File Formats Used by PGP 2.6

Philip Zimmermann

22 May 94

Note: packets that contain a version byte of 2 will contain a version byte of 3 when using versions of PGP >= 2.6 after 9/1/94.

This appendix describes the file formats used externally by Pretty Good Privacy (PGP), the RSA public key cryptography application. The intended audience includes software engineers trying to port PGP to other hardware environments or trying to implement other PGP- compatible cryptography products, or anyone else who is curious.

[To be included: a description of ASCII armor. An ASCII armored file is just like a binary file described here, but with an extra layer of encoding added, framing lines, and a 24-bit CRC at the end.]

1 Data represeentation

1.1 Byte order

All integer data used by PGP is externally stored most significant byte (MSB) first, regardless of the byte order used internally by the host CPU architecture. This is for portability of messages and keys between hosts. This covers multiprecision RSA integers, bit count prefix fields, byte count prefix fields, checksums, key IDs, and timestamps.

The MSB-first byte order for external packet representation was chosen only because many other crypto standards use it.

1.2 Multiprecision integers

RSA arithmetic involves a lot of multiprecision integers, often having hundreds of bits of precision. PGP externally stores a multiprecision integer (MPI) with a 16-bit prefix that gives the number of significant bits in the integer that follows. The integer that follows this bitcount field is stored in the usual byte order, with the MSB padded with zero bits if the bitcount is not a multiple of 8. The bitcount always specifies the exact number of significant bits. For example, the integer value 5 would be stored as these three bytes:

```
00 03 05
```

An MPI with a value of zero is simply stored with the 16-bit bitcount prefix field containing a 0, with no value bytes following it.

1.3 Key ID

Some packets use a *key ID* field. The key ID is the least significant 64 bits of the RSA public modulus that was involved in creating the packet. For all practical purposes it unique to each RSA public key.

1.4 User ID

Some packets contain a *user ID*, which is an ASCII string that contains the user's name. Unlike a C string, the user ID has a length byte at the beginning that has a byte count of the rest of the string. This length byte does not include itself in the count.

1.5 Timestamp

Some packets contain a timestamp, which is a 32-bit unsigned integer of the number of seconds elapsed since 1970 Jan 1 00:00:00 GMT. This is the standard format used by Unix timestamps. It spans 136 years.

1.6 Cipher type byte (CTB)

Many of these data structures begin with a Cipher Type Byte (CTB), which specifies the type of data structure that follows it. The CTB bit fields have the following meaning (bit 0 is the LSB, bit 7 is the MSB):

```
Bit 7:     Always 1, which designates this as a CTB
Bit 6:     Reserved.
Bits 5-2:  CTB type field, specifies type of packet that follows
           0001 - public-key-encrypted packet
           0010 - secret-key-encrypted (signature) packet
           0101 - Secret key certificate
           0110 - Public key certificate
           1000 - Compressed data packet
           1001 - Conventional-Key-Encrypted data
           1011 - Raw literal plaintext data, with filename and mode
           1100 - Keyring trust packet
           1101 - User ID packet, associated with public or secret key
           1110 - Comment packet
           Other CTB packet types are unimplemented.
Bits 1-0:  Length-of-length field:
           00 - 1 byte packet length field follows CTB
           01 - 2 byte packet length field follows CTB
           10 - 4 byte packet length field follows CTB
           11 - no length field follows CTB, unknown packet length.
           The 8-, 16-, or 32-bit packet length field after the CTB
           gives the length in bytes of the rest of the packet, not
           counting the CTB and the packet length field.
```

1.7 RSA public-key-encrypted packet

```
Offset  Length  Meaning
0       1       CTB for RSA public-key-encrypted packet
1       2       16-bit (or maybe 8-bit) length of packet
3       1       Version byte (=2).  May affect rest of fields that follow.
4       8       64-bit Key ID
12      1       Algorithm byte for RSA (=1 for RSA).
                --Algorithm byte affects field definitions that follow.
13      ?       RSA-encrypted integer, encrypted conventional key
                packet.  (MPI with bitcount prefix)
```

The conventionally-encrypted ciphertext packet begins right after the RSA public-key-encrypted packet that contains the conventional key.

1.8 Signature packet

Offset	Length	Meaning
0	1	CTB for secret-key-encrypted (signed) packet
1	2	16-bit (or maybe 8-bit) length of packet
3	1	Version byte (=2). May affect rest of fields that follow. Version byte (=3) for >= PGP2.6 after 9/1/94
4	1	Length of following material that is implicitly included in MD calculation (=5).
5	1	Signature classification field (see below). Implicitly append this to message for MD calculation.
6	4	32-bit timestamp of when signature was made. Implicitly append this to message for MD calculation.
10	8	64-bit Key ID
18	1	Algorithm byte for public key scheme (RSA=0x01). --Algorithm byte affects field definitions that follow.
19	1	Algorithm byte for message digest (MD5=0x01).
20	2	First 2 bytes of the Message Digest inside the RSA-encrypted integer, to help us figure out if we used the right RSA key to check the signature.
22	?	RSA-encrypted integer, encrypted message digest (MPI with bitcount prefix).

If the plaintext that was signed is included in the same file as the signature packet, it begins right after the RSA secret-key-signed packet that contains the message digest. The plaintext has a *literal* CTB prefix.

The original idea had a variable length field following the length of following material byte, before the Key ID. In particular, the possibility of a 2-byte validity period was defined, although no previous version of PGP ever generated those bytes.

Owing to the way the MD5 is computed for the signature, if that field is variable length, it is possible to generate two different messages with the same MD5 hash. One would be a file of length N, with a 7-byte following section consisting of a signature type byte, 4 bytes of timestamp, and 2 of validity period, while the other would be a file of length N+2, whose last two bytes would be the siganture type byte and the first byte of timestamp, and the last three bytes of timestamp and the validity period would instead be interpreted as a signature type byte and a timestmap.

It should be emphasized that the messages are barely different and special circumstances must arise for this to be possible, so it is extremely unlilely that this would be exploitable, but it is a potential weakness. It has been plugged by allowing only the currently implemented 5-byte option. Validity periods will be added later with a different format.

The signature classification field describes what kind of signature certificate this is. There are various hex values:

00 -	Signature of a message or document, binary image.
01 -	Signature of a message or document, canonical text.
10 -	Key certification, generic. Only version of key certification supported by PGP 2.5. Material signed is public key pkt and User ID pkt.
11 -	Key certification, persona. No attempt made at all to identify the user with a real name. Material signed is public key pkt and User ID pkt.
12 -	Key certification, casual identification. Some casual attempt made to identify user with his name. Material signed is public key pkt and User ID pkt.
13 -	Key certification, positive ID. Heavy-duty identification efforts, photo ID, direct contact

```
            with personal friend, etc.
            Material signed is public key pkt and User ID pkt.
    20 -    Key compromise.  User signs his own compromise
            certificate.  Independent of user ID associations.
            Material signed is public key pkt ONLY.
    30 -    Key/userid revocation.  User can sign his own
            revocation to dissolve an association between a key
            and a user ID, or certifier may revoke his previous
            certification of this key/userid pair.
            Material signed is public key pkt and User ID pkt.
    40 -    Timestamping a signature certificate made by someone
            else.  Can be used to apply trusted timestamp, and
            log it in notary's log.  Signature of a signature.
            (Planned, not implemented.)
```

When a signature is made to certify a key/UserID pair, it is computed across two packets—the public key packet, and the separate User ID packet. See below.

The packet headers (CTB and length fields) for the public key packet and the user ID packet are both omitted from the signature calculation for a key certification.

A key compromise certificate may be issued by someone to revoke his own key when his secret key is known to be compromised. If that happens, a user would sign his own key compromise certificate with the very key that is being revoked. A key revoked by its own signature means that this key should never be used or trusted again, in any form, associated with any user ID. A key compromise certificate issued by the keyholder shall take precedence over any other key certifications made by anyone else for that key. A key compromise signed by someone other than the key holder is invalid.

Note that a key compromise certificate just includes the key packet in its signature calculation, because it kills the whole key without regard to any userid associations. It isn't tied to any particular userid association. It should be inserted after the key packet, before the first userid packet.

When a key compromise certificate is submitted to PGP, PGP will place it on the public keyring. A key compromise certificate is always accompanied in its travels by the public key and userIDs it affects. If the affected key is NOT already on the keyring, the compromise certificate (and its key and user ID) is merely added to the keyring anywhere. If the affected key IS already on the keyring, the compromise certificate is inserted after the affected key packet. This assumes that the actual key packet is identical to the one already on the key ring, so no duplicate key packet is needed. If a key has been revoked, PGP will not allow its use to encipher any messages, and if an incoming signature uses it, PGP will display a stern warning that this key has been revoked.

NOTE: Key/userid revocation certificates ARE NOT SUPPORTED in this version of PGP. But if we ever get around to supporting them, here are some ideas on how they should work...

A key/userid revocation certificate may be issued by someone to dissolve the association between his own key and a user ID. He would sign it with the very key that is being revoked. A key/userid revocation certificate issued by the keyholder shall take precedence over any other key certifications made by anyone else for that key/userid pair. Also, a third party certifier may revoke his own previous certification of this key/userid pair by issuing a key/userid revocation certificate. Such a revocation should not affect the certifications by other third parties for this same key/userid pair.

When a key/userid revocation certificate is submitted to PGP, PGP will place it on the public keyring. A key/userid revocation certificate is always accompanied in its travels by the public key it affects (the key packet and user ID packet precedes the revocation certificate). If the affected key is NOT already on the keyring, the revocation certificate (and its key and user ID) is merely added to the keyring anywhere. If the affected key IS already on the keyring, the revocation certificate is integrated in with the key's other certificates as though it were just another key certification. This assumes that the actual key packet is identical to the one already on the key ring, so no duplicate key packet is needed.

1.9 Message digest "packet"

The Message digest has no CTB packet framing. It is stored packetless and naked, with padding, encrypted inside the MPI in the Signature packet.

PGP versions 2.3 and later use a new format for encoding the message digest into the MPI in the signature packet, a format which is compatible with RFC1425 (formerly RFC1115). This format is accepted but not written by version 2.2. The older format used by versions 2.2 is acepted by versions up to 2.4, but the RSAREF code in 2.5 is not capable of parsing it.

PGP versions 2.2 and earlier encode the MD into the MPI as follows:

```
MSB                .   .   .                LSB

 0   1   MD(16 bytes)   0   FF(n bytes)   1
```

Enough bytes of FF padding are added to make the length of this whole string equal to the number of bytes in the modulus.

PGP versions 2.3 and later encode the MD into the MPI as follows:

```
MSB                  .   .   .                  LSB

 0   1   FF(n bytes)   0   ASN(18 bytes)   MD(16 bytes)
```

See RFC1423 for an explanation of the meaning of the ASN string. It is the following 18 byte long hex value:

```
3020300c06082a864886f70d020505000410
```

Enough bytes of FF padding are added to make the length of this whole string equal to the number of bytes in the modulus.

All this mainly affects the `rsa_private_encrypt()` and `rsa_public_decrypt()` functions in *rsaglue.c*.

There is no checksum included. The padding serves to verify that the correct RSA key was used.

1.10 Conventional data encryption key (DEK) "packet"

The DEK has no CTB packet framing. The DEK is stored packetless and naked, with padding, encrypted inside the MPI in the RSA public-key-encrypted packet.

PGP versions 2.3 and later use a new format for encoding the message digest into the MPI in the signature packet. (This format is not presently based on any RFCs due to the use of the IDEA encryption system.) This format is accepted but not written by version 2.2. The older format used by versions 2.2 and earlier is also accepted by versions up to 2.4, but the RSAREF code in 2.5 is unable to cope with it.

PGP versions 2.2 and earlier encode the MD into the MPI as follows:

```
MSB                        .   .   .                            LSB

 0   1   DEK(16 bytes)   CSUM(2 bytes)   0   RND(n bytes)   2
```

CSUM refers to a 16-bit checksum appended to the high end of the DEK. RND is a string of NONZERO pseudorandom bytes, enough to make the length of this whole string equal to the number of bytes in the modulus.

PGP versions 2.3 and later encode the MD into the MPI as follows:

```
MSB                        .   .   .                     LSB

 0   2   RND(n bytes)   0   1   DEK(16 bytes)   CSUM(2 bytes)
```

CSUM refers to a 16-bit checksum appended to the high end of the DEK. RND is a string of NONZERO pseudorandom bytes, enough to make the length of this whole string equal to the number of bytes in the modulus.

For both versions, the 16-bit checksum is computed on the rest of the bytes in the DEK key material, and does not include any other material in the calculation. In the above MSB-first representation, the checksum is also stored MSB-first. The checksum is there to help us determine if we used the right RSA secret key for decryption.

All this mainly affects the `rsa_public_encrypt()` and `rsa_private_decrypt()` functions in *rsaglue.c*.

1.11 Conventional key encrypted data packet

```
Offset  Length  Meaning
0       1       CTB for Conventional-Key-Encrypted data packet
1       4       32-bit (or maybe 16-bit) length of packet
5       ?       conventionally-encrypted data.
                plaintext has 64 bits of random data prepended,
                plus 16 bits prepended for "key check" purposes
```

The decrypted ciphertext may contain a compressed data packet or a literal plaintext packet.

After decrypting the conventionally-encrypted data, a special 8-byte random prefix and 2 *key check* bytes are revealed. The random prefix and key check prefix are inserted before encryption and discarded after decryption. This prefix group is visible after decrypting the ciphertext in the packet.

The random prefix serves to start off the cipher feedback chaining process with 64 bits of random material. It may be discarded after decryption. The first 8 bytes is the random prefix material, followed by the 2-byte *key-check* prefix.

The key-check prefix is composed of two identical copies of the last 2 random bytes in the random prefix, in the same order. During decryption, the 9th and 10th bytes of decrypted plaintext are checked to see if they match the 7th and 8th bytes, respectively. If these key-check bytes meet this criterion, then the conventional key is assumed to be correct.

One unusual point about the way encryption is done. Using the IDEA cipher in CFB mode, the first 10 bytes are decrypted normally, but bytes 10 to 17, the first 8 bytes of the data proper, are encrypted using bytes 2 to 9 (the last 8 bytes of the key check prefix) as the IV. This is essentially using CFB-16 for one part of the encryption, while CFB-64 is used elsewhere.

1.12 Compressed data packet

```
Offset  Length  Meaning
0       1       CTB for Compressed data packet
1       1       Compression algorithm selector byte (1=ZIP)
2       ?       compressed data
```

The compressed data begins right after the algorithm selector byte. The compressed data may decompress into a raw literal plaintext data packet with its own CTB. Currently, compressed data packets are always the last ones in their enclosing object, and the decompressor knows when to stop, so the length field is omitted. The low two bits of the CTB are set to 11. This is the only case in PGP where this is currently done.

1.13 Literal data packet, with filename and mode

```
Offset  Length  Meaning
0       1       CTB for raw literal data packet
1       4       32-bit (or maybe 16-bit) length of packet
```

5	1	mode byte, 'b'= binary or 't'= canonical text
6	?	filename, with leading string length byte
?	4	Timestamp of last-modified date, or 0, or right now
?	?	raw literal plaintext data

The timestamp may be have to be derived in a system dependent manner. ANSI C functions should be used to get it if available, otherwise store the current time in it. Or maybe store 0 if it's somehow not applicable.

Whne calculating a signature on a literal packet, the signature calculation only includes the raw literal plaintext data that begins AFTER the header fields in the literal packet—after the CTB, the length, the mode byte, the filename, and the timestamp. The reason for this is to guarantee that detached signatures are exactly the same as attached signatures prefixed to the message. Detached signatures are calculated on a separate file that has no packet encapsulation.

1.14 Comment packet

A comment packet is generally just skipped over by PGP, although it may be displayed to the user when processed. It can be put in a keyring, or anywhere else.

Offset	Length	Meaning
0	1	CTB for Comment packet
1	1	8-bit length of packet
2	?	ASCII comment, size is as in preceding length byte

Comment packets are currently not generated by PGP.

1.15 Secret key certificate

Offset	Length	Meaning
0	1	CTB for secret key certificate
1	2	16-bit (or maybe 8-bit) length of packet
3	1	Version byte (=2). May affect rest of fields that follow. Version byte (=3) for >= PGP2.6 after 9/1/94
4	4	Timestamp
8	2	Validity period, in number of DAYS (0 means forever)
10	1	Algorithm byte for RSA (=1 for RSA). --Algorithm byte affects field definitions that follow.
?	?	MPI of RSA public modulus n
?	?	MPI of RSA public encryption exponent e
?	1	Algorithm byte for cipher that protects following secret components (0=unencrypted, 1=IDEA cipher)
?	8	Cipher Feedback IV for cipher that protects secret components (not present if unencrypted)
?	?	MPI of RSA secret decryption exponent d
?	?	MPI of RSA secret factor p
?	?	MPI of RSA secret factor q
?	?	MPI of RSA secret multiplicative inverse u (All MPI's have bitcount prefixes)
?	2	16-bit checksum of all preceding secret component bytes

All secret fields in the secret key certificate may be password- encrypted, including the checksum. The checksum is calculated from all of the bytes of the unenciphered secret components. The public fields are not encrypted. The encrypted fields are done in CFB mode, and the checksum is used to tell if the password was good. The CFB IV field is just encrypted random data, assuming the dfntrue IV was zero.

NOTE: The secret key packet does not contain a User ID field. The User ID is enclosed in a separate packet that always follows the secret key packet on a keyring or in any other context.

1.16 Public key certificate

```
Offset  Length  Meaning
0       1       CTB for public key certificate
1       2       16-bit (or maybe 8-bit) length of packet
3       1       Version byte (=2).  May affect rest of fields that follow.
                Version byte (=3) for >= PGP2.6 after 9/1/94
4       4       Timestamp of key creation
8       2       Validity period, in number of DAYS (0 means forever)
10      1       Algorithm byte for RSA (=1 for RSA).
                --Algorithm byte affects field definitions that follow.
?       ?       MPI of RSA public modulus n
?       ?       MPI of RSA public encryption exponent e
                (All MPI's have bitcount prefixes)
```

NOTE: The public key packet does not contain a User ID field. The User ID is enclosed in a separate packet that always follows somewhere after the public key packet on a keyring or in any other context.

The validity period is currently always set to 0.

1.17 User ID packet

```
Offset  Length  Meaning
0       1       CTB for User ID packet
1       1       8-bit length of packet
2       ?       User ID string, size is as in preceding length byte
```

The User ID packet follows a public key on a public key ring. It also follows a secret key on a secret key ring.

When a key is certified by a signature, the signature covers both the public key packet and the User ID packet. The signature certificate thereby logically *binds* together the user ID with the key. The user ID packet is always associated with the most recently occurring public key on the key ring, regardless of whether there are other packet types appearing between the public key packet and the associated user ID packet.

There may be more than one User ID packet after a public key packet. They all would be associated with the preceding public key packet.

1.18 Keyring trust packet

The three different forms of this packet each come after: a public key packet, a user ID packet, or a signature packet on the public key ring. They exist only on a public key ring, and are never extracted with a key. Don't copy this separate trust byte packet from keyring, and do add it in back in when adding to keyring.

The meaning of the keyring trust packet is context sensitive. The trust byte has three different definitions depending on whether it follows a key packet on the ring, or follows a user ID packet on the ring, or follows a signature on the ring.

```
Offset  Length  Meaning
0       1       CTB for Keyring trust packet
1       1       8-bit length of packet (always 1 for now)
2       1       Trust flag byte, with context-sensitive bit
                definitions given below.
```

For trust bytes that apply to the preceding key packet, the following bit definitions apply:

```
Bits 0-2 - OWNERTRUST bits - Trust bits for this key owner.  Values are:
     000 - undefined, or uninitialized trust.
     001 - unknown, we don't know the owner of this key.
     010 - We usually do not trust this key owner to sign other keys.
     011 - reserved
     100 - reserved
     101 - We usually do trust this key owner to sign other keys.
     110 - We always trust this key owner to sign other keys.
     111 - This key is also present in the secret keyring.
Bits 3-4 - Reserved.
Bit 5 - DISABLED bit - Means that this key is disabled, and
        should not be used.
Bit 6 - Reserved
Bit 7 - BUCKSTOP bit - Means this key also appears in secret key ring.
        Signifies the ultimately-trusted "keyring owner".
        "The buck stops here".  This bit computed from looking
        at secret key ring.  If this bit is set, then all the
        KEYLEGIT fields are set to maximum for all the user IDs for
        this key, and OWNERTRUST is also set to ultimate trust.
```

For trust bytes that apply to the preceding user ID packet, the following bit definitions apply:

```
Bit 0-1 - KEYLEGIT bits - Validity bits for this key.
        Set if we believe the preceding key is legitimately owned by
        who it appears to belong to, specified by the preceding user
        ID.  Computed from various signature trust packets that
        follow.  Also, always fully set if BUCKSTOP is set.
        To define the KEYLEGIT byte does not require that
        OWNERTRUST be nonzero, but OWNERTRUST nonzero does require
        that KEYLEGIT be fully set to maximum trust.
     00 - unknown, undefined, or uninitialized trust.
     01 - We do not trust this key's ownership.
     10 - We have marginal confidence of this key's ownership.
          Totally useless for certifying other keys, but may be useful
          for checking message signatures with an advisory warning
          to the user.
     11 - We completely trust this key's ownership.
          This requires either:
              - 1 ultimately trusted signature (a signature from
                yourself, SIGTRUST=111)
              - COMPLETES_NEEDED completely trusted signatures
                (SIGTRUST=110)
              - MARGINALS_NEEDED marginally trusted signatures
                (SIGTRUST=101)
              COMPLETES_NEEDED and MARGINALS_NEEDED are configurable
              constants.
Bit 7 - WARNONLY bit - If the user wants to use a not fully validated
        key for encryption, he is asked if he really wants to use this
        key.  If the user answers 'yes', the WARNONLY bit gets set,
        and the next time he uses this key, only a warning will be
        printed. This bit gets cleared during the maintenance pass.
```

For a trust byte that applies to the preceding signature, the following bit definitions apply:

```
Bits 0-2 - SIGTRUST bits - Trust bits for this signature.  Value is
           copied directly from OWNERTRUST bits of signer:
     000 - undefined, or uninitialized trust.
```

```
      001 - unknown
      010 - We do not trust this signature.
      011 - reserved
      100 - reserved
      101 - We reasonably trust this signature.
      110 - We completely trust this signature.
      111 - ultimately trusted signature (from the owner of the ring)
  Bits 3-6 - Reserved.
  Bit 6 - CHECKED bit - This means that the key checking pass (pgp -kc,
          also invoked automatically whenever keys are added to the
          keyring) has tested this signature and found it good.  If
          this bit is not set, the maintenance pass considers this
          signature untrustworthy.
  Bit 7 - CONTIG bit - Means this signature leads up a contiguous trusted
          certification path all the way back to the ultimately-
          trusted keyring owner, where the buck stops.  This bit is derived
          from other trust packets.  It is currently not used for anything
          in PGP.
```

The OWNERTRUST bits are set by the user. PGP does not modify them. PGP computes the BUCKSTOP bit by checking to see if the key is on the secret key ring. If it is, it was created by this user, and thus controlled by him.

All other trust is derived from the BUCKSTOP keys in a special maintenance pass over the keyring. Any good signature made by a given key has its SIGTRUST equal to the key's OWNERTRUST. Based on COMPLETES_NEEDED and MARGINALS_NEEDED, if enough trusted signatures are on a key/userID pair, the key/userid association is considered legitimate.

To be precise, an ultimately trusted key has weight 1, a completely trusted key has weight 1/COMPLETES_NEEDED (or 0 if COMPLETES_NEEDED is 0), and a marginally trsuted key has weight 1/MARGINALS_NEEDED. Other trust values have weight 0. If the total weight of the signatures on a key/userid pair is 1 or more, the userid is considered legitimate.

When a key has a legitimate userid, the user is asked to set the OWNERTRUST for the corresponding key. Ths idea is that the userid identifies someone the user knows, at least by reputation, so once it has been established who holds the secret key, that person's trustworthiness as an introducer can be established and assigned to the key.

Once that is done, the key's signatures then have weight establishing other key/userid associations.

There is a limit to the depth to which this can go. Keys on the secret keyring are at depth 0. Keys signed by those keys are at depth 1. Keys which are fully certified using only signatures from keys at depth 1 or less are at depth 2. Keys which are fully certified using only signatures from keys at depth 2 or less are at depth 3, and so on.

If you know all of your trusted introducers personally, and have signed their keys, then you will never have a key at a depth of greater than 2. The maximum depth is limited my MAX_CERT_DPETH. It never gets very large in a well-connected "web of trust".

This redundant and decentralized method of determining public key legitimacy is one of the principal strengths of PGP's key management architecture, as compared with PEM, when used in social structures that are not miltiary-style rigid hierarchies.

The trust of a key owner (OWNERTRUST) does not just reflect our estimation of their personal integrity, it also reflects how competent we think they are at understanding key management and using good judgement in signing keys. The OWNERTRUST bits are not computed from anything — it requires asking the user for his opinion.

To define the OWNERTRUST bits for a key owner, ask:

```
Would you always trust "Oliver North"
to certify other public keys?
(1=Yes, 2=No, 3=Usually, 4=I don't know) ? _
```

When a key is added to the key ring the trust bytes are initialized to zero (undefined).

[–manual setting of SIGTRUST/OWNERTRUST not implemented] Normally, we derive the value of the SIGTRUST field by copying it directly from the signer key's OWNERTRUST field. Under special circumstances, if the user explicitly requests it with a special PGP command, we may let the user override the copied value for SIGTRUST by displaying an advisory to him and asking him for ratification, like so:

```
This key is signed by "Oliver North",
whom you usually trust to sign keys.
Do you trust "Oliver North"
to certify the key for "Daniel Ellsberg"?
(1=Yes, 2=No, 3=Somewhat, 4=I don't know) ? _      <default is yes>
```

Or:

```
This key is signed by "Oliver North",
whom you usually do not trust to sign keys.
Do you trust "Oliver North"
to certify the key for "Daniel Ellsberg"?
(1=Yes, 2=No, 3=Somewhat, 4=I don't know) ? _      <default is no>
```

An *I don't know* response to this question would have the same effect as a response of *no*.

If we had no information about the trustworthiness of the signer (the OWNERTRUST field was uninitialized), we would leave the advisory note off.

Certifying a public key is a serious matter, essentially promising to the world that you vouch for this key's ownership. But sometimes I just want to make a *working assumption* of trust for someone's public key, for my own purposes on my own keyring, without taking the serious step of actually certifying it for the rest of the world. In that case, we can use a special PGP keyring management command to manually set the KEYLEGIT field, without relying on it being computed during a maintenance pass. Later, if a maintenance pass discovers a KEYLEGIT bit set that would not have been otherwise computed as set by the maintenance pass logic, it alerts me and asks me to confirm that I really want it set.

[–end of not implemented section]

During routine use of the public keyring, we don't actually check the associated signatures certifying a public key. Rather, we always rely on trust bytes to tell us whether to trust the key in question. We depend on a separate checking pass (pgp -kc) to actually check the key signature certificates against the associated keys, and to set the trust bytes accordingly. This pass checks signatures, and if a signature fails to verify, obnoxiously alerts the user and drops it from the key ring. Then it tuns a maintenance pass to calculate the ring-wide effects of this.

A failed signature should be exceedingly rare, and it may not even result in a KEYLEGIT field being downgraded. Having several signatures certifying each key should prevent damage from spreading too far from a failed certificate. But if dominoes do keep falling from this, it may indicate the discovery of an important elaborate attack.

The maintenance pass is run every time the keyring changes, and operates in a top-of-pyramid-down manner as follows.

If at any time during any of these steps the KEYLEGIT field goes from not fully set to fully set, and the OWNERTRUST bits are still undefined, the user is asked a question to define the OWNERTRUST bits. First, for all keys with BUCKSTOP set, check if they are really present in the secret keyring, if not, the BUCKSTOP bit is cleared. SIGTRUST and KEYLEGIT is initialized to zero for non-buckstop keys.

The real maintenance pass is done in a recursive scan: Start with BUCKSTOP keys, find all userid/key pairs signed by a key and update the trust value of these signatures by copying the OWNERTRUST of the signer to the SIGTRUST of the signature. If this makes a key fully validated, start looking for signatures made by this key, and update the trust value for them. Repeat until everything has settled down.

2 Public Key Ring Overall Structure

A public key ring is comprised of a series of public key packets, keyring trust packets, user ID packets, and signature certificates.

Here is an example of an ordered collection of packets on a ring:

```
Public key packet
    Keyring trust packet for preceding key
  User ID packet for preceding key
      Keyring trust packet for preceding user ID/key association
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate

Public key packet
    Keyring trust packet for preceding key
  User ID packet for preceding key
      Keyring trust packet for preceding user ID/key association
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate
  User ID packet for preceding key
      Keyring trust packet for preceding user ID/key association
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate

Public key packet
    Keyring trust packet for preceding key
  Compromise certificate for preceding key
  User ID packet for preceding key
      Keyring trust packet for preceding user ID/key association
    Signature certificate to bind preceding User ID and key pkt
        Keyring trust packet for preceding signature certificate
```