENGTH.

The CERT_SPECS_DATA field contains a list of possible certificate formats supported by the client, in order of (the client's) preference. Each element in the list is a two-byte field containing a code representing a certificate format. The entire length of the list (twice the number of elements) is placed in CERT_SPECS_LENGTH.

The EXCH_SPECS_DATA field contains a list of possible asymmetric key exchange algorithms supported by the client, in order of (the client's) preference. Each element in the list is a two-byte field containing a code representing a key exchange algorithm type. The entire length of the list (twice the number of elements) is placed in EXCH_SPECS_LENGTH.

The KEY_ARG_DATA field contains an initialization vector to be used in a reconnected session when the cipher type is a block cipher (any cipher type except PCT_CIPHER_RC4, and any key exchange type except PCT_EXCH_RSA_PKCS1_TOKEN_RC4). If a new session is being requested (i.e., the value of CH_SESSION_ID_DATA is PCT_SESSION_ID_NONE), then KEY_ARG_LENGTH must be zero.

The CLIENT_HELLO message must be the first message sent by the client to the server. After the message is sent the client waits for a SERVER_HELLO message. Any other message returned by the server (other than ERROR) generates the PCT_ERR_ILLEGAL_MESSAGE error.

The server, on receiving a CLIENT_HELLO message, checks the version number and the offset field to determine where the variable-length data fields start. (The OFFSET value should be at least PCT_CH_OFFSET_V1.) The server then checks whether there is a non-null SESSION_ID field, and if so, whether it recognizes the SESSION_ID. In that case, the server responds with a SERVER_HELLO message containing a non-zero RESTART_SESSION_OK field, and the appropriate value (see below) in the RESPONSE and CONNECTION_ID fields. Otherwise, it checks whether the CIPHER_SPECS, HASH_SPECS, CERT_SPECS and EXCH_SPECS lists in the CLIENT_HELLO message each contain at least one type supported by the server. If so, then the server sends a SERVER_HELLO message to the client as described below; otherwise, the server detects a SPECS_MISMATCH error.

5.2.2 SERVER_HELLO (Sent in the clear)

```
char SH_MSG_SERVER_HELLO
char SH_PAD
char SH_SERVER_VERSION_MSB
char SH_SERVER_VERSION_LSB
char SH_RESTART_SESSION_OK
char SH_CLIENT_AUTH_REQ
char SH_CIPHER_SPECS_DATA[4]
char SH_HASH_SPECS_DATA[2]
char SH_CERT_SPECS_DATA[2]
char SH_EXCH_SPECS_DATA[2]
char SH_CONNECTION_ID_DATA[32]
char SH_CERTIFICATE_LENGTH_MSB
char SH_CERTIFICATE_LENGTH_LSB
char SH_CLIENT_CERT_SPECS_LENGTH_MSB
char SH_CLIENT_CERT_SPECS_LENGTH_LSB
char SH_CLIENT_SIG_SPECS_LENGTH_MSB
char SH_CLIENT_SIG_SPECS_LENGTH_LSB
char SH_RESPONSE_LENGTH_MSB
char SH_RESPONSE_LENGTH_LSB
char SH_CERTIFICATE_DATA[(MSB << 8)|LSB]
char SH_CLIENT_CERT_SPECS_DATA[(MSB << 8)|LSB]
char SH_CLIENT_SIG_SPECS_DATA[(MSB << 8)|LSB]
char SH_RESPONSE_DATA[(MSB << 8)|LSB]
```

The server sends this message after receiving the client's CLIENT_HELLO message. The PCT version number in SH_SERVER_VERSION is always the maximum protocol version that the server supports; the remainder of the message and all subsequent messages will conform to the format specified by the protocol version corresponding to the minimum of the client and server protocol version numbers. Unless there is an error, the server always returns a random value 32 bytes in length in the CONNECTION_ID field. This value doubles as challenge data if the server requests client authentication, and should therefore be random in the same sense as the challenge data in the CLIENT_HELLO message. The SH_PAD field may contain any value.

There are two cases for RESTART_SESSION_OK. In the first case, the server returns a zero RESTART_SESSION_OK flag because the CLIENT_HELLO message did not contain a session id or because the one it contained is unrecognized by the server. In this case, the server must behave as follows:

The server selects any choice with which it is compatible, from each of the CH_CIPHER_SPECS, CH_HASH_SPECS, CH_CERT_SPECS and CH_EXCH_SPECS lists supplied in the CLIENT_HELLO message. (These values are returned to the client in the SH_CIPHER_SPECS_DATA,

SH_HASH_SPECS_DATA, SH_CERT_SPECS_DATA, and SH_EXCH_SPECS_DATA fields, respectively.) The certificate of the type specified in SH_CERT_SPECS_DATA and SH_EXCH_SPECS_DATA is placed in the CERTIFICATE_DATA field. The SH_RESPONSE_DATA field is empty, and its length is zero.

In the second case, the server returns a non-zero RESTART_SESSION_OK flag because the CLIENT_HELLO message contained a session-identifier known by the server (i.e. in the server's session-identifier cache). In this case, the server must behave as follows:

The server omits the CERTIFICATE_DATA field (with CERTIFICATE_LENGTH set to zero), and sets the CIPHER_SPECS_DATA, HASH_SPECS_DATA, CERT_SPECS_DATA and EXCH_SPECS_DATA values to the values stored along with the session identifier. There are two subcases: (1) If the SH_EXCH_SPECS_DATA does not refer to a TOKEN type, then the CLIENT_MAC, SERVER_MAC, CLIENT_WRITE, and SERVER_WRITE keys are rederived using the MASTER_KEY from the old session, as well as the CONNECTION_ID and CH_CHALLENGE values from the SERVER_HELLO and CLIENT_HELLO messages, respectively, for this connection. (2) If the SH_EXCH_SPECS_DATA refers to a TOKEN type, then the keys from the on-going session are reused. In order to obtain fresh key material or change the sequence number, TOKEN implementations must use the redo handshake mechanism (PCT_ET_REDO_CONN security escape). When this mechanism is used with a TOKEN exchange type, the client must send PCT_SESSION_ID_NONE in the CH_SESSION_ID_DATA field of the subsequent CLIENT_HELLO message.

The RESPONSE_DATA is constructed by computing the function

Hash(SERVER_MAC_KEY, Hash("sr", CH_CHALLENGE_DATA,
SH_CONNECTION_ID_DATA, CH_SESSION_ID_DATA)).

The CH_CHALLENGE_DATA and CH_SESSION_ID_DATA values are found in the CLIENT_HELLO message for this connection. The SH_CONNECTION_ID_DATA value is from this SERVER_HELLO message. The SERVER_MAC_KEY is the one rederived for this connection as described in section 5.3.1. If the length of SERVER_MAC_KEY is not an exact multiple of eight bits, then SERVER_MAC_KEY is considered, for the purposes of MAC computation, to have (fewer than eight) zero bits appended to it, to create a string of an integral number of bytes for input into the MAC hash function. The hash function choice used is determined by the SH_HASH_SPECS_DATA field in this SERVER_HELLO message. The values are input into the interior invocation of the hash function in the exact order specified above, with the string in quotation marks representing actual ASCII text.

In both reconnection cases, if the server requires client authentication, then the CLIENT_AUTH_REQ field is set to a non-zero value. Also, a list of (client) certificate types acceptable to the server, in order of (the server's) preference, is placed in the CLIENT_CERT_SPECS_DATA field, and a list of (client's) signature algorithms supported by the server, in order of (the server's) preference, is placed in the CLIENT_SIG_SPECS_DATA field. The certificate type values in the list are from the same set of two byte codes used for the CERT_SPECS list appearing in the CLIENT_HELLO message, and the signature algorithm type codes are also two bytes long. (See section 5.3.4 and 5.3.5 below.) The lengths of the lists in bytes (twice the number of elements) are placed in the CLIENT_CERT_SPECS_LENGTH and CLIENT_SIG_SPECS_LENGTH fields. If no client authentication is required, then these length fields, as well as the CLIENT_AUTH_REQ field, are set to zero, and the corresponding data fields are empty.

When the client receives a SERVER_HELLO message, it checks whether the server has accepted a reconnection of an old session or is establishing a new session. If a new session is being initiated, and client authentication is requested, then the client checks whether it is compatible with any of the certificate and signature types listed in the CLIENT_CERT_SPECS and CLIENT_SIG_SPECS lists. (Note that the server can make client authentication optional for the client simply by including PCT_CERT_NONE and PCT_SIG_NONE as a "last resort".) If the client can provide a compatible certificate, then it sends a CLIENT_MASTER_KEY message as described below; otherwise, it generates a SPECS_MISMATCH error.

If the session is an old one, then the client establishes the new CLIENT_WRITE_KEY, SERVER_WRITE_KEY, CLIENT_MAC_KEY and SERVER_MAC_KEY according to the cipher-specific rules described below in section 5.3.1. **The client then checks the contents of the RESPONSE_DATA field** in the SERVER_HELLO message for correctness. If the response matches the value calculated by the client (exactly as described above for the server), then the handshake is finished, and the client begins sending data; otherwise, a SERVER_AUTH_FAILED error occurs.

5.2.3 CLIENT_MASTER_KEY (sent in the clear, except for encrypted keys)

```
char CMK_MSG_CLIENT_MASTER_KEY
char CMK_PAD
char CMK_CLIENT_CERT_SPECS_DATA[2]
char CMK_CLIENT_SIG_SPECS_DATA[2]
char CMK_CLEAR_KEY_LENGTH_MSB
char CMK_CLEAR_KEY_LENGTH_LSB
char CMK_ENCRYPTED_KEY_LENGTH_MSB
char CMK_ENCRYPTED_KEY_LENGTH_LSB
char CMK_KEY_ARG_LENGTH_MSB
char CMK_KEY_ARG_LENGTH_LSB
char CMK_VERIFY_PRELUDE_LENGTH_MSB
char CMK_VERIFY_PRELUDE_LENGTH_LSB
```

```
char CMK_CLIENT_CERT_LENGTH_MSB
char CMK_CLIENT_CERT_LENGTH_LSB
char CMK_RESPONSE_LENGTH_MSB
char CMK_RESPONSE_LENGTH_LSB
```

```
char CMK_CLEAR_KEY_DATA[(MSB << 8)|LSB]
char CMK_ENCRYPTED_KEY_DATA[(MSB << 8)|LSB]
char CMK_KEY_ARG_DATA[(MSB << 8)|LSB]
char CMK_VERIFY_PRELUDE_DATA[(MSB << 8)|LSB]
char CMK_CLIENT_CERT_DATA[(MSB << 8)|LSB]
char CMK_RESPONSE_DATA[(MSB << 8)|LSB]
```

The client sends this message after receiving the SERVER_HELLO message from the server if a new session is being started or if client authentication has been requested. If no client authentication has been requested in the SERVER_HELLO message and an old session is being reconnected (i.e., if the CLIENT_AUTH_REQ field is zero and the RESTART_SESSION_OK field is nonzero), then the CLIENT_MASTER_KEY message is not sent.

For TOKEN exchange types, both client and server (re)set the sequence numbers to zero when this message is sent/received.

The contents of the CLEAR_KEY_DATA, ENCRYPTED_KEY_DATA, and KEY_ARG_DATA fields depend on the contents of the SH_CIPHER_SPECS_DATA and SH_EXCH_SPECS_DATA fields in the preceding SERVER_HELLO message. These will be described for each possible choice of these values in section 5.3.1 and 5.3.2, along with how the various keys (CLIENT_WRITE_KEY, SERVER_WRITE_KEY, CLIENT_MAC_KEY, and SERVER_MAC_KEY) are derived in each case. The CMK_PAD field may contain any value.

The CMK_VERIFY_PRELUDE_DATA field contains the value

Hash(CLIENT_MAC_KEY, Hash("cvp", CLIENT_HELLO, SERVER_HELLO)).

If the length of CLIENT_MAC_KEY is not an exact multiple of eight bits, then CLIENT_MAC_KEY is considered, for the purposes of MAC computation, to have (fewer than eight) zero bits appended to it, to create a string of an integral number of bytes for input into the MAC hash function. The hash function used is the one specified in SH_HASH_SPECS_DATA. The parameters are input into the hash function in the order presented above, with the strings in quotation marks representing ASCII text. Note that the client need only keep a "running hash" of all the values passed in these first two messages as they appear, then hash the result using the CLIENT_MAC_KEY when generated, to compute the value of VERIFY_PRELUDE.

If SH_CLIENT_AUTH_REQ is zero, then CMK_CLIENT_CERT_SPECS_DATA and CMK_CLIENT_SIG_SPECS_DATA are both zero, and the CMK_CLIENT_CERT and CMK_RESPONSE_DATA fields are empty. Otherwise, the CMK_RESPONSE_DATA field contains the client's authentication response, and the CMK_CLIENT_CERT_SPECS_DATA and CMK_CLIENT_SIG_SPECS_DATA fields contain the client's choices from the SH_CLIENT_CERT_SPECS_DATA and SH_CLIENT_SIG_SPECS_DATA lists, respectively. The CMK_CLIENT_CERT_DATA field contains the client's certificate, which must match the certificate type specified in the

CMK_CLIENT_CERT_SPECS_DATA field. Also, the public key in the certificate must be a signature key of the type specified in CMK_CLIENT_SIG_SPECS_DATA, which in turn must match one of the types in the SH_CLIENT_SIG_SPECS_DATA list.

CMK_RESPONSE_DATA is simply a digital signature, using the private signature key associated with the public key in the client's certificate, of the value in the CMK_VERIFY_PRELUDE_DATA field. The signature algorithm is determined by the CMK_CLIENT_SIG_SPECS_DATA field. (Note that the signature algorithm may itself require that a hash function be applied to the data being signed, apart from the one used to compute the value in CMK_VERIFY_PRELUDE_DATA.)

Upon receiving a CLIENT_MASTER_KEY message, the server performs the cipher-specific functions described in section 5.3 to establish the new CLIENT_WRITE_KEY, SERVER_WRITE_KEY, CLIENT_MAC_KEY and SERVER_MAC_KEY. The server then checks the VERIFY_PRELUDE_DATA value, the client certificate, and the client response for correctness and validity (the latter two only if client authentication had been requested). The checks of the VERIFY_PRELUDE_DATA and RESPONSE_DATA are performed by recomputing their correct value, and comparing with the values received. The certificate is verified using whatever mechanism has been implemented to validate certificates, and the signature in the RESPONSE_DATA field is verified using the verification algorithm associated with the signature scheme being used. If all of these values pass their checks, then the server sends the SERVER_VERIFY message; otherwise, an error occurs (INTEGRITY_CHECK_FAILED, BAD_CERTIFICATE, or CLIENT_AUTH_FAILED, respectively).

5.2.4 SERVER_VERIFY (Sent in the clear)

```
char SV_MSG_SERVER_VERIFY
char SV_PAD
char SV_SESSION_ID_DATA[32]
char SV_RESPONSE_LENGTH_MSB
char SV_RESPONSE_LENGTH_LSB
char SV_RESPONSE_DATA[(MSB << 8)|LSB]
```

The server sends this message upon receiving a valid CLIENT_MASTER_KEY message from the client. The SV_PAD field may contain any value. If an old session is being reconnected, then the RESPONSE_DATA field is empty, its length is zero, and the SESSION_ID_DATA field may contain any value. Otherwise, the SV_SESSION_ID_DATA field contains a value 32 **bytes in length, which should be generated randomly (in the same** sense as the CHALLENGE_DATA field in the CLIENT_HELLO message). The value PCT_SESSION_ID_NONE should not be used as a SV_SESSION_ID_DATA value. The contents of the SV_RESPONSE_DATA field are constructed by computing the function

```
Hash( SERVER_MAC_KEY, Hash( "sr", CH_CHALLENGE_DATA,
SH_CONNECTION_ID_DATA, SV_SESSION_ID_DATA ) ).
```

The CH_CHALLENGE_DATA and SH_CONNECTION_ID_DATA values, the choice of hash function used, and the value of SERVER_MAC_KEY are determined by the CLIENT_HELLO, SERVER_HELLO, SERVER_HELLO and CLIENT_MASTER_KEY messages, respectively, immediately preceding the SERVER_VERIFY message. The values are input into the interior invocation of the hash function in the exact order specified above, with the string in quotation marks representing actual ASCII text. If the length of SERVER_MAC_KEY is not an exact multiple of eight bits, then SERVER_MAC_KEY is considered, for the purposes of MAC computation, to have (fewer than eight) zero bits appended to it, to create a string of an integral number of bytes for intput into the MAC hash function.

When the client receives this message, it verifies the correctness of the response data, by computing the hash value as described above and comparing it with the one received. If it is correct, then the client proceeds with the first data record transmission; otherwise, a SERVER_AUTH_FAILED error occurs. An implementation may choose to send initial data immediately after the CLIENT_MASTER_KEY message, without waiting for the SERVER_VERIFY message to arrive, if verifying the server's identity before sending it any data is unimportant.

5.3 Algorithm and Certificate Types

5.3.1 Key Exchange Algorithms

PCT version 1 permits the following key exchange types:

PCT_EXCH_RSA_PKCS1
PCT_EXCH_RSA_PKCS1_TOKEN_DES
PCT_EXCH_RSA_PKCS1_TOKEN_DES3
PCT_EXCH_RSA_PKCS1_TOKEN_RC2
PCT_EXCH_RSA_PKCS1_TOKEN_RC4
PCT_EXCH_DH_PKCS3
PCT_EXCH_DH_PKCS3_TOKEN_DES
PCT_EXCH_DH_PKCS3_TOKEN_DES3
PCT_EXCH_FORTEZZA_TOKEN

Note that the token-based key exchange types specify cipher as well (including, implicitly, the FORTEZZA key exchange type); if one of these is chosen, then its choice of cipher overrides whatever choice of cipher appears in the SH_CIPHER_SPECS_DATA field of the SERVER_HELLO message.

For the PCT_EXCH_RSA_PKCS1 key exchange type, a MASTER_KEY value is generated by the client, which should be random in the following strong sense: attackers must not be able to predict any of the bits in the MASTER_KEY. It is recommended that the bits used be either truly random and uniformly generated (using some random physical process) or else generated using a cryptographically secure pseudorandom number generator, which was in turn seeded with a truly random and uniformly generated seed. This MASTER_KEY value is encrypted using the server's public encryption key, as obtained from the server's certificate in the SH_CERTIFICATE_DATA field of the SERVER_HELLO message. The encryption must follow the RSA PKCS#1 standard format (see [2]), block type 2. This encryption is sent to the server in the CMK_ENCRYPTED_KEY_DATA field of the CLIENT_MASTER_KEY message, and is decrypted by the server to obtain the MASTER_KEY.

For the PCT_EXCH_DH_PKCS3 key exchange type, a random private value x (generated in the same way as the MASTER_KEY above) and corresponding public value y are generated by the client following RSA PKCS#3 standard format (see [3]). The value y is then sent to the server in the CMK_ENCRYPTED_KEY_DATA field of the CLIENT_MASTER_KEY message. The client's private value x, along with the public value y' included in the server's certificate in the SH_CERTIFICATE_DATA field of the SERVER_HELLO message, is used to generate the MASTER_KEY. The server uses its private value, x', along with the y value sent by the client, to obtain the same MASTER_KEY value.

For the various TOKEN key exchange types, all the key material is contained in the CMK_ENCRYPTED_KEY_DATA field, but the format of the data is defined by the token implementation, or by other future documents.

The length of the MASTER_KEY depends on the key exchange type. For the PCT_EXCH_RSA_PKCS1 and PCT_EXCH_DH_PKCS3 exchange types, the MASTER_KEY is a 128-bit value. The CLIENT_WRITE_KEY and SERVER_WRITE_KEY are computed as follows:

```
CLIENT_WRITE_KEY_i = Hash( i, "cw", MASTER_KEY, "cw"^i,
SH_CONNECTION_ID_DATA, "cw"^i, SH_CERTIFICATE_DATA,"cw"^i,
CH_CHALLENGE_DATA, "cw"^i )

SERVER_WRITE_KEY_i = Hash( i, "svw", MASTER_KEY, "svw"^i,
SH_CONNECTION_ID_DATA, "svw"^i, CH_CHALLENGE_DATA, "svw"^i )
```

The values in quotation marks are treated as (sequences of) ASCII characters; "x"^i denotes i copies of the string "x" concatenated together. The function "Hash" is the one determined by the value of SH_HASH_SPECS_DATA. The parameters are input into the hash function in the order presented above; the variable i is input as a single-byte unsigned integer. The WRITE_KEYs (i.e., CLIENT_WRITE_KEY and SERVER_WRITE_KEY) are obtained by concatenating WRITE_KEY_1 through WRITE_KEY_m, where m is the negotiated encryption key length (the value in the third byte of the SH_CIPHER_SPECS_DATA field) divided by the hash output length, in bits, rounded up to the nearest integer. This resulting string is then truncated if necessary (by removing bits from the end) to produce a string of the correct length.

The CLIENT_MAC_KEY and SERVER_MAC_KEY are computed as follows:

CLIENT_MAC_KEY_i = Hash(i, MASTER_KEY, "cmac"^i,
SH_CONNECTION_ID_DATA, "cmac"^i, SH_CERTIFICATE_DATA, "cmac"^i,

CH_CHALLENGE_DATA, "cmac"^i)

SERVER_MAC_KEY_i = Hash(i, MASTER_KEY, "svmac"^i,
SH_CONNECTION_ID_DATA, "svmac"^i, CH_CHALLENGE_DATA, "svmac"^i)

The values in quotation marks are treated as (sequences of) ASCII characters; "x"^i denotes i copies of the string "x" concatenated together. The function "Hash" is the one determined by the value of SH_HASH_SPECS_DATA. The parameters are input into the hash function in the order presented above; the variable i is input as a single-byte unsigned integer. The MAC_KEYs (ie., CLIENT_MAC_KEY and SERVER_MAC_KEY) are obtained by concatenating MAC_KEY_1 through MAC_KEY_m, where m is the negotiated MAC key length (64 plus the value in the fourth byte of the SH_CIPHER_SPECS_DATA field) divided by the hash output length, in bits, rounded up to the nearest integer. This resulting string is then truncated if necessary (by removing bits from the end) to produce a string of the correct length.

Note that tokens which are capable of deriving keys using "keyed hashes", as described above, are free to use the PCT_EXCH_RSA_PKCS1 or PCT_EXCH_DH_PKCS3 key exchange type to exchange the MASTER_KEY, and then to derive the rest of the keys normally. The TOKEN key exchange types are for tokens that cannot do such keyed-hash key derivation, and can only use an exchanged key for bulk encryption (of, for example, other keys). Such tokens can exchange multiple keys by using an initially exchanged MASTER_KEY to encrypt other keys, as described above.

5.3.2 Cipher Types

PCT version 1 permits the following cipher types to be specified:

PCT_CIPHER_DES
PCT_CIPHER_IDEA
PCT_CIPHER_RC2
PCT_CIPHER_RC4
PCT_CIPHER_DES_112
PCT_CIPHER_DES_168

Each of these types is denoted by a two-byte code, and is followed in CIPHER_SPECS_DATA fields by two one-byte length specifications, as described in section 5.2.1. An encryption length specification of zero associated with any cipher denotes the choice of no encryption; a key exchange is performed in such cases solely to share keys for MAC computation. The MAC key length must always be at least 64 bits (see section 5.2.1).

The CLEAR_KEY_DATA field is used only when encryption keys of length less than the standard length for the specified cipher are used;

otherwise, the field is empty. When a key length is specified which is less than the standard key length for the specified cipher, then keys of the specified length are derived normally as described in section 5.3.1, and then "expanded" to derive standard-length keys. The expansion proceeds as follows:

1. Assign to d the result of dividing the standard key length for the cipher, in bits, by the output length of the hash function, in bits, rounded up to the nearest integer.

2. Divide CLEAR_KEY_DATA sequentially into d equal subsegments. (Note that the length of the CLEAR_KEY_DATA field must therefore be a multiple of d bytes, and that no two of its d equal parts, when so divided, may be identical.) Denote these subsegments CLEAR_KEY_DATA_1 through CLEAR_KEY_DATA_d.

3. Compute the d hash values

STANDARD_LENGTH_KEY_i := Hash(i, "sl"^i, WRITE_KEY, "sl"^i, CLEAR_KEY_DATA_i).

The values in quotation marks are treated as (sequences of) ASCII characters; "x"^i denotes i copies of the string "x" concatenated together. The function "Hash" is the one determined by the value of SH_HASH_SPECS_DATA. The parameters are input into the hash function in the order presented above; the variable i is input as a single-byte unsigned integer. The WRITE_KEY is the encryption key (CLIENT_WRITE_KEY or SERVER_WRITE_KEY) being expanded to standard length. If the length of the WRITE_KEY is not an exact number of bytes, then its final byte is padded with zeroes to increase its length to an exact number of bytes.

4. Concatenate STANDARD_LENGTH_KEY_1 through STANDARD_LENGTH_KEY_d, and then truncate as necessary (by removing bits from the end) to produce the STANDARD_LENGTH_KEY which is actually used for encryption.

The KEY_ARG_DATA field contains a random eight-byte value to be used as an initialization vector (IV) for the first encrypted message when a block cipher (any cipher except RC4) is used. The IV for the first block encrypted in any subsequent encrypted message is simply the last encrypted block of the previous message. The KEY_ARG_DATA field is empty when cipher type PCT_CIPHER_RC4 (or key exchange type PCT_EXCH_RSA_PKCS1_TOKEN_RC4) is used.

PCT_CIPHER_DES denotes DES (see [4]). Its standard key length is 56 bits. PCT_CIPHER_DES_112 and PCT_CIPHER_DES_168 denote ciphers in which the input is first encrypted under DES with a first key, then "decrypted" under DES with a second key, then encrypted under DES with a third key. For PCT_CIPHER_DES_112, the first and third keys are identical, and correspond to the initial 56 bits of the 112-bit WRITE_KEY. The second key corresponds to the final 56 bits of the WRITE_KEY. For PCT_CIPHER_DES_168, the three keys are distinct, and correspond to the first, second, and third 56-bit subsegments of the WRITE_KEY. All three of these DES-based cipher types have 64-bit data blocks and are used with cipher block chaining (CBC).

The standard key lengths for PCT_CIPHER_DES_112 and PCT_CIPHER_DES_168 are 112 bits and 168 bits, respectively. If a key length less than the standard length is specified for one of these ciphers (or for PCT_CIPHER_DES), then the WRITE_KEY is expanded to the standard length as described above.

Note that before use, each 56-bit DES key must be "adjusted" to add eight parity bits to form an eight-byte DES key (see [4]). Similarly, if the specified WRITE_KEY length is less than its corresponding standard length, then each WRITE_KEY is expanded to the standard length using CLEAR_KEY_DATA as described above, to produce one, two, or three keys of 56 bits each, which are then each "adjusted" by adding parity bits to form an eight-byte key.

PCT_CIPHER_IDEA denotes the IDEA block cipher (see [5]), with 64-bit data blocks and cipher block chaining. This cipher has a standard key length of 128 bits.

PCT_CIPHER_RC2 denotes the RC2 block cipher, with 64-bit blocks and cipher block chaining. Like IDEA, this cipher has a standard key length of 128 bits.

PCT_CIPHER_RC4 denotes the RC4 stream cipher. Like the IDEA and RC2 block ciphers, this cipher has a standard key length of 128 bits.

5.3.3 Hash Types

PCT version 1 permits the following hash function types to be specified:

PCT_HASH_MD5
PCT_HASH_MD5_TRUNC_64
PCT_HASH_SHA
PCT_HASH_SHA_TRUNC_80
PCT_HASH_DES_DM

PCT_CIPHER_MD5 denotes the MD5 hash function (see [6]), with 128-bit output. PCT_CIPHER_MD5_TRUNC_64 denotes the MD5 hash function, with its 128-bit output truncated to 64 bits. PCT_HASH_SHA denotes the Secure Hash Algorithm (see [7]), with 160-bit output. PCT_HASH_SHA_TRUNC_80 denotes the Secure Hash Algorithm, with its 160-bit output truncated to 80 bits. PCT_HASH_DES_DM denotes the DES-based Davies-Meyer hash algorithm (see [8]), with 64-bit output.

5.3.4 Certificate Types

PCT version 1 permits the following certificate types to be specified:

```
PCT_CERT_NONE
PCT_CERT_X509
PCT_CERT_PKCS7
```

These types apply equally to the client's and server's certificates. PCT_CERT_NONE denotes that no certificate is necessary; this type can be included by, say, the server as a choice, thereby making authentication optional for the client. PCT_CERT_X509 denotes a CCITT X.509 standard-conformant certificate (see [9]). PCT_CERT_PKCS7 denotes an RSA PKCS#7 standard-conformant certificate (see [10]).

5.3.5 Signature Types

PCT version 1 permits the following signature key types to be specified:

PCT_SIG_NONE
PCT_SIG_RSA_MD5
PCT_SIG_RSA_SHA
PCT_SIG_DSA_SHA

PCT_SIG_NONE denotes that no signature is necessary; this type can be included by the server as a choice, thereby making authentication optional for the client. PCT_SIG_RSA_MD5 denotes the signature scheme consisting of hashing the data to be signed using the MD5 hash algorithm, and then performing an RSA private-key signature function (the inverse of RSA encryption) on the result. The signature must conform to RSA PKCS#1, block type 1 (see [2]). PCT_SIG_RSA_SHA denotes the same signature scheme with SHA substituted for MD5. PCT_SIG_DSA_SHA denotes the signature scheme consisting of hashing the data to be signed using the SHA hash algorithm, then computing a signature of the resulting value using the Digital Signature Algorithm (DSA; see [11]).

5.4 Errors

Error handling in the PCT protocol is very simple. When an error is detected during the handshake phase, the detecting party sends a message to the other party indicating the error so that both parties will know about it, and then closes the connection. If a party detects an error after it has sent its last handshake message, the detecting party simply closes the connection without sending an error message. In the second case there are only two possible errors, and the party that does not detect the error can distinguish them as follows: if the server sees an aborted connection and the most recent message it sent the client was a handshake message, then the error was SERVER_AUTH_FAILED; otherwise, the error was INTEGRITY_CHECK_FAILED.

Receiving an error message also causes the receiving party to close the connection. Servers and clients should not make any further use of any keys, challenges, connection identifiers, or session identifiers associated with such an aborted connection.

It is recommended that implementations perform some kind of alert or logging function when errors are generated to facilitate monitoring of various types of attack on the system.

The message sent in the event of a handshake-phase error has the following form:

```
char MSG_ERROR
char ERROR_CODE_MSB
char ERROR_CODE_LSB
char ERROR_INFO_LENGTH_MSB
char ERROR_INFO_LENGTH_LSB
char ERROR_INFO_DATA[(MSB << 8)|LSB]
```

The ERROR_INFO_LENGTH field is zero except in the case of the SPECS_MISMATCH error message, which has a six-byte ERROR_INFO_DATA field.

The PCT Handshake Protocol defines the following errors:

PCT_ERR_BAD_CERTIFICATE

This error occurs when the client receives a SERVER_HELLO message in which the certificate is invalid, either because one or more of the signatures in the certificate is invalid, or because the identity or attributes on the certificate are in some way incorrect.

PCT_ERR_CLIENT_AUTH_FAILED

This error occurs when the server receives a CLIENT_MASTER_KEY message from the client in which the client's authentication response is incorrect. The certificate may be invalid, the signature may be invalid, or the contents of the signed response may be incorrect.

PCT_ERR_ILLEGAL_MESSAGE

This error occurs under a number of circumstances. For example, it occurs when an unrecognized security escape code is received, when an unrecognized handshake message is encountered, or when the value of CH_OFFSET is to large for its CLIENT_HELLO message.

PCT_ERR_INTEGRITY_CHECK_FAILED

This error occurs when either the client or the server receives a message in which the MAC_DATA is incorrect. It is also recommended that the record be treated as if it contained no data, in order to ensure that applications do not receive and process invalid data before learning that it has failed its integrity check.

This error also occurs when the VERIFY_PRELUDE_DATA value sent by the client in the CLIENT_MASTER_KEY message (during the handshake phase) is incorrect. In this case, an error message is sent.

PCT_ERR_SERVER_AUTH_FAILED

This error occurs when the client receives a SERVER_HELLO or SERVER_VERIFY message in which the authentication response is incorrect.

PCT_ERR_SPECS_MISMATCH

This error occurs when a server cannot find a cipher, hash function, certificate type, or key exchange algorithm it supports in the lists supplied by the client in the CLIENT_HELLO message. It also occurs when the client cannot find a certificate or signature type it supports in the list supplied by the server in a SERVER_HELLO message that requests client authentication. (Note that the client or server can select the "NONE" option as the last resort for any security feature it wishes to make optional. For example, the server can make client authentication optional for the client by passing a list of certificate and signature types, each list containing the "NONE" type as the last entry.) This error may also occur as a result of a mismatch in cipher specifications or client authentication requests between the initial specifications and those that resulted from a redo handshake sequence.

The error message for this error includes a six-byte informational field, defined as follows:

```
char SPECS_MISMATCH_CIPHER
char SPECS_MISMATCH_HASH
char SPECS_MISMATCH_CERT
char SPECS_MISMATCH_EXCH
char SPECS_MISMATCH_CLIENT_CERT
char SPECS_MISMATCH_CLIENT_SIG
```

Each field is set to a non-zero value if and only if the corresponding list resulted in a mismatch. For example, if and only if the SPECS_MISMATCH error message is being sent because server failed to find a certificate type it supports in the list supplied by the client in the CH_CERT_SPECS_DATA field, then the SPECS_MISMATCH_CERT field in the error message would be non-zero.

5.5 Constants

Following is a list of constant values used in the PCT protocol version 1.

5.5.1 Message type codes

These codes are each placed in the first byte of the corresponding PCT handshake phase message.

```
PCT_MSG_CLIENT_HELLO              :=      0x01
PCT_MSG_SERVER_HELLO              :=      0x02
PCT_MSG_CLIENT_MASTER_KEY         :=      0x03
PCT_MSG_SERVER_VERIFY             :=      0x04
PCT_MSG_ERROR                     :=      0x05
```

5.5.2 Specification Type Codes

These are codes used to specify types of cipher, key exchange, hash function, certificate, and digital signature in the protocol.

```
PCT_EXCH_RSA_PKCS1                :=      0x0001
PCT_EXCH_RSA_PKCS1_TOKEN_DES      :=      0x0002
PCT_EXCH_RSA_PKCS1_TOKEN_DES3     :=      0x0003
PCT_EXCH_RSA_PKCS1_TOKEN_RC2      :=      0x0004
PCT_EXCH_RSA_PKCS1_TOKEN_RC4      :=      0x0005
PCT_EXCH_DH_PKCS3                 :=      0x0006
PCT_EXCH_DH_PKCS3_TOKEN_DES       :=      0x0007
PCT_EXCH_DH_PKCS3_TOKEN_DES3      :=      0x0008
PCT_EXCH_FORTEZZA_TOKEN           :=      0x0009

PCT_CIPHER_DES          :=      0x0001
PCT_CIPHER_IDEA         :=      0x0002
PCT_CIPHER_RC2          :=      0x0003
PCT_CIPHER_RC4          :=      0x0004
PCT_CIPHER_DES_112      :=      0x0005
PCT_CIPHER_DES_168      :=      0x0006

PCT_HASH_MD5            :=      0x0001
PCT_HASH_MD5_TRUNC_64   :=      0x0002
PCT_HASH_SHA            :=      0x0003
PCT_HASH_SHA_TRUNC_80   :=      0x0004
PCT_HASH_DES_DM         :=      0x0005

PCT_CERT_NONE           :=      0x0000
PCT_CERT_X509           :=      0x0001
PCT_CERT_PKCS7          :=      0x0002

PCT_SIG_NONE            :=      0x0000
PCT_SIG_RSA_MD5         :=      0x0001
PCT_SIG_RSA_SHA         :=      0x0002
PCT_SIG_DSA_SHA         :=      0x0003
```

5.5.3 Error Codes

These codes are used to identify errors, when they occur, in error messages.

```
PCT_ERR_BAD_CERTIFICATE           :=      0x0001
```

```
PCT_ERR_CLIENT_AUTH_FAILED      :=      0x0002
PCT_ERR_ILLEGAL_MESSAGE         :=      0x0003
PCT_ERR_INTEGRITY_CHECK_FAILED  :=      0x0004
```

```
PCT_ERR_SERVER_AUTH_FAILED      :=      0x0005
PCT_ERR_SPECS_MISMATCH          :=      0x0006
```

5.5.4 Miscellaneous Codes

These include escape type codes, version numbers, and assorted constants associated with the PCT protocol.

```
PCT_SESSION_ID_NONE             :=      0x00 (32 bytes of zeros)

PCT_ET_OOB_DATA                 :=      0x01
PCT_ET_REDO_CONN                :=      0x02

PCT_VERSION_1                   :=      0x8001

PCT_CH_OFFSET_V1                :=      0x000A

PCT_MAX_RECORD_LENGTH_2_BYTE_HEADER :=  32767
PCT_MAX_RECORD_LENGTH_3_BYTE_HEADER :=  16383
```

6. Security Considerations

This entire document is about security.

References

[1] K. Hickman and T. Elgamal. The SSL Protocol. Internet-draft, June 1995.

[2] RSA Laboratories, "PKCS #1: RSA Encryption Standard", Version 1.5, November 1993.

[3] RSA Laboratories, "PKCS #3: "Diffie-Hellman Key-Agreement Standard", Version 1.4, November 1993.

[4] NBS FIPS PUB 46, "Data Encryption Standard", National Bureau of Standards, US Department of Commerce, Jan. 1977.

[5] X. Lai, "On the Design and Security of Block Ciphers", ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.

[6] R. Rivest, RFC 1321: "The MD5 Message Digest Algorithm", April 1992.

[7] NIST FIPS PUB 180-1, "Secure Hash Standard", National Institute of Standards and Technology, US Department of Commerce, Apr. 1995.

[8] ISO/IEC 9797, "Data Cryptographic Techniques--Data Integrity

Mechanism Using a Cryptographic Check Function Employing a Block
Cipher Algorithm", 1989.

[9] CCITT. Recommendation X.509: "The Directory - Authentication Framework". 1988.

[10] RSA Laboratories, "PKCS #7: Cryptographc Message Syntax Standard", Version 1.5, November 1993.

[11] NIST FIPS PUB 186, "Digital Signature Standard", National Institute of Standards and Technology, US Department of Commerce, May 1994.

[12] B. Schneier, "Applied Cryptography: Protocols, Algorithms, and Source Code in C", John Wiley & Sons, Inc., 1994.

[13] R.L. Rivest, A. Shamir, L. Adelman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems" MIT Laboratory for Computer Science and Department of Mathematics, S.L. Graham, R.L. Rivest ed. Communications of the ACM, February 1978 (Vol 21, No. 2) pages 120-126.

Patent Statement

This version of the PCT protocol relies on the use of patented public key encryption technology for authentication and encryption. The Internet Standards Process as defined in RFC 1310 requires a written statement from the Patent holder that a license will be made available to applicants under reasonable terms and conditions prior to approving a specification as a Proposed, Draft or Internet Standard.

See existing RFCs, including RFC 1170, that discuss known public key cryptography patents and licensing terms and conditions.

The Internet Society, Internet Architecture Board, Internet Engineering Steering Group and the Corporation for National Research Initiatives take no position on the validity or scope of the patents and patent applications, nor on the appropriateness of the terms of the assurance. The Internet Society and other groups mentioned above have not made any determination as to any other intellectual property rights which may apply to the practice of this standard. Any further consideration of these matters is the user's own responsibility.

Author's Address

Josh Benaloh/Butler Lampson/Daniel R. Simon/Terence Spies/Bennet Yee
Microsoft Corp.
One Microsoft Way
Redmond WA 98052
USA

pct@microsoft.com

This Internet-Draft expires 27 March 1996.