Network Working Group Jon Callas
Category: INTERNET-DRAFT Pretty Good Privacy
draft-ietf-openpgp-formats-00.txt Lutz Donnerhacke
Expires May 1998 IN-Root-CA Individual Network e.V.
November 1997 Hal Finney
Pretty Good Privacy
Rodney Thayer
Sable Technology

OP Formats - OpenPGP Message Format
draft-ietf-openpgp-formats-00.txt

Status of this Memo

Abstract

This document is maintained in order to publish all necessary information needed to develop interoperable applications based on the OP format. It is not a step-by-step cookbook for writing an application, it describes only the format and methods needed to read, check, generate and write conforming packets crossing any network. It does not deal with storing and implementation questions albeit it is necessary to avoid security flaws.

OP (Open-PGP) software uses a combination of strong public-key and conventional cryptography to provide security services for electronic communications and data storage. These services include confidentiality, key management, authentication and digital signatures. This document specifies the message formats used in OP.

Table of Contents

1. Introduction

This document provides information on the message-exchange packet formats used by OP to provide encryption, decryption, signing, key management and functions. It builds on the foundation provided RFC 1991 **"PGP Message Exchange Formats" [1].**

1.1 Terms

OP - OpenPGP. This is a definition for security software that uses PGP 5.x as a basis.

PGP - Pretty Good Privacy. PGP is a family of software systems developed by Philip R. Zimmermann from which OP is based.

PGP 2.6.x - This version of PGP has many variants, hence the term PGP 2.6.x. It used only RSA and IDEA for its cryptography.

PGP 5.x - This version of PGP is formerly known as "PGP 3" in the community and also in the predecessor of this document, RFC1991. It has new formats and corrects a number of problems in the PGP 2.6.x. It is referred to here as PGP 5.x because that software was the first release of the "PGP 3" code base.

"PGP", "Pretty Good", and "Pretty Good Privacy" are trademarks of

Pretty Good Privacy, Inc.

2. General functions

OP provides data integrity services for messages and data files by using these core technologies:

-digital signature -encryption -compression -radix-64 conversion

In addition, OP provides key management and certificate services.

2.1 Confidentiality via Encryption

OP offers two encryption options to provide confidentiality: conventional (symmetric-key) encryption and public key encryption. With public-key encryption, the message is actually encrypted using a conventional encryption algorithm. In this mode, each conventional key is used only once. That is, a new key is generated as a random number for each message. Since it is used only once, the "session key" is bound to the message and transmitted with it. To protect the key, it is encrypted with the receiver's public key. The sequence is as follows:

1. The sender creates a message.
2. The sending OP generates a random number to be used as a session key for this message only.
3. The session key is encrypted using each recipient's public key. These "encrypted session keys" start the message.
4. The sending OP encrypts the message using the session key, which forms the remainder of the message. Note that the message is also usually compressed.
5. The receiving OP decrypts the session key using the recipient's private key.
6. The receiving OP decrypts the message using the session key. If the message was compressed, it will be decompressed.

Both digital signature and confidentiality services may be applied to the same message. First, a signature is generated for the message and attached to the message. Then, the message plus signature is encrypted using a conventional session key. Finally, the session key is encrypted using public-key encryption and prepended to the encrypted block.

2.2 Authentication via Digital signature

The digital signature uses a hash code or message digest algorithm, and a public-key signature algorithm. The sequence is as follows:

1. The sender creates a message.
2. The sending software generates a hash code of the message
3. The sending software generates a signature from the hash code using the sender's private key.

4. The binary signature is attached to the message.
5. The receiving software keeps a copy of the message signature.
6. The receiving software generates a new hash code for the received message and verifies it using the message's signature. If the

verification is successful, the message is accepted as authentic.

2.3 Compression

OP implementations MAY compress the message after applying the signature but before encryption.

2.4 Conversion to Radix-64

OP's underlying native representation for encrypted messages, signature certificates, and keys is a stream of arbitrary octets. Some systems only permit the use of blocks consisting of seven-bit, printable text. For transporting OP's native raw binary octets through email channels, a printable encoding of these binary octets is needed. OP provides the service of converting the raw 8-bit binary octet stream to a stream of printable ASCII characters, called Radix-64 encoding or ASCII Armor.

In principle, any printable encoding scheme that met the requirements of the email channel would suffice, since it would not change the underlying binary bit streams of the native OP data structures. The OP standard specifies one such printable encoding scheme to ensure interoperability.

OP's Radix-64 encoding is composed of two parts: a base64 encoding of the binary data, and a checksum. The base64 encoding is identical to the MIME base64 content-transfer-encoding [RFC 2045, Section 6.8]. An OP implementation MAY use ASCII Armor to protect the raw binary data.

The checksum is a 24-bit CRC converted to four characters of radix-64 encoding by the same MIME base64 transformation, preceded by an equals sign (=). The CRC is computed by using the generator 0x864CFB and an initialization of 0xB704CE. The accumulation is done on the data before it is converted to radix-64, rather than on the converted data. (For more information on CRC functions, see chapter 19 of [CAMPBELL].)

{{Editor's note: This is old text, dating back to RFC 1991. I have never liked the glib way the CRC has been dismissed, but I also know that this is no place to start a discussion of CRC theory. Should we construct a sample implementation in C and put it in an appendix? --jdcc}}

The checksum with its leading equal sign MAY appear on the first line after the Base64 encoded data.

Rationale for CRC-24: The size of 24 bits fits evenly into printable base64. The nonzero initialization can detect more errors than a zero initialization.

2.4.1 Forming ASCII Armor

When OP encodes data into ASCII Armor, it puts specific headers around the data, so OP can reconstruct the data later. OP informs the user what kind of data is encoded in the ASCII armor through the use of the headers.

Concatenating the following data creates ASCII Armor:

- An Armor Header Line, appropriate for the type of data - Armor Headers - A blank (zero-length, or containing only whitespace) line - The ASCII-Armored data - An Armor Checksum - The Armor Tail, which depends on the Armor Header Line.

An Armor Header Line consists of the appropriate header line text surrounded by five (5) dashes ('-', 0x2D) on either side of the header line text. The header line text is chosen based upon the type of data that is being encoded in Armor, and how it is being encoded. Header line texts include the following strings:

BEGIN PGP MESSAGE used for signed, encrypted, or compressed files

BEGIN PGP PUBLIC KEY BLOCK used for armoring public keys

BEGIN PGP PRIVATE KEY BLOCK used for armoring private keys

BEGIN PGP MESSAGE, PART X/Y used for multi-part messages, where the armor is split amongst Y parts, and this is the Xth part out of Y.

BEGIN PGP MESSAGE, PART X used for multi-part messages, where this is the Xth part of an unspecified number of parts. Requires the MESSAGE-ID Armor Header to be used.

BEGIN PGP SIGNATURE used for detached signatures, OP/MIME signatures, and signatures following clearsigned messages

The Armor Headers are pairs of strings that can give the user or the receiving OP message block some information about how to decode or use the message. The Armor Headers are a part of the armor, not a part of the message, and hence are not protected by any signatures applied to the message.

The format of an Armor Header is that of a key-value pair. A colon (':' 0x38) and a single space (0x20) separate the key and value. OP should consider improperly formatted Armor Headers to be corruption of the ASCII Armor. Unknown keys should be reported to the user, but OP should continue to process the message. Currently defined Armor Header Keys include "Version" and "Comment", which define the OP Version used to encode the message and a user-defined comment.

The "MessageID" Armor Header specifies a 32-character string of printable characters. The string must be the same for all parts of a multi-part message that uses the "PART X" Armor Header. MessageID strings should be chosen with enough internal randomness that no two messages would have the same MessageID string.

The MessageID should not appear unless it is in a multi-part message. If it appears at all, it should be computed from the message in a deterministic fashion, rather than contain a purely random value. This is to allow anyone to determine that the MessageID cannot serve as a covert means of leaking cryptographic key information.

{{Editor's note: This needs to be cleaned up, with a table of the defined headers. Also, the MessageID description is too vague about how random the id has to be.}}

The Armor Tail Line is composed in the same manner as the Armor Header Line, except the string "BEGIN" is replaced by the string "END."

2.4.2 Encoding Binary in Radix-64

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating three 8-bit input groups. These 24 bits are then treated as four concatenated 6-bit groups, each of which is translated into a single digit in the Radix-64 alphabet. When encoding a bit stream with the Radix-64 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first. That is, the first bit in the stream will be the high-order bit in the first 8-bit byte, and the eighth bit will be the low-order bit in the first 8-bit byte, and so on.

```
+--first octet--+-second octet--+--third octet--+
|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|
+-----------+---+-------+-------+---+-----------+
|5 4 3 2 1 0|5 4 3 2 1 0|5 4 3 2 1 0|5 4 3 2 1 0|
+--1.index--+--2.index--+--3.index--+--4.index--+
```

Each 6-bit group is used as an index into an array of 64 printable characters from the table below. The character referenced by the index is placed in the output string.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/

13 N	30 e	47 v	
14 O	31 f	48 w	(pad) =
15 P	32 g	49 x	
16 Q	33 h	50 y	

The encoded output stream must be represented in lines of no more than 76 characters each.

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. There are three possibilities:

- The last data group has 24 bits (3 octets). No special processing is needed.

- The last data group has 16 bits (2 octets). The first two 6-bit groups are processed as above. The third (incomplete) data group has two zero-value bits added to it, and is processed as above. A pad character (=) is added to the output.

- The last data group has 8 bits (1 octet). The first 6-bit group is processed as above. The second (incomplete) data group has four zero-value bits added to it, and is processed as above. Two pad characters (=) are added to the output.

2.4.3 Decoding Radix-64

Any characters outside of the base64 alphabet are ignored in Radix-64 data. Decoding software must ignore all line breaks or other characters not found in the table above.

In Radix-64 data, characters other than those in the table, line breaks, and other white space probably indicate a transmission error, about which a warning message or even a message rejection might be appropriate under some circumstances.

Because it is used only for padding at the end of the data, the occurrence of any "=" characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three and no "=" characters are present.

2.4.4 Examples of Radix-64

```
Input data:  0x14fb9c03d97e
Hex:     1   4    f   b    9   c     | 0   3    d   9    7   e
8-bit:   00010100 11111011 10011100  | 00000011 11011001 11111110
6-bit:   000101 001111 101110 011100 | 000000 111101 100111 111110
Decimal: 5      15     46     28       0      61     37     63
Output:  F      P      u      c        A      9      l      /

Input data:  0x14fb9c03d9
Hex:     1   4    f   b    9   c     | 0   3    d   9
8-bit:   00010100 11111011 10011100  | 00000011 11011001
```

```
                                                pad with 00
6-bit:   000101 001111 101110 011100 | 000000 111101 100100
Decimal: 5      15     46     28       0      61     36
                                                   pad with =
```

```
Output:  F      P      u      c       A      9      k      =

Input data:  0x14fb9c03
Hex:     1   4    f   b    9   c     | 0   3
8-bit:   00010100 11111011 10011100  | 00000011
                                       pad with 0000
6-bit:   000101 001111 101110 011100 | 000000 110000
Decimal: 5      15     46     28       0      48
                                            pad with =      =
Output:  F      P      u      c       A      w      =      =
```

2.5 Example of an ASCII Armored Message

```
  -----BEGIN PGP MESSAGE-----
  Version: OP V0.0

  owFbx8DAYFTCWlySkpkHZDKEFCXmFedmFhdn5ucpZKdWFiv4hgaHKPj5hygUpSbn
  l6UWpabo8XIBAA==
  =3m1o
  -----END PGP MESSAGE-----
```

Note that this example is indented by two spaces.

2.6 Cleartext signature framework

Sometimes it is necessary to sign a textual octet stream without ASCII armoring the stream itself, so the signed text is still readable without special software. In order to bind a signature to such a cleartext, this framework is used. (Note that RFC 2015 defines another way to clear sign messages for environments that support MIME.)

The cleartext signed message consists of:

- The cleartext header '-----BEGIN PGP SIGNED MESSAGE-----' on a single line,
- Zero or more "Hash" Armor Headers (3.1.2.4),
- Exactly one empty line not included into the message digest,
- The dash-escaped cleartext that is included into the message digest,
- The ASCII armored signature(s) including the Armor Header and Armor Tail Lines.

If the "Hash" armor header is given, the specified message digest algorithm is used for the signature. If this header is missing, SHA-1 is assumed. If more than one message digest is used in the signature, the "Hash" armor header contains a comma-delimited list of used message digests. As an abbreviation, the "Hash" armor header may be placed on the cleartext header line, inserting a comma after the word 'MESSAGE', as follows:

'-----BEGIN PGP SIGNED MESSAGE, Hash: MD5, SHA1'.

{{Editor's note: Should the above armor header line stay or go?
There's no reason that the "Hash:" armor header can't have multiple

hashes in it. I think anything that reduces parsing complexity is a Good Thing. --jdcc}}

Current message digest names are:

- "SHA1"
- "MD5"
- "RIPEMD160"

Dash escaped cleartext is the ordinary cleartext where every line starting with a dash '-' (0x2D) is prepended by the sequence dash '-' (0x2D) and space ' ' (0x20). This prevents the parser from recognizing armor headers of the cleartext itself. The message digest is computed using the cleartext itself, not the dash escaped form.

As with binary signatures on text documents (see below), the cleartext signature is calculated on the text using canonical <CR><LF> line endings. The line ending (i.e. the <CR><LF>) before the '-----BEGIN PGP SIGNATURE-----' line that terminates the signed text is not considered part of the signed text.

Also, any trailing whitespace (spaces, and tabs, 0x09) at the end of any line is ignored when the cleartext signature is calculated.

3. Data Element Formats

This section describes the data elements used by OP.

3.1 Scalar numbers

Scalar numbers are unsigned, and are always stored in big-endian format. Using n[k] to refer to the kth octet being interpreted, the value of a two-octet scalar is ((n[0] << 8) + n[1]). The value of a four-octet scalar is ((n[0] << 24) + (n[1] << 16) + (n[2] << 8) + n[3]).

3.2 Multi-Precision Integers

Multi-Precision Integers (also called MPIs) are unsigned integers used to hold large integers such as the ones used in cryptographic calculations.

An MPI consists of two pieces: a two-octet scalar that is the length of the MPI in bits followed by a string of octets that contain the actual integer.

These octets form a big-endian number; a big-endian number can be made into an MPI by prefixing it with the appropriate length.

Examples:

(all numbers are in hexadecimal)

The string of octets [00 01 01] forms an MPI with the value 1. The string [00 09 01 FF] forms an MPI with the value of 511.

Additional rules:

The size of an MPI is ((MPI.length + 7) / 8) + 2.

The length field of an MPI describes the length starting from its most significant non-zero bit. Thus, the MPI [00 02 01] is not formed correctly. It should be [00 01 01].

3.3 Counted Strings

A counted string consists of a length and then N octets of string data. Its default character set is UTF-8 [RFC2044] encoding of Unicode [ISO10646].

3.4 Time fields

A time field is an unsigned four-octet number containing the number of seconds elapsed since midnight, 1 January 1970 UTC.

3.5 String-to-key (S2K) specifiers

String-to-key (S2K) specifiers are used to convert passphrase strings into conventional encryption/decryption keys. They are used in two places, currently: to encrypt the secret part of private keys in the private keyring, and to convert passphrases to encryption keys for conventionally encrypted messages.

3.5.1 String-to-key (S2k) specifier types

There are three types of S2K specifiers currently supported, as follows:

3.5.1.1 Simple S2K

This directly hashes the string to produce the key data. See below for how this hashing is done.

```
        Octet 0:                0x00
        Octet 1:                hash algorithm
```

3.5.1.2 Salted S2K

This includes a "salt" value in the S2K specifier -- some arbitrary data -- that gets hashed along with the passphrase string, to help prevent dictionary attacks.

```
Octet 0:                0x01
Octet 1:                hash algorithm
Octets 2-9:             8-octet salt value
```

3.5.1.3 Iterated and Salted S2K

This includes both a salt and an octet count. The salt is combined with the passphrase and the resulting value is hashed repeatedly. This further increases the amount of work an attacker must do to try dictionary attacks.

```
Octet 0:            0x03
Octet 1:            hash algorithm
Octets 2-9:         8-octet salt value
Octet 10:           count, in special format (described below)
```

3.5.2 String-to-key usage

Implementations MUST implement simple S2K and salted S2K specifiers. Implementations MAY implement iterated and salted S2K specifiers. Implementations SHOULD use salted S2K specifiers, as simple S2K specifiers are more vulnerable to dictionary attacks.

3.5.2.1 Secret key encryption

An S2K specifier can be stored in the secret keyring to specify how to convert the passphrase to a key that unlocks the secret data. Older versions of PGP just stored a cipher algorithm octet preceding the secret data or a zero to indicate that the secret data was unencrypted. The MD5 hash function was always used to convert the passphrase to a key for the specified cipher algorithm.

For compatibility, when an S2K specifier is used, the special value 255 is stored in the position where the hash algorithm octet would have been in the old data structure. This is then followed immediately by a one-octet algorithm identifier, and then by the S2K specifier as encoded above.

Therefore, preceding the secret data there will be one of these possibilities:

```
0               secret data is unencrypted (no pass phrase)
255             followed by algorithm octet and S2K specifier
Cipher alg      use Simple S2K algorithm using MD5 hash
```

This last possibility, the cipher algorithm number with an implicit use of MD5 is provided for backward compatibility; it should be understood, but not generated.

These are followed by an 8-octet Initial Vector for the decryption of the secret values, if they are encrypted, and then the secret key values themselves.

3.5.2.2 Conventional message encryption

PGP 2.X always used IDEA with Simple string-to-key conversion when conventionally encrypting a message. PGP 5 can create a Conventional Encrypted Session Key packet at the front of a message. This can be used to allow S2K specifiers to be used for the passphrase conversion, to allow other ciphers than IDEA to be used, or to create messages with a mix of conventional ESKs and public key ESKs. This allows a message to be decrypted either with a passphrase or a public key.

3.5.3 String-to-key algorithms

3.5.3.1 Simple S2K algorithm

Simple S2K hashes the passphrase to produce the session key. The manner in which this is done depends on the size of the session key (which will depend on the cipher used) and the size of the hash algorithm's output. If the hash size is greater than or equal to the session key size, the leftmost octets of the hash are used as the key.

If the hash size is less than the key size, multiple instances of the hash context are created -- enough to produce the required key data. These instances are preloaded with 0, 1, 2, ... octets of zeros (that is to say, the first instance has no preloading, the second gets preloaded with 1 octet of zero, the third is preloaded with two octets of zeros, and so forth).

As the data is hashed, it is given independently to each hash context. Since the contexts have been initialized differently, they will each produce different hash output. Once the passphrase is hashed, the output data from the multiple hashes is concatenated, first hash leftmost, to produce the key data, with any excess octets on the right discarded.

3.5.3.2 Salted S2K algorithm

Salted S2K is exactly like Simple S2K, except that the input to the hash function(s) consists of the 8 octets of salt from the S2K specifier, followed by the passphrase.

3.5.3.3 Iterated-Salted S2K algorithm

{{Editor's note: This is very complex, with bizarre things like an 8-bit floating point format. Should we just drop it? --jdcc}}

Iterated-Salted S2K hashes the passphrase and salt data multiple times. The total number of octets to be hashed is encoded in the count octet that follows the salt in the S2K specifier. The count value is stored as a normalized floating-point value with 4 bits of exponent and 4 bits of mantissa. The formula to convert from the count octet to a count of the number of octets to be hashed is as follows, letting the high 4 bits of the count octet be CEXP and the low four bits be CMANT:

count of octets to be hashed = (16 + CMANT) << (CEXP + 6)

This allows encoding hash counts as low as 16 << 6 or 1024 (using an octet value of 0), and as high as 31 << 21 or 65011712 (using an octet value of 0xff). Note that the resulting count value is an octet count of how many octets will be hashed, not an iteration count.

Initially, one or more hash contexts are set up as with the other S2K algorithms, depending on how many octets of key data are needed. Then the salt, followed by the passphrase data is repeatedly hashed until the number of octets specified by the octet count has been hashed. The one exception is that if the octet count is less than the size of the salt plus passphrase, the full salt plus passphrase will be hashed even though that is greater than the octet count. After the hashing is done the data is unloaded from the hash context(s) as with the other S2K algorithms.

4. Packet Syntax

This section describes the packets used by OP.

4.1 Overview

An OP message is constructed from a number of records that are traditionally called packets. A packet is a chunk of data that has a tag specifying its meaning. An OP message, keyring, certificate, and so forth consists of a number of packets. Some of those packets may contain other OP packets (for example, a compressed data packet, when uncompressed, contains OP packets).

Each packet consists of a packet header, followed by the packet body. The packet header is of variable length.

4.2 Packet Headers

The first octet of the packet header is called the "Packet Tag." It determines the format of the header and denotes the packet contents. The remainder of the packet header is the length of the packet.

Note that the most significant bit is the left-most bit, called bit 7. A mask for this bit is 0x80 in hexadecimal.

```
       +---------------+
  PTag |7 6 5 4 3 2 1 0|
       +---------------+
  Bit 7 -- Always one
  Bit 6 -- New packet format if set
```

PGP 2.6.X only uses old format packets. Thus, software that interoperates with those versions of PGP must only use old format packets. If interoperability is not an issue, either format may be used.

Old format packets contain:
  Bits 5-2 -- content tag

Bits 1-0 - length-type

New format packets contain:
Bits 5-0 -- content tag

The meaning of the length-type in old-format packets is:

0 - The packet has a one-octet length. The header is 2 octets long.

1 - The packet has a two-octet length. The header is 3 octets long.

2 - The packet has a four-octet length. The header is 5 octets long.

3 - The packet is of indeterminate length. The header is 1 byte long, and the application must determine how long the packet is. If the packet is in a file, this means that the packet extends until the end of the file. In general, an application should not use indeterminate length packets except where the end of the data will be clear from the context.

New format packets have three possible ways of encoding length. A one-octet Body Length header encodes packet lengths of up to 191 octets, and a two-octet Body Length header encodes packet lengths of **192 to 8383 octets.** For cases where longer packet body lengths are needed, or where the length of the packet body is not known in advance by the issuer, Partial Body Length headers can be used. These are one-octet length headers that encode the length of only part of the data packet.

Each Partial Body Length header is followed by a portion of the packet body data. The Partial Body Length header specifies this portion's length. Another length header (of one of the three types) follows that portion. The last length header in the packet must always be a regular Body Length header. Partial Body Length headers may only be used for the non-final parts of the packet.

A one-octet Body Length header encodes a length of from 0 to 191 octets. This type of length header is recognized because the one octet value is less than 192. The body length is equal to:

```
bodyLen = length_octet;
```

A two-octet Body Length header encodes a length of from 192 to 8383 octets. It is recognized because its first octet is in the range 192 to 223. The body length is equal to:

```
bodyLen = (1st_octet - 192) * 256 + (2nd_octet) + 192
```

A Partial Body Length header is one octet long and encodes a length which is a power of 2, from 1 to 2147483648 (2 to the 31st power). It is recognized because its one octet value is greater than or equal to **224. The partial body length is equal to:**

partialBodyLen = 1 << (length_octet & 0x1f);

Examples:

A packet with length 100 may have its length encoded in one octet: 0x64. This is followed by 100 octets of data.

A packet with length 1723 may have its length coded in two octets: 0xC5, 0xFB. This header is followed by the 1723 octets of data.

A packet with length 100000 might be encoded in the following octet stream: 0xE1, first two octets of data, 0xE0, next one octet of data, 0xEF, next 32768 octets of data, 0xF0, next 65536 octets of data, 0xC5, 0xDD, last 1693 octets of data. This is just one possible encoding, and many variations are possible on the size of the Partial Body Length headers, as long as a regular Body Length header encodes the last portion of the data. Note also that the last Body Length header can be a zero-length header.

Please note that in all of these explanations, the total length of the packet is the length of the header(s) plus the length of the body.

4.3 Packet Tags

The packet tag denotes what type of packet the body holds. Note that old format packets can only have tags less than 16, whereas new format packets can have tags as great as 63. The defined tags (in decimal) are:

0   -- Reserved. A packet must not have a tag with this value.
1   -- Encrypted Session Key Packet
2   -- Signature Packet
3   -- Conventionally Encrypted Session Key Packet
4   -- One-Pass Signature Packet
5   -- Secret Key Packet
6   -- Public Key Packet
7   -- Secret Subkey Packet
8   -- Compressed Data Packet
9   -- Symmetrically Encrypted Data Packet
10   -- Marker Packet
11   -- Literal Data Packet
12   -- Trust Packet
13   -- Name Packet
14   -- Subkey Packet
15   -- Reserved
16   -- Comment Packet
60 to 63 -- Private or Experimental Values

5. Packet Types

5.1 Encrypted Session Key Packets (Tag 1)

An Encrypted Session Key packet holds the key used to encrypt a message that is itself encrypted with a public key. Zero or more Encrypted Session Key packets and/or Conventional Encrypted Session Key packets may precede a Symmetrically Encrypted Data Packet, which holds an encrypted message. The message is encrypted with a session key, and the session key is itself encrypted and stored in the Encrypted Session Key packet or the Conventional Encrypted Session Key packet. The Symmetrically Encrypted Data Packet is preceded by one Encrypted Session Key packet for each OP key to which the message is encrypted. The recipient of the message finds a session key that is encrypted to their public key, decrypts the session key, and then uses the session key to decrypt the message.

The body of this packet consists of:

- A one-octet number giving the version number of the packet type. The currently defined value for packet version is 3. An implementation should accept, but not generate a version of 2, which is equivalent to V3 in all other respects.
- An eight-octet number that gives the key ID of the public key that the session key is encrypted to.
- A one-octet number giving the public key algorithm used.
- A string of octets that is the encrypted session key. This string takes up the remainder of the packet, and its contents are dependent on the public key algorithm used.

Algorithm Specific Fields for RSA encryption
- multiprecision integer (MPI) of RSA encrypted value m**e.

Algorithm Specific Fields for Elgamal encryption:
- MPI of DSA value g**k.
- MPI of DSA value m * y**k.

The encrypted value "m" in the above formulas is derived from the session key as follows. First the session key is prepended with a one-octet algorithm identifier that specifies the conventional encryption algorithm used to encrypt the following Symmetrically Encrypted Data Packet. Then a two-octet checksum is appended which is equal to the sum of the preceding octets, including the algorithm identifier and session key, modulo 65536. This value is then padded as described in PKCS-1 block type 02 [PKCS1] to form the "m" value used in the formulas above.

5.2 Signature Packet (Tag 2)

A signature packet describes a binding between some public key and some data. The most common signatures are a signature of a file or a block of text, and a signature that is a certification of a user ID.

Two versions of signature packets are defined. Version 3 provides basic signature information, while version 4 provides an expandable format with subpackets that can specify more information about the signature. PGP 2.6.X only accepts version 3 signatures.

Implementations MUST accept V3 signatures. Implementations SHOULD generate V4 signatures, unless there is a need to generate a signature that can be verified by PGP 2.6.x.

5.2.1 Version 3 Signature Packet Format

A version 3 Signature packet contains:

- One-octet version number (3).
- One-octet length of following hashed material. MUST be 5.
- One-octet signature type.
- Four-octet creation time.
- Eight-octet key ID of signer.
- One-octet public key algorithm.
- One-octet hash algorithm.
- Two-octet field holding left 16 bits of signed hash value.
- One or more multi-precision integers comprising the signature. This portion is algorithm specific, as described below.

The data being signed is hashed, and then the signature type and creation time from the signature packet are hashed (5 additional octets). The resulting hash value is used in the signature algorithm. The high 16 bits (first two octets) of the hash are included in the signature packet to provide a quick test to reject some invalid signatures.

Algorithm Specific Fields for RSA signatures:

- multiprecision integer (MPI) of RSA signature value m**d.

Algorithm Specific Fields for DSA signatures:

- MPI of DSA value r.
- MPI of DSA value s.

The signature calculation is based on a hash of the signed data, as described above. The details of the calculation are different for DSA signature than for RSA signatures.

With RSA signatures, the hash value is encoded as described in PKCS-1 section 10.1.2, "Data encoding", producing an ASN.1 value of type DigestInfo, and then padded using PKCS-1 block type 01 [PKCS1]. This requires inserting the hash value as an octet string into an ASN.1 structure. The object identifier for the type of hash being used is included in the structure. The hexadecimal representations for the currently defined hash algorithms are:

```
- SHA-1:           0x2b, 0x0e, 0x03, 0x02, 0x1a
- MD5:             0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x02, 0x05
- RIPEMD-160:   0x2b, 0x24, 0x03, 0x02, 0x01
```

The ASN.1 OIDs are:

```
        - MD5:       1.2.840.113549.2.5
        - SHA-1:     1.3.14.3.2.26
        - RIPEMD160: 1.3.36.3.2.1
```

DSA signatures SHOULD use hashes with a size of 160 bits, to match q, the size of the group generated by the DSA key's generator value. The hash function result is treated as a 160 bit number and used directly in the DSA signature algorithm.

5.2.2 Version 4 Signature Packet Format

A version 4 Signature packet contains:

- One-octet version number (4).
- One-octet signature type.
- One-octet public key algorithm.
- One-octet hash algorithm.
- Two-octet octet count for following hashed subpacket data.
- Hashed subpacket data.
- Two-octet octet count for following unhashed subpacket data.
- Unhashed subpacket data.
- Two-octet field holding left 16 bits of signed hash value.
- One or more multi-precision integers comprising the signature. This portion is algorithm specific, as described above.

The data being signed is hashed, and then the signature data from the version number through the hashed subpacket data is hashed. The resulting hash value is what is signed. The left 16 bits of the hash are included in the signature packet to provide a quick test to reject some invalid signatures.

There are two fields consisting of signature subpackets. The first field is hashed with the rest of the signature data, while the second is unhashed. The second set of subpackets is not cryptographically protected by the signature and should include only advisory information.

The algorithms for converting the hash function result to a signature are described above.

5.2.2.1 Signature Subpacket Specification

The subpacket fields consist of zero or more signature subpackets. Each set of subpackets is preceded by a two-octet count of the length of the set of subpackets.

Each subpacket consists of a subpacket header and a body. The header consists of:

- subpacket length (1 or 2 octets):
  Length includes the type octet but not this length,
  1st octet < 192, then length is octet value

```
     1st octet >= 192, then length is 2 octets and equal to
       (1st octet - 192) * 256 + (2nd octet) + 192
   - subpacket type (1 octet):
     If bit 7 is set, subpacket understanding is critical,
```

```
           2 = signature creation time,
           3 = signature expiration time,
           4 = exportable,
           5 = trust signature,
           6 = regular expression,
           7 = revocable,
           9 = key expiration time,
          10 = additional recipient request,
          11 = preferred symmetric algorithms,
          12 = revocation key,
          16 = issuer key ID,
      20 = notation data,
      21 = preferred hash algorithms,
      22 = preferred compression algorithms,
      23 = key server preferences,
      24 = preferred key server

        - subpacket specific data:
```

Bit 7 of the subpacket type is the "critical" bit. If set, it implies that it is critical that the subpacket be one which is understood by the software. If a subpacket is encountered which is marked critical but the software does not understand, the handling depends on the relationship between the issuing key and the key that is signed. If the signature is a valid self-signature (for which the issuer is the key that is being signed, either directly or via a username binding), then the key should not be used. In other cases, the signature containing the critical subpacket should be ignored.

5.2.2.2 Signature Subpacket Types

Several types of subpackets are currently defined. Some subpackets apply to the signature itself and some are attributes of the key. Subpackets that are found on a self-signature are placed on a user name certification made by the key itself. Note that a key may have more than one user name, and thus may have more than one self-signature, and differing subpackets.

Implementing software should interpret a self-signature's preference subpackets as narrowly as possible. For example, suppose a key has two usernames, Alice and Bob. Suppose that Alice prefers the symmetric algorithm CAST5, and Bob prefers IDEA or Triple-DES. If the software locates this key via Alice's name, then the preferred algorithm is CAST5, if software locates the key via Bob's name, then the preferred algorithm is IDEA. If the key is located by key id, then algorithm of the default user name of the key provides the default symmetric algorithm.

The descriptions below describe whether a subpacket is typically found in the hashed or unhashed subpacket sections. If a subpacket is not hashed, then it cannot be trusted.

    Signature creation time (4 octet time field) (Hashed)

The time the signature was made. Always included with new signatures.

Issuer (8 octet key ID) (Non-hashed)

The OP key ID of the key issuing the signature.

Key expiration time (4 octet time field) (Hashed)

The validity period of the key. This is the number of seconds after the key creation time that the key expires. If this is not present or has a value of zero, the key never expires. This is found only on a self-signature.

Preferred symmetric algorithms (array of one-octet values) (Hashed)

Symmetric algorithm numbers that indicate which algorithms the key holder prefers to use. This is an ordered list of octets with the most preferred listed first. It should be assumed that only algorithms listed are supported by the recipient's software. Algorithm numbers in section 6. This is only found on a self-signature.

Preferred hash algorithms (array of one-octet values) (Hashed)

Message digest algorithm numbers that indicate which algorithms the key holder prefers to receive. Like the preferred symmetric algorithms, the list is ordered. Algorithm numbers are in section 6. This is only found on a self-signature.

{{Editor's note: The above preference (hash algs) is controversial. I included it in for symmetry, because if someone wants to build a minimal OP implementation, there needs to be a way to tell someone that you won't be able to verify a signature unless it's made with some set of algorithms. It also permits to prefer DSA with RIPEMD-160, for example. If you have an opinion, please state it.}}

Preferred compression algorithms (array of one-octet values) (Hashed)

Compression algorithm numbers that indicate which algorithms the key holder prefers to use. Like the preferred symmetric algorithms, the list is ordered. Algorithm numbers are in section 6. If this subpacket is not included, ZIP is preferred. A zero denotes that no compression is preferred; the key holder's software may not have compression software. This is only found on a self-signature.

Signature expiration time (4 octet time field) (Hashed)

The validity period of the signature. This is the number of seconds

after the signature creation time that the signature expires. If this
is not present or has a value of zero, it never expires.

    Exportable (1 octet of exportability, 0 for not, 1 for exportable)

(Hashed)

Signature's exportability status. Packet body contains a boolean flag indicating whether the signature is exportable. Signatures which are not exportable are ignored during export and import operations. If this packet is not present the signature is assumed to be exportable.

Revocable (1 octet of revocability, 0 for not, 1 for revocable) (Hashed)

Signature's revocability status. Packet body contains a boolean flag indicating whether the signature is revocable. Signatures which are not revocable get any later revocation signatures ignored. They represent a commitment by the signer that he cannot revoke his signature for the life of his key. If this packet is not present the signature is assumed to be revocable.

Trust signature (1 octet of "level" (depth), 1 octet of trust amount) (Hashed)

Signer asserts that the key is not only valid, but also trustworthy, at the specified level. Level 0 has the same meaning as an ordinary validity signature. Level 1 means that the signed key is asserted to be a valid trusted introducer, with the 2nd octet of the body specifying the degree of trust. Level 2 means that the signed key is asserted to be trusted to issue level 1 trust signatures, i.e. that it is a "meta introducer". Generally, a level n trust signature asserts that a key is trusted to issue level n-1 trust signatures. The trust amount is in a range from 0-255, interpreted such that values less than 120 **indicate partial trust and values of 120 or greater indicate** complete trust. Implementations SHOULD emit values of 60 for partial trust and 120 for complete trust.

Regular expression (null-terminated regular expression) (Hashed)

Used in conjunction with trust signature packets (of level > 0) to limit the scope of trust which is extended. Only signatures by the target key on user IDs which match the regular expression in the body of this packet have trust extended by the trust packet.

Additional recipient request (1 octet of class, 1 octet of algid, 20 octets of fingerprint) (Hashed)

Key holder requests encryption to additional recipient when data is encrypted to this username. If the class octet contains 0x80, then the key holder strongly requests that the additional recipient be added to an encryption. Implementing software may treat this subpacket in any way it sees fit. This is found only on a self-signature.

Revocation key (1 octet of class, 1 octet of algid, 20 octets of fingerprint) (Hashed)

Authorizes the specified key to issue revocation self-signatures on this key. Class octet must have bit 0x80 set, other bits are for future expansion to other kinds of signature authorizations. This is found on a self-signature.

    Notation Data (4 octets of flags, 2 octets of name length,
                   2 octets of value length, M octets of name data,
                   N octets of value data) (Hashed)

This subpacket describes a "notation" on the signature that the issuer wishes to make. The notation has a name and a value, each of which are strings of octets. There may be more than one notation in a signature. Notations can be used for any extension the issuer of the signature cares to make. The "flags" field holds four octets of flags. All undefined flags MUST be zero. Defined flags are:

        First octet: 0x80 = human-readable. This note is text, a note
                            from one person to another, and has no
                            meaning to software.
        Other octets: none.

    Key server preferences (N octets of flags) (Hashed)

This is a list of flags that indicate preferences that the key holder has about how the key is handled on a key server. All undefined flags MUST be zero.

       First octet: 0x80 = No-modify -- the key holder requests that
                           this key only be modified or updated by the
                           key holder or an authorized administrator of
                           the key server.

This is found only on a self-signature.

    Preferred key server (String) (Hashed)

This is a URL of a key server that the key holder prefers be used for updates. Note that keys with multiple user names can have a preferred key server for each user name. This is found only on a self-signature.

Implementations SHOULD implement a "preference" and MAY implement a "request."

{{Editor's note: None of the preferences have a way to specify a negative preference (for example, I like Triple-DES, don't use algorithm X). Tacitly, the absence of an algorithm from a set is a negative preference, but should there be an explicit way to give a negative preference? -jdcc}}

{{Editor's note: A missing feature is to invalidate (or revoke) a user id, rather than the entire key. Lots of people want this, and many people have keys cluttered with old work email addresses. There is another related issue, that that is with key rollover -- suppose I'm retiring an old key, but I don't want to have to lose all my certification signatures. It would be nice if there were a way for a key to transfer itself to a new one. Lastly, if either (or both) of these is desirable, do we handle them with a new signature type, or with notations, which are an extension mechanism. I think that it makes sense to make a revocation type (because it's analogous to the other forms of revocation), but rollover might be best implemented as an extension. --jdcc}}

{{Editor's note: PGP 3 designed, but never implemented a number of other subpacket types. They were: A signature version number; A set of key usage flags (signing key, encryption key for communication, and encryption key for storage); User ID of the signer; Policy URL; net location of the key.

Some of these options are things the WG has talked about as being a Good Thing -- like flags denoting if a key is a comm key or a storage key. My design of such a feature would be different than the other one, though. I think it would be a great idea to have a URL that's a location to find the key, so people who prefer to have a web, ftp, or finger location can use those. However, some of them (like a URL) are also perfect for designing in with extensions. After all, we only have 128 **subpacket constants.**

--jdcc}}

5.2.3 Signature Types

There are a number of possible meanings for a signature, which are specified in a signature type octet in any given signature. These meanings are:

- 0x00: Signature of a binary document.

Typically, this means the signer owns it, created it, or certifies that it has not been modified.

- 0x01: Signature of a canonical text document.

Typically, this means the signer owns it, created it, or certifies that it has not been modified. The signature will be calculated over the textual data with its line endings converted to <CR><LF>.

- 0x02: Standalone signature.

This signature is a signature of only its own subpacket contents. It is calculated identically to a signature over a zero-length binary document.

- 0x10: The generic certification of a User ID and Public Key packet.

The issuer of this certification does not make any particular assertion as to how well the certifier has checked that the owner of the key is in fact the person described by the user ID. Note that all PGP "key signatures" are this type of certification.

- 0x11: This is a persona certification of a User ID and Public Key packet.

It means that the issuer of this certification has not done any verification of the claim that the owner of this key is the user ID specified. Note that no released version of PGP has generated this type of certification.

- 0x12: This is the casual certification of a User ID and Public Key packet.

It means that the issuer of this certification has done some casual verification of the claim of identity. Note that no version of PGP has generated this type of certification, nor is there any definition of what constitutes a casual certification.

- 0x13: This is the positive certification of a User ID and Public Key packet.

It means that the issuer of this certification has done substantial verification of the claim of identity. Note that no version of PGP has generated this type of certification, nor is there any definition of what constitutes a positive certification. Please also note that the vagueness of these certification systems is not a flaw, but a feature of the system. Because PGP places final authority for validity upon the receiver of a certification, it may be that one authority's casual certification might be more rigorous than some other authority's positive certification.

{{Editor's note: While there is a scale of identification signatures in the range 0x10 to 0x13, most of them have never been implemented or used. Current implementations only use 0x10, the "generic certification." Should the others be removed? RFC 1991 went to some trouble to explain which ones were defined but not implemented, or read but not generated. I think we should not do that. If we define them, they should be MAY features at the very least. If we're not going to use them, they shouldn't be in the spec. --jdcc}}

- 0x18: This is used for a signature by a signature key to bind a subkey which will be used for encryption.

The signature is calculated directly on the subkey itself, not on any User ID or other packets.

- 0x20: This signature is used to revoke a key.

The signature is calculated directly on the key being revoked. A revoked key is not to be used. Only revocation signatures by the key being revoked, or by an authorized revocation key, should be considered.

- 0x28: This is used to revoke a subkey.

The signature is calculated directly on the subkey being revoked. A revoked subkey is not to be used. Only revocation signatures by the top-level signature key which is bound to this subkey, or by an authorized revocation key, should be considered.

- 0x30: This signature revokes an earlier user ID certification signature (signature class 0x10 - 0x13).

It should be issued by the same key which issued the revoked signature, and should have a later creation date.

- 0x40: Timestamp signature.

{{Editor's note: The timestamp signature is left over from RFC 1991, and has never been fully designed nor implemented. Is this the sort of thing best handled by notations? --jdcc}}

{{Editor's note: It would be nice to have a signature that applied to the key alone, rather than a key plus a user name. Perhaps this is best done with a notation. --jdcc}}

{{Editor's note: There is presently no way for a key-signer (a.k.a. certifier) to sign a main key along with a subkey. There are a number of useful situations for a set of keys (main plus subkeys) to all be signed together. How do we solve this? --jdcc}}

5.3 Conventional Encrypted Session-Key Packets (Tag 3)

The Conventional Encrypted Session Key packet holds the conventional-cipher encryption of a session key used to encrypt a message. Zero or more Encrypted Session Key packets and/or Conventional Encrypted Session Key packets may precede a Symmetrically Encrypted Data Packet that holds an encrypted message. The message is encrypted with a session key, and the session key is itself encrypted and stored in the Encrypted Session Key packet or the Conventional Encrypted Session Key packet.

If the Symmetrically Encrypted Data Packet is preceded by one or more Conventional Encrypted Session Key packets, each specifies a passphrase which may be used to decrypt the message. This allows a message to be encrypted to a number of public keys, and also to one or more pass phrases. This packet type is new, and is not generated by PGP 2.x or PGP 5.0.

The body of this packet consists of:

- A one-octet version number. The only currently defined version is 4.
- A one-octet number describing the symmetric algorithm used.
- A string-to-key (S2K) specifier, length as defined above.
- Optionally, the encrypted session key itself, which is decrypted with the string-to-key object.

If the encrypted session key is not present (which can be detected on the basis of packet length and S2K specifier size), then the S2K algorithm applied to the passphrase produces the session key for decrypting the file, using the symmetric cipher algorithm from the Conventional Encrypted Session Key packet.

If the encrypted session key is present, the result of applying the S2K algorithm to the passphrase is used to decrypt just that encrypted session key field, using CFB mode with an IV of all zeros. The decryption result consists of a one-octet algorithm identifier that specifies the conventional encryption algorithm used to encrypt the following Symmetrically Encrypted Data Packet, followed by the session key octets themselves.

Note: because an all-zero IV is used for this decryption, the S2K specifier MUST use a salt value, either a a Salted S2K or an Iterated-Salted S2K. The salt value will insure that the decryption key is not repeated even if the passphrase is reused.

5.4 One-Pass Signature Packets (Tag 4)

The One-Pass Signature packet precedes the signed data and contains enough information to allow the receiver to begin calculating any hashes needed to verify the signature. It allows the Signature Packet to be placed at the end of the message, so that the signer can compute the entire signed message in one pass.

A One-Pass Signature does not interoperate with PGP 2.6.x or earlier.

The body of this packet consists of:

- A one-octet version number. The current version is 3.
- A one-octet signature type. Signature types are described in section 5.2.3.
- A one-octet number describing the hash algorithm used.
- A one-octet number describing the public key algorithm used.
- An eight-octet number holding the key ID of the signing key.
- A one-octet number holding a flag showing whether the signature

is nested. A zero value indicates that the next packet is another One-Pass Signature packet which describes another signature to be applied to the same message data.

5.5 Key Material Packet

A key material packet contains all the information about a public or private key. There are four variants of this packet type, and two major versions. Consequently, this section is complex.

5.5.1 Key Packet Variants

5.5.1.1 Public Key Packet (Tag 6)

A Public Key packet starts a series of packets that forms an OP key (sometimes called an OP certificate).

5.5.1.2 Public Subkey Packet (Tag 14)

A Public Subkey packet (tag 14) has exactly the same format as a Public Key packet, but denotes a subkey. One or more subkeys may be associated with a top-level key. By convention, the top-level key

provides signature services, and the subkeys provide encryption
services.

Note: in PGP 2.6.X, tag 14 was intended to indicate a comment packet.

This tag was selected for reuse because no previous version of PGP ever emitted comment packets but they did properly ignore them. Public Subkey packets are ignored by PGP 2.6.X and do not cause it to fail, providing a limited degree of backwards compatibility.

5.5.1.3 Secret Key Packet (Tag 5)

A Secret Key packet contains all the information that is found in a Public Key packet, including the public key material, but also includes the secret key material after all the public key fields.

5.5.1.4 Secret Subkey Packet (Tag 7)

A Secret Subkey packet (tag 7) is the subkey analog of the Secret Key packet, and has exactly the same format.

5.5.2 Public Key Packet Formats

There are two versions of key-material packets. Version 3 packets were first generated PGP 2.6. Version 2 packets are identical in format to Version 3 packets, but are generated by PGP 2.5 or before. PGP 5.0 introduces version 4 packets, with new fields and semantics. PGP 2.6.X will not accept key-material packets with versions greater than 3.

OP implementations SHOULD create keys with version 4 format. An implementation MAY generate a V3 key to ensure interoperability with old software; note, however, that V4 keys correct some security deficiencies in V3 keys. These deficiencies are described below. An implementation MUST NOT create a V3 key with a public key algorithm other than RSA.

A version 3 public key or public subkey packet contains:

- A one-octet version number (3).
- A four-octet number denoting the time that the key was created.
- A two-octet number denoting the time in days that this key is valid. If this number is zero, then it does not expire.
- A one-octet number denoting the public key algorithm of this key
- A series of multi-precision integers comprising the key material:
- a multiprecision integer (MPI) of RSA public modulus n;
- an MPI of RSA public encryption exponent e.

The fingerprint of the key is formed by hashing the body (but not the two-octet length) of the MPIs that form the key material (public modulus n, followed by exponent e) with MD5.

The eight-octet key ID of the key consists of the low 64 bits of the public modulus of an RSA key.

Since the release of V3 keys, there have been a number of improvements
desired in the key format. For example, if the key ID is a function of
the public modulus, it is easy for a person to create a key that has
the same key ID as some existing key. Similarly, MD5 is no longer the

preferred hash algorithm, and not hashing the length of an MPI with its body increases the chances of a fingerprint collision.

The version 4 format is similar to the version 3 format except for the absence of a validity period. This has been moved to the signature packet. In addition, fingerprints of version 4 keys are calculated differently from version 3 keys, as described elsewhere.

A version 4 packet contains:

- A one-octet version number (4).
- A four-octet number denoting the time that the key was created.
- A one-octet number denoting the public key algorithm of this key
- A series of multi-precision integers comprising the key material. This algorithm-specific portion is:

Algorithm Specific Fields for RSA public keys:

- multiprecision integer (MPI) of RSA public modulus n;
- MPI of RSA public encryption exponent e.

Algorithm Specific Fields for DSA public keys:

- MPI of DSA prime p;
- MPI of DSA group order q (q is a prime divisor of p-1);
- MPI of DSA group generator g;
- MPI of DSA public key value y (= g**x where x is secret).

Algorithm Specific Fields for Elgamal public keys:

- MPI of Elgamal prime p;
- MPI of Elgamal group generator g;
- MPI of Elgamal public key value y (= g**x where x is secret).

5.5.3 Secret Key Packet Formats

The Secret Key and Secret Subkey packets contain all the data of the Public Key and Public Subkey packets, with additional algorithm-specific secret key data appended, in encrypted form.

The packet contains:

- A Public Key or Public Subkey packet, as described above
- One octet indicating string-to-key usage conventions. 0 indicates that the secret key data is not encrypted. 255 indicates that a string-to-key specifier is being given. Any other value is a conventional encryption algorithm specifier.
  - [Optional] If string-to-key usage octet was 255, a one-octet conventional encryption algorithm.
  - [Optional] If string-to-key usage octet was 255, a string-to-key specifier. The length of the string-to-key specifier is implied by its type, as described above.

- [Optional] If secret data is encrypted, eight-octet Initial Vector (IV).
- Encrypted multi-precision integers comprising the secret key data. These algorithm-specific fields are as described below.

- Two-octet checksum of the plaintext of the algorithm-specific portion (sum of all octets, mod 65536).

Algorithm Specific Fields for RSA secret keys:
- multiprecision integer (MPI) of RSA secret exponent d.
- MPI of RSA secret prime value p.
- MPI of RSA secret prime value q (p < q).
- MPI of u, the multiplicative inverse of p, mod q.

Algorithm Specific Fields for DSA secret keys:
- MPI of DSA secret exponent x.

Algorithm Specific Fields for Elgamal secret keys:
- MPI of Elgamal secret exponent x.

Secret MPI values can be encrypted using a passphrase. If a string-to-key specifier is given, that describes the algorithm for converting the passphrase to a key, else a simple MD5 hash of the passphrase is used. Implementations SHOULD use a string-to-key specifier; the simple hash is for backwards compatibility. The cipher for encrypting the MPIs is specified in the secret key packet.

Encryption/decryption of the secret data is done in CFB mode using the key created from the passphrase and the Initial Vector from the packet. A different mode is used with RSA keys than with other key formats. With RSA keys, the MPI bit count prefix (i.e., the first two octets) is not encrypted. Only the MPI non-prefix data is encrypted. Furthermore, the CFB state is resynchronized at the beginning of each new MPI value, so that the CFB block boundary is aligned with the start of the MPI data.

With non-RSA keys, a simpler method is used. All secret MPI values are encrypted in CFB mode, including the MPI bitcount prefix.

The 16-bit checksum that follows the algorithm-specific portion is the algebraic sum, mod 65536, of the plaintext of all the algorithm-specific octets (including MPI prefix and data). With RSA keys, the checksum is stored in the clear. With non-RSA keys, the checksum is encrypted like the algorithm-specific data. This value is used to check that the passphrase was correct.

5.6 Compressed Data Packet (Tag 8)

The Compressed Data packet contains compressed data. Typically, this packet is found as the contents of an encrypted packet, or following a Signature or One-Pass Signature packet, and contains literal data packets.

The body of this packet consists of:

- One octet that gives the algorithm used to compress the packet.
- The remainder of the packet is compressed data.

A Compressed Data Packet's body contains an RFC1951 DEFLATE block that compresses some set of packets. See section 7 for details on how messages are formed.

5.7 Symmetrically Encrypted Data Packet (Tag 9)

The Symmetrically Encrypted Data packet contains data encrypted with a conventional (symmetric-key) algorithm. When it has been decrypted, it will typically contain other packets (often literal data packets or compressed data packets).

The body of this packet consists of:

- Encrypted data, the output of the selected conventional cipher operating in PGP's variant of Cipher Feedback (CFB) mode.

The conventional cipher used may be specified in an Encrypted Session Key or Conventional Encrypted Session Key packet which precedes the Symmetrically Encrypted Data Packet. In that case, the cipher algorithm octet is prepended to the session key before it is encrypted. If no packets of these types precede the encrypted data, the IDEA algorithm is used with the session key calculated as the MD5 hash of the passphrase.

The data is encrypted in CFB mode, with a CFB shift size equal to the cipher's block size [Ref]. The Initial Vector (IV) is specified as all zeros. Instead of using an IV, OP prepends a 10 octet string to the data before it is encrypted. The first eight octets are random, and the 9th and 10th octets are copies of the 7th and 8th octets, respectivelly. After encrypting the first 10 octets, the CFB state is resynchronized if the cipher block size is 8 octets or less. The last

8 octets of ciphertext are passed through the cipher and the block
boundary is reset.

The repetition of 16 bits in the 80 bits of random data prepended to the message allows the receiver to immediately check whether the session key is correct.

5.8 Marker Packet (Obsolete Literal Packet) (Tag 10)

An experimental version of PGP used this packet as the Literal packet, but no released version of PGP generated Literal packets with this tag. With PGP 5.x, this packet has been re-assigned and is reserved for use as the Marker packet.

The body of this packet consists of:
- The three octets 0x60, 0x47, 0x60 (which spell "PGP" in UTF-8).

Such a packet should be ignored on input. It may be placed at the beginning of a message that uses features not available in PGP 2.6.X in order to cause that version to report that newer software necessary to process the message.

5.9 Literal Data Packet (Tag 11)

A Literal Data packet contains the body of a message; data that is not to be further interpreted.

The body of this packet consists of:

- A one-octet field that describes how the data is formatted.

If it is a 'b' (0x62), then the literal packet contains binary data. If it is a 't' (0x74), then it contains text data, and thus may need line ends converted to local form, or other text-mode changes. RFC 1991 also defined a value of 'l' as a 'local' mode for machine-local conversions. This use is now deprecated.

- File name as a string (one-octet length, followed by file name), if the encrypted data should be saved as a file.

If the special name "_CONSOLE" is used, the message is considered to be "for your eyes only". This advises that the message data is unusually sensitive, and the receiving program should process it more carefully, perhaps avoiding storing the received data to disk, for example.

- A four-octet number that indicates the modification date of the file, or the creation time of the packet, or a zero that indicates the present time.

- The remainder of the packet is literal data.

Text data is stored with <CR><LF> text endings. This should be converted to native line endings by the receiving software.

5.10 Trust Packet (Tag 12)

The Trust packet is used only within keyrings and is not normally exported. Trust packets contain data that record the user's specifications of which key holders are trustworthy introducers, along with other information that implementing software uses for trust information.

Trust packets SHOULD NOT be emitted to output streams that are transferred to other users, and they SHOULD be ignored on any input other than local keyring files.

{{Editor's note: I have brushed aside the description of the old PGP trust packets for a number of reasons. They are context dependent; their meaning depends on the packet preceding them in a keyring.

There is also a security problem with trust packets. For example, malicious software can write a new public key into a user's key ring with trust packets that make it trusted.

A number of us have discussed this problem, and think that trust information should always be self-signed to act as an integrity check, but other people may have other solutions.

My solution is to make trust packets implementation dependent. They are not emitted on export and ignored on import. Because of this, they are arguably out of scope of this document anyway. Given that the PGP implementation of trust packets has security flaws, this seems to be the best way to deal with them.

--jdcc}}

5.11 User ID Packet (Tag 13)

A User ID packet consists of data which is intended to represent the name and email address of the key holder. By convention, it includes an RFC822 mail name, but there are no restrictions on its content. The packet length in the header specifies the length of the user name. If it is text, it is encoded in UTF-8.

{{Editor's note: PRZ thinks there should be more types of "user ids" other than the traditional name, such as photos, and so on. The above definition, which assiduously avoids saying that the content of the packet is a counted string, is one potential way to handle it. Another would be to explicitly state that this packet is a string, and introduce a free-form user identification packet.

A related issue with this document is that sometimes it says "user id" and sometimes "user name." We need some work here. Present plan is to use "User ID" everywhere. --jdcc}}

{{Editor's note: Carl Ellison pointed out to me that if we have non-exportable (local to one's own keyring) usernames that I can assign to keys I use, then essentially we have SDSI naming in PGP. This is a Good Thing, in my opinion, but we have to have a way to define it. --jdcc}}

5.12 Comment Packet (Tag 16)

A Comment packet is used for holding data that is not relevant to software. Comment packets should be ignored.

{{Editor's note: should? Must? What does it mean to ignore them? For example, if it's desirable to show a comment to a user, then how does that interact with should/must and a suitable definition of "ignore." I believe that they MUST be ignored, but displaying them to a user is ignoring them. Looking inside them for cryptographic content (like OP packets) is *not* ignoring them.}}

{{Editor's note: should we put in an X.509 encapsulation packet type?}}

6. Constants

This section describes the constants used in OP.

Note that these tables are not exhaustive lists; an implementation MAY implement an algorithm not on these lists.

6.1 Public Key Algorithms

1 - RSA (Encrypt or Sign)
2 - RSA Encrypt-Only
3 - RSA Sign-Only
16 - Elgamal
17 - DSA (Digital Signature Standard)
100 to 110 - Private/Experimental algorithm.

Implementations MUST implement DSA for signatures, and Elgamal for encryption. Implementations SHOULD implement RSA encryption. Implementations MAY implement any other algorithm.

{{Editor's note: reserve an algorithm for elliptic curve? Note that I've left Elgamal signatures completely unmentioned. I think this is good. --jdcc}}

6.2 Symmetric Key Algorithms

0 - Plaintext
1 - IDEA
2 - Triple-DES (DES-EDE, as per spec - 168 bit key derived from 192)
3 - CAST5 (128 bit key)
4 - Blowfish (128 bit key)
5 - ROT-N (128 bit N)
6 - SAFER-SK128
7 - DES/SK
100 to 110 - Private/Experimental algorithm.

Implementations MUST implement Triple-DES. Implementations SHOULD implement IDEA and CAST5.Implementations MAY implement any other algorithm.

6.3 Compression Algorithms

0 - Uncompressed
1 - ZIP
100 to 110 - Private/Experimental algorithm.

Implementations MUST implement uncompressed data. Implementations SHOULD implement ZIP.

6.4 Hash Algorithms

1 - MD5
2 - SHA-1
3 - RIPE-MD/160
4 - HAVAL

100 to 110 - Private/Experimental algorithm.

Implementations MUST implement SHA-1. Implementations SHOULD implement MD5.

7. Packet Composition

OP packets may be assembled into sequences in order to create messages and transfer keys. Not all possible packet sequences are meaningful and correct. This describes the rules for how packets should be placed into sequences.

7.1 Transferable Public Keys

OP users may transfer public keys. The essential elements of a transferable public key are:

- One Public Key packet
- Zero or more revocation signatures
- One or more User ID packets
- After each User ID packet, zero or more Signature packets
- Zero or more Subkey packets
- After each Subkey packet, one or more Signature packets

The Public Key packet occurs first. Each of the following User ID packets provides the identity of the owner of this public key. If there are multiple User ID packets, this corresponds to multiple means of identifying the same unique individual user; for example, a user may enjoy the use of more than one e-mail address, and construct a User ID packet for each one.

Immediately following each User ID packet, there are zero or more signature packets. Each signature packet is calculated on the immediately preceding User ID packet and the initial Public Key packet. The signature serves to certify the corresponding public key and user ID. In effect, the signer is testifying to his or her belief that this public key belongs to the user identified by this user ID.

After the User ID packets there may be one or more Subkey packets. Subkeys are used in cases where the top-level public key is a signature-only key. The subkeys are then encryption-only keys that are bound to the signature key. Each Subkey packet must be followed by at least one Signature packet, which should be of the subkey binding signature type, and issued by the top level key.

{{Editor's note: I think it is a good idea to have signature-only subkeys, too (or even encrypt-and-sign subkeys), but no implementation does this. Should we generalize here? --jdcc}}

Subkey and Key packets may each be followed by a revocation Signature packet to indicate that the key is revoked. Revocation signatures are only accepted if they are issued by the key itself, or by a key which is authorized to issue revocations via a revocation key subpacket in a self-signature by the top level key.

Transferable public key packet sequences may be concatenated to allow transferring multiple public keys in one operation.

7.2 OP Messages

An OP message is a packet or sequence of packets that corresponds to the following grammatical rules (comma represents sequential composition, and vertical bar separates alternatives):

```
   OP Message :- Encrypted Message | Signed Message | Compressed Message
                                   | Literal Message.

   Compressed Message :- Compressed Data Packet.

   Literal Message :- Literal Data Packet.

   ESK :- Encrypted Session Key Packet |
          Conventionally Encrypted Session Key Packet.

   ESK Sequence :- ESK | ESK Sequence, ESK.

   Encrypted Message :- Symmetrically Encrypted Data Packet |
                        ESK Sequence, Symmetrically Encrypted Data Packet.

   One-Pass Signed Message :- One-Pass Signature Packet, OP Message,
                                Signature Packet.

   Signed Message :- Signature Packet, OP Message |
                        One-Pass Signed Message.
```

In addition, the decrypting a Symmetrically Encrypted Data packet and decompressing a Compressed Data packet must yield a valid OP Message.

8. Enhanced Key Formats

8.1 Key Structures

The format of V3 OP key using RSA is as follows. Entries in square brackets are optional and ellipses indicate repetition.

```
    RSA Public Key
       [Revocation Self Signature]
        User ID [Signature ...]
       [User ID [Signature ...] ...]
```

Each signature certifies the RSA public key and the preceding user ID. The RSA public key can have many user IDs and each user ID can have many signatures.

The format of an OP V4 key that uses two public keys is very similar

except that the second key is added to the end as a 'subkey' of the
primary key.

    Primary-Key

```
       [Revocation Self Signature]
        User ID [Signature ...]
       [User ID [Signature ...] ...]
       [Subkey Primary-Key-Signature]
```

The subkey always has a single signature after it that is issued using the primary key to tie the two keys together. The new format can use either the new signature packets or the old signature packets.

In an Elgamal/DSA key, the DSA public key is the primary key, the Elgamal public key is the subkey, and either version 3 or 4 of the signature packet can be used. There may be other types of V4 keys, too. For example, there may be a single-key RSA key in V4 format, a DSA primary key with an RSA encryption key, etc, or RSA primary key with an Elgamal subkey.

It is also possible to have a signature-only subkey. This permits a primary key that collects certifications (key signatures) but is used only used for certifying subkeys that are used for encryption and signatures.

8.2 V4 Key IDs and Fingerprints

A V4 fingerprint is the 160-bit SHA-1 hash of the one-octet Packet Tag, followed by the two-octet packet length, followed by the entire Public Key packet starting with the version field. The key ID is either the low order 32 bits or 64 bits of the fingerprint. Here are the fields of the hash material, with the example of a DSA key:

```
    a.1) 0x99 (1 byte)
    a.2) high order length byte of (b)-(f) (1 byte)
    a.3) low order length byte of (b)-(f) (1 byte)
    b) version number = 4 (1 byte);
    c) time stamp of key creation (4 bytes);
    e) algorithm (1 byte):
         17 = DSA;
    f) Algorithm specific fields.

    Algorithm Specific Fields for DSA keys (example):
    f.1) MPI of DSA prime p;
    f.2) MPI of DSA group order q (q is a prime divisor of p-1);
    f.3) MPI of DSA group generator g;
    f.4) MPI of DSA public key value y (= g**x where x is secret).
```

9. Security Considerations

As with any technology involving cryptography, you should check the current literature to determine if any algorithms used here have been found to be vulnerable to attack.

This specification uses Public Key Cryptography technologies.

Possession of the private key portion of a public-private key pair is assumed to be controlled by the proper party or parties.

Certain operations in this specification involve the use of random numbers. An appropriate entropy source should be used to generate these numbers. See RFC 1750.

The MD5 hash algorithm has been found to have weaknesses (pseudo-collisions in the compress function) that make some people deprecate its use. They consider the SHA-1 algorithm better.

If you are building an authentication system, the recipient may specify a preferred signing algorithm. However, the signer would be foolish to use a weak algorithm simply because the recipient requests it.

Some of the encryption algorithms mentioned in this document have been analyzed less than others. For example, although CAST5 is presently considered strong, it has been analyzed less than Triple-DES. Other algorithms may have other controversies surrounding them.

Some technologies mentioned here may be subject to government control in some countries.

10. Authors and Working Group Chair

The working group can be contacted via the current chair:

John W. Noerenberg, II Qualcomm, Inc 6455 Lusk Blvd San Diego, CA 92131 USA Email: jwn2@qualcomm.com Tel: +1 619 658 3510

The principal authors of this draft are (in alphabetical order):

Jon Callas Pretty Good Privacy, Inc. 555 Twin Dolphin Drive, #570 Redwood Shores, CA 94065, USA Email: jon@pgp.com Tel: +1-650-596-1960

Lutz Donnerhacke IKS GmbH Wildenbruchstr. 15 07745 Jena, Germany EMail: lutz@iks-jena.de Tel: +49-3641-675642

Hal Finney Pretty Good Privacy, Inc. 555 Twin Dolphin Drive, #570 Redwood Shores, CA 94065, USA Email: hal@pgp.com Tel: +1-650-572-0430

Rodney Thayer Sable Technology Corporation 246 Walnut Street Newton, MA 02160 USA Email: rodney@sabletech.com Tel: +1-617-332-7292

This draft also draws on much previous work from a number of other authors who include: Derek Atkins, Charles Breed, Dave Del Torto, Marc Dykstehouse, Gail Haspert, Gene Hoffman, Paul Hoffman, Raph Levine, Colin Plumb, Will Price, William Stallings, Mark Weaver, and Philip R. Zimmermann.

11. References

[CAMPBELL} Campbell, Joe, "C Programmer's Guide to Serial Communications"

[DONNERHACKE] Donnerhacke, L., et. al, "PGP263in - an improved international version of PGP", ftp://ftp.iks-jena.de/mitarb/lutz/crypt/software/pgp/

[ISO-10646] ISO/IEC 10646-1:1993. International Standard -- Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. UTF-8 is described in Annex R, adopted but not yet published. UTF-16 is described in Annex Q, adopted but not yet published.

[PKCS1] RSA Laboratories, "PKCS #1: RSA Encryption Standard," version 1.5, November 1993

[RFC822] D. Crocker, "Standard for the format of ARPA Internet text messages", RFC 822, August 1982

[RFC1423] D. Balenson, "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, October 1993

[RFC1641] Goldsmith, D., and M. Davis, "Using Unicode with MIME", RFC 1641, Taligent inc., July 1994.

[RFC1750] Eastlake, Crocker, & Schiller., Randomness Recommendations for Security. December 1994.

[RFC1951] Deutsch, P., DEFLATE Compressed Data Format Specification version 1.3. May 1996.

[RFC1983] G. Malkin., Internet Users' Glossary. August 1996.

[RFC1991] Atkins, D., Stallings, W., and P. Zimmermann, "PGP Message Exchange Formats", RFC 1991, August 1996.

[RFC2015] Elkins, M., "MIME Security with Pretty Good Privacy (PGP)", RFC 2015, October 1996.

[RFC2044] F. Yergeau., UTF-8, a transformation format of Unicode and ISO 10646. October 1996.

[RFC2045] Borenstein, N., and Freed, N., "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.", November 1996

[RFC2119] Bradner, S., Key words for use in RFCs to Indicate

Requirement Level. March 1997.

12. Full Copyright Statement