PART PART PART PART PART PART PART PART

III

Messaging: Creating a Secure Channel

10 *Encryption Overview* *615*

11 *PGP* *633*

CHAPTER 10

Encryption Overview

When you mention encryption, images of spies and secret agents come to mind. You may visualize spies in trench coats meeting beneath street lamps on foggy evenings to talk in whispered tongues. This is Hollywood's version of encryption. Reality splits in two directions. First, encryption is a great deal more boring than ever depicted in any film. Creating and breaking codes is tedious work wherein many hours are spent getting nowhere. Second, encryption is a weapon. The U.S. government has classified encryption as munitions. Stringent laws prevent it from being exported overseas and govern its uses internally.

To understand why this is the case, it is important to classify what encryption is and to view it in historical perspective.

What Is Encryption?

Simply put, *encryption* is the art of hiding a message within another, or making a message unreadable to all but the intended party. This can be accomplished in innumerable ways. One of the most rudimentary techniques is hiding a message within another. The following are two poems from Brad Truitt (reprinted with permission of the author):

Untitled Fourteen

Plastic butterfly frozen

In flight with nowhere to go

Chewing on plastic nectar

Telling the real flowers no

Under the pain and primer

Rests some type of molded paste

Every now and then someone

Asks if you're more than a waste

Let their questions go ignored

But offer a paradox

Under your plastic, are your

Memories governed by clocks?

The Ballad of Johnny B. Gutt

Hypocrisy is my lifeblood

And anger is how things are viewed.

Racism is my long fingers and

Decadence provides me with food.

Do you dare doubt my principles,

Realizing the cheer they give?

Indeed, you had better think twice

Noting that I know where you live.

Kindness is a most useless tool

In that it never built a thing.

Narrow-mindedness, though, is a

Gift that will never fail to sing.

Although they appear to be mediocre poems in nature, notice that selecting the first character of each line translates into a message: Picture Album and Hard Drinking, respectively.

Lewis Carroll is known to have used this method of writing to hide the names of girls within his works. This method of encryption is extremely vulnerable. Its only strength lies in the fact that you hope no one will take the time to examine it carefully, for it is plainly written in sight for anyone to see. Once someone figures out the secret, however, it can be applied to every message intercepted. Security here is meaningless.

Other measures must be used if the security of the data is truly important. Two of the most common methods of encryption are substitution and transposition.

Transposition

In *transposition*, the same characters that make up the message are still used, but their order is jumbled in a way that makes it difficult to read the message. Suppose, for example, that you want to encrypt the message: Trucks and vehicles with trailers use right lane. The message can be written in two columns in the following manner:

T	r
u	c
k	s
	a
n	d
	v
e	h
i	c
l	e

s	
w	i
t	h
	t
r	a
i	l
e	r
s	
u	s
e	
r	i
g	h
t	
l	a
n	e

If you print the first column of the message, followed by the second column, it becomes: Tuk n vhce wt talr uergtlnrcsadvhce ihtalr s ih ae. This is much more difficult to read and takes some time to break the code.

The code becomes more difficult if you add additional columns and stagger their order. By using five columns and staggering the order in which they are presented, the code becomes:

5	*3*	*1*	*4*	*2*
T	r	u	c	k
s		a	n	d
	v	e	h	i
c	l	e	s	
w	i	t	h	
t	r	a	i	l

e	r	s		u
s	e		r	i
g	h	t		l
a	n	e		

This translates into: uaeetas tekdi luilr vlirrehncnhshi r Ts cwtesga. Because the exact same message you are trying to send is still here, the only key to deciphering it is figuring out the number of columns used to create it and the order in which they are being presented.

Deciphering

For practice, assume that transposition was used, and the message you find is:

E NYIN BHA RWD OE T AU NFNTEERW EINFWA ALMTPDOSHR TAAR

You now must ascertain two things: the number of columns that were used and the order in which the columns were placed. The length of the string is 55 characters, and if you assume that five columns were used, each column will consist of eleven characters:

1	*2*	*3*	*4*	*5*
E	R		E	P
	W		I	D
N	D	N	N	O
Y		F	F	S
I	O	N	W	H
N	E	T	A	R
		E		
B	T	E	A	T
H		R	L	A
A	A	W	M	A
	U		T	R

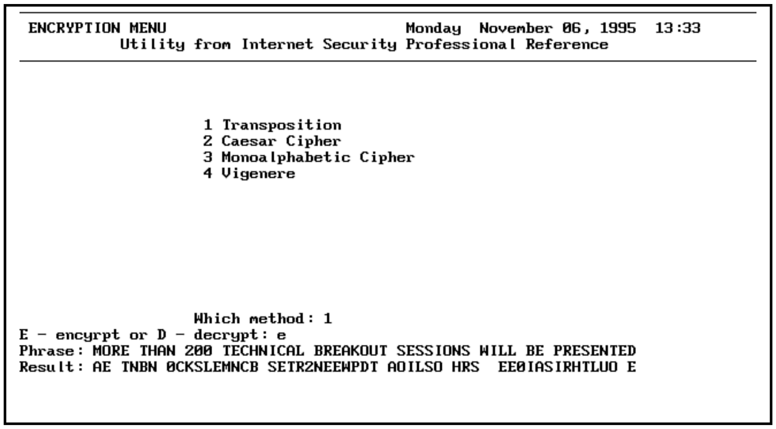
Regardless of the order in which you arrange the columns, it fails to be intelligible—an indication that five is not the correct number of columns. Increasing the number to eight renders the following:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
E		D	A	T	I	L	H
	B		U	E	N	M	R
N	H	O		E	F	T	
Y	A	E		R	W	P	T
I			N	W	A	D	A
N	R	T	F			O	A
	W		N	E	A	S	R

By mixing the order of the columns about and concentrating on making a legible word in the first row, it is possible to produce the following:

3	*6*	*4*	*7*	*2*	*5*	*8*	*1*
D	I	A	L		T	H	E
	N	U	M	B	E	R	
O	F		T	H	E		N
E	W		P	A	R	T	Y
	A	N	D		W	A	I
T		F	O	R		A	N
	A	N	S	W	E	R	

In other words, dial the number of the new party and wait for an answer. The CD included with this book contains a DOS-executable file called ENCRYPT, in which you can choose Transposition from the menu and practice encrypting and decrypting messages. ENCRYPT is hard coded to use the same transposition method utilized in the previous example, where the column ordering becomes: 8-5-1-3-6-2-4-7. This means that the first column of the encrypted message was the eighth column of the original message; the second column of the encrypted message was the fifth column of the original message; and so on. The following screen shots illustrate this in action.

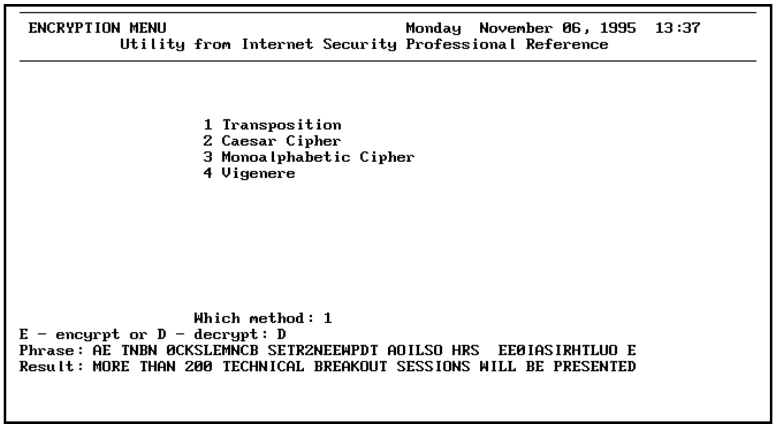


```
ENCRYPTION MENU                          Monday  November 06, 1995  13:33
          Utility from Internet Security Professional Reference

                    1 Transposition
                    2 Caesar Cipher
                    3 Monoalphabetic Cipher
                    4 Vigenere

                   Which method: 1
E - encyrpt or D - decrypt: e
Phrase: MORE THAN 200 TECHNICAL BREAKOUT SESSIONS WILL BE PRESENTED
Result: AE TNBN 0CKSLEMNCB SETR2NEEWPDT AOILSO HRS  EE0IASIRHTLUO E
```

Figure 10.1
Encrypting a message with transcription.



```
ENCRYPTION MENU                          Monday  November 06, 1995  13:37
          Utility from Internet Security Professional Reference

                    1 Transposition
                    2 Caesar Cipher
                    3 Monoalphabetic Cipher
                    4 Vigenere

                   Which method: 1
E - encyrpt or D - decrypt: D
Phrase: AE TNBN 0CKSLEMNCB SETR2NEEWPDT AOILSO HRS  EE0IASIRHTLUO E
Result: MORE THAN 200 TECHNICAL BREAKOUT SESSIONS WILL BE PRESENTED
```

Figure 10.2
Decrypting a message with transcription.

Substitution

Substitution differs from transposition in one key way: with transposition, all characters from the original message are still there; with substitution, none of them are. *Substitution* replaces each character of the original message with another. Breaking the code is based upon ascertaining which character the one you are seeing is replacing.

Caesar Cipher

Julius Caesar was one of the first to use substitution encryption to send messages to troops during war. The substitution method he is credited with involves advancing each character three spaces in the alphabet. Thus:

DIAL THE NUMBER OF THE NEW PARTY AND WAIT FOR AN ANSWER

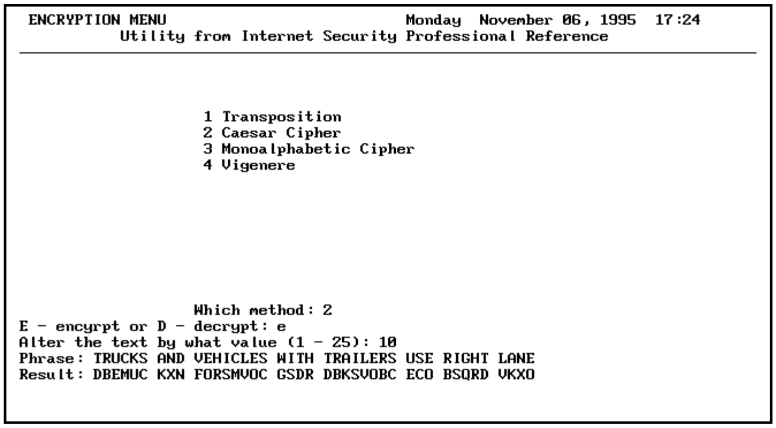
becomes

GLDO WKH QXPEHU RI WKH QHZ SDUWB DQG ZDLW IRU DQ DQVZHU

Note that at the end of the alphabet, characters wrap back around to the beginning; thus, Z becomes C, Y becomes B, and so on.

This encrypting method serves its purpose well, but once one figures out that they need only subtract three characters from each given to obtain the correct answer, the code loses all security. A modification of this is allowing the number of characters shifted to differ for each message. In so doing, the number of possibilities changes from 1 to 26 and becomes increasingly more difficult.

Figure 10.3 shows an example of the ENCRYPT utility included on the CD with this book, and uses a modified version of the Caesar Cipher to change a message 10 spaces. Figure 10.4 shows its counterpart, decrypting the same message.



```
 ENCRYPTION MENU                        Monday  November 06, 1995  17:24
            Utility from Internet Security Professional Reference

                        1 Transposition
                        2 Caesar Cipher
                        3 Monoalphabetic Cipher
                        4 Vigenere

                       Which method: 2
E - encyrpt or D - decrypt: e
Alter the text by what value (1 - 25): 10
Phrase: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
Result: DBEMUC KXN FORSMVOC GSDR DBKSVOBC ECO BSQRD VKXO
```

Figure 10.3
Encrypting using a modified version of the Caesar Cipher.

Although this method is stronger than always shifting three characters, it is still relatively easy to break. The only unknown that must be determined is the number of places in the alphabet to switch each character. Because there are only 25 possibilities, a routine can be written to quickly run through all feasible combinations and print the results, as follows:

DBEMUC KXN FORSMVOC GSDR DBKSVOBC ECO BSQRD VKXO

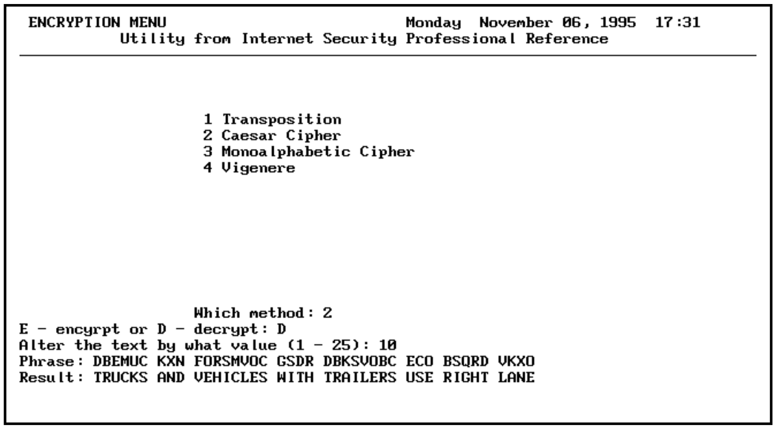
ECFNVD LYO GPSTNWPD HTES ECLTWPCD FDP CTRSE WLYP

FDGOWE MZP HQTUOXQE IUFT FDMUXQDE GEQ DUSTF XMZQ

GEHPXF NAQ IRUVPYRF JVGU GENVYREF HFR EVTUG YNAR

```
HFIQYG OBR JSVWQZSG KWHV HFOWZSFG IGS FWUVH ZOBS
IGJRZH PCS KTWXRATH LXIW IGPXATGH JHT GXVWI APCT
JHKSAI QDT LUXYSBUI MYJX JHQYBUHI KIU HYWXJ BQDU
KILTBJ REU MVYZTCVJ NZKY KIRZCVIJ LJV IZXYK CREV
LJMUCK SFV NWZAUDWK OALZ LJSADWJK MKW JAYZL DSFW
MKNVDL TGW OXABVEXL PBMA MKTBEXKL NLX KBZAM ETGX
NLOWEM UHX PYBCWFYM QCNB NLUCFYLM OMY LCABN FUHY
OMPXFN VIY QZCDXGZN RDOC OMVDGZMN PNZ MDBCO GVIZ
PNQYGO WJZ RADEYHAO SEPD PNWEHANO QOA NECDP HWJA
QORZHP XKA SBEFZIBP TFQE QOXFIBOP RPB OFDEQ IXKB
RPSAIQ YLB TCFGAJCQ UGRF RPYGJCPQ SQC PGEFR JYLC
SQTBJR ZMC UDGHBKDR VHSG SQZHKDQR TRD QHFGS KZMD
TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
USVDLT BOE WFIJDMFT XJUI USBJMFST VTF SJHIU MBOF
VTWEMU CPF XGJKENGU YKVJ VTCKNGTU WUG TKIJV NCPG
WUXFNV DQG YHKLFOHV ZLWK WUDLOHUV XVH ULJKW ODQH
XVYGOW ERH ZILMGPIW AMXL XVEMPIVW YWI VMKLX PERI
YWZHPX FSI AJMNHQJX BNYM YWFNQJWX ZXJ WNLMY QFSJ
ZXAIQY GTJ BKNOIRKY COZN ZXGORKXY AYK XOMNZ RGTK
AYBJRZ HUK CLOPJSLZ DPAO AYHPSLYZ BZL YPNOA SHUL
BZCKSA IVL DMPQKTMA EQBP BZIQTMZA CAM ZQOPB TIVM
CADLTB JWM ENQRLUNB FRCQ CAJRUNAB DBN ARPQC UJWN
```

Running such a routine quickly reveals the correct translation of the code, as it is the only sentence that is readable of the lot.



```
 ENCRYPTION MENU                          Monday  November 06, 1995  17:31
          Utility from Internet Security Professional Reference


                    1 Transposition
                    2 Caesar Cipher
                    3 Monoalphabetic Cipher
                    4 Vigenere


                   Which method: 2
E - encyrpt or D - decyrpt: D
Alter the text by what value (1 - 25): 10
Phrase: DBEMUC KXN FORSMVOC GSDR DBKSVOBC ECO BSQRD VKXO
Result: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
```

Figure 10.4
Decrypting using a modified version of the Caesar Cipher.

Monoalphabetic Substitutions

Monoalphabetic substitutions, or ciphers, are more difficult to break than their Caesarean counterparts. Here, each character can stand for another—including itself—and there is no reason why one replaces another. Monoalphabetic substitutions are often found in newspaper leisure sections under the name Cryptoquotes, or something similar.

In the following example, the code used is:

A=P	N=Y
B=R	O=S
C=O	P=V
D=D	Q=X
E=U	R=J
F=C	S=Z
G=E	T=B
H=L	U=W
I=A	V=N
J=T	W=Q
K=M	X=H
L=I	Y=K
M=F	Z=G

Thus, the message:

DIAL THE NUMBER OF THE NEW PARTY AND WAIT FOR AN ANSWER

becomes:

DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ

This code is much more difficult to break, as each character now has 26 possibilities. A program can be written that will try every possibility for each character and print the results. You can then read all the entries and look for the one that makes sense, or you can apply some rules to the sentence.

One such method is looking for small words and trying substitutions on them. It is safe to assume that one of the three-letter words in the sentence is "THE." Four possibilities exist, and thus four substitution trials:

```
DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ
     THE     E     THE  E     T         T            E
```

or

```
DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ
       H T   H       H THE        T  E         T  T EH
```

or

```
DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ
E T      H             H    T    THE  T       TE TE
```

or

```
DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ
              E HT           E            THE         E
```

The third scenario is immediately dismissed, as there is no two-letter word TE. The fourth scenario is immediately dismissed as well, for there is no two-letter word HT. That leaves the first and second scenarios as possiblities. Of the two, the first is far and away the most complete and the one upon which a decryption expert would focus.

The next area to focus upon is the two-letter words, now represented as SC and PY. These cannot be BE, TO, or AT, as they do not contain either a "T" or an "E". That leaves few other possibilities, and through some trial and error, their identity can be ascertained, such that the code now appears as the following:

```
DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ
  A  THE N   E  OF THE NE   A T  AN   A T FO  AN AN  E
```

Assuming that PYD must be AND, and CSJ is FOR:

```
DAPI BLU YWFRUJ SC BLU YUQ VPJBK PYD QPAB CSJ PY PYZQUJ
D A  THE N   ER OF THE NE   ART  AND  A T FOR AN AN  ER
```

It is only a matter of time and trial and error until the rest of the puzzle falls into place.

Another Way

In modern English, there is a propensity to use some characters more than others. The "Q", for example, is rarely used. When it is used, it must be followed by a "U". According to *Cryptography: An Introduction to Computer Security* (Seberry and Pieprzyk, Prentice Hall, 1989), the following relative frequency of use can be applied to each letter:

E	12.75	U	3.00
T	9.25	M	2.75
R	8.50	P	2.75
N	7.75	Y	2.25
I	7.75	G	2.00
O	7.50	W	1.50
A	7.25	V	1.50
S	6.00	B	1.35
D	4.25	K	0.50
L	3.75	X	0.50
H	3.50	Q	0.50
C	3.50	J	0.25
F	3.00	Z	0.25

The number of times each character appears in the encrypted message is as follows:

A	2	M	0
B	4	N	0
C	2	O	0
D	2	P	6
E	0	Q	3
F	1	R	1
G	0	S	2

H	0	T	0
I	1	U	5
J	4	V	1
K	1	W	1
L	2	X	0
Z	1	Y	5

Naturally, the larger the piece of encrypted data, the more true it will be to the frequency chart. Nevertheless, applying it to this message, the most frequently used characters in the code are:

B, J, P,U, and Y.

If the frequency theory is correct, these should be replaceable with:

E, T, R, N, and I—not necessarily in that order.

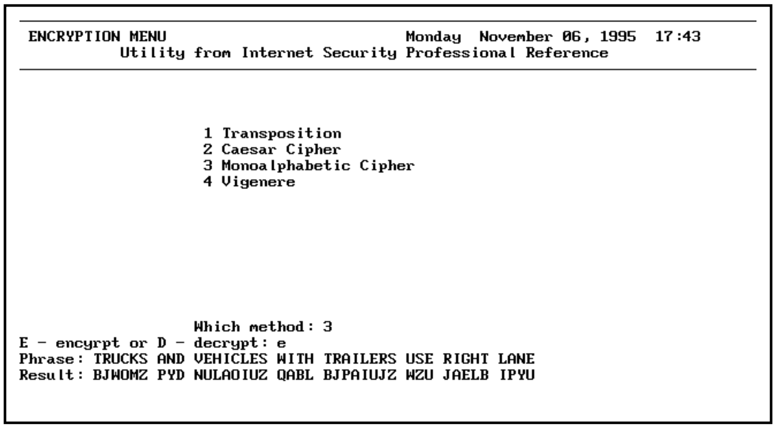
The most common characters in the actual, unencrypted message are:

A, E, N, R, and T.

Thus, four of the five characters that should be there, match up with the characters that are there. This is a respectable beginning. Again, the larger the encrypted text, the more likely the frequency distribution is to be accurate.

Practice

Figure 10.5 shows an example of encrypting a message by using the ENCRYPT utility found on the CD accompanying this book. Figure 10.6 shows unencrypting it with monoalphabetic substitution.



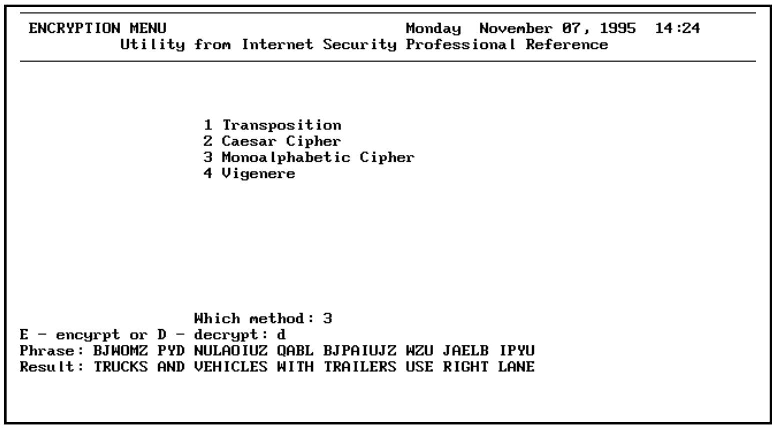
```
 ENCRYPTION MENU                          Monday  November 06, 1995  17:43
          Utility from Internet Security Professional Reference


                    1 Transposition
                    2 Caesar Cipher
                    3 Monoalphabetic Cipher
                    4 Vigenere



                   Which method: 3
E - encyrpt or D - decrypt: e
Phrase: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
Result: BJWOMZ PYD NULAOIUZ QABL BJPAIUJZ WZU JAELB IPYU
```

Figure 10.5
Encrypting a message with monoalphabetic encryption.



```
ENCRYPTION MENU                              Monday  November 07, 1995  14:24
         Utility from Internet Security Professional Reference


                      1 Transposition
                      2 Caesar Cipher
                      3 Monoalphabetic Cipher
                      4 Vigenere






                     Which method: 3
E - encyrpt or D - decrypt: d
Phrase: BJWOMZ PYD NULAOIUZ QABL BJPAIUJZ WZU JAELB IPYU
Result: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
```

Figure 10.6
Decrypting a monoalphabetic encryption message.

Vigenere Encryption

With standard monoalphabetic encryption, the key to breaking the code is figuring out what each character stands for. Once done, the code is solved, for each character maintains its same meaning throughout the duration of the encryption. *Vigenere encryption* adds one more level of difficulty, in that the value of each character is different each time it is used.

The key to understanding the way this is done is knowing that Vigenere adds something to the equation none of the others have thus far: a key. The key is a word or phrase that is used to encrypt and decrypt the message. To understand the way this works, consider the following matrix:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

When encrypting a character using this matrix, compare it with a matching character in the key, and find where the two correspond to ascertain the encryption character. Although it seems complicated, it is really very simple. Note the following example:

Key: OPPORTUNITY

Phrase: PUBLISHING

Result: DJQZZLBVVZ

Looking at the matrix, the O and P match up with a result of D. Likewise, P and U match up with a result of J, and so on, for the duration of the encryption. When the phrase to be encrypted is longer than the key—as is almost always the case—then the key repeats itself over and over. Thus, OPPORTUNITY really is OPPORTUNITYOPPORTUNITYOPPORTUNITY, and so on.

To decrypt the message, you must know the key that was used to create the encryption. Although not impossible, without this vital piece of information, it becomes extremely difficult to break the code.

The following example shows a phrase that has been used throughout this chapter encrypted with a key:

Key: OPPORTUNITY

Phrase: DIAL THE NUMBER OF THE NEW PARTY AND WAIT FOR AN ANSWER

Result: RXPZ1MBR(GSAQTF1HZ-BAC.CTK1IUEBR8OCS.NTCG(YMF/PB1THFEXP

Notice that in the complete version of the matrix, spaces and other punctuation are also included. The following shows three attempts to decrypt the message without knowing the correct key:

Attempt One

Key: CONSIDER

Phrase: RXPZ1MBR(GSAQTF1HZ-BAC.CTK1IUEBR8OCS.NTCG(YMF/PB1THFEXP

Result: PJCHCJXA&SFIIQB FL JSZDLRW$QMBXAPAPA&KPLE_LUXFLKIFUNWUL

Attempt Two

Key: ORANGEBOOK

Phrase: RXPZ1MBR(GSAQTF1HZ-BAC.CTK1IUEBR8OCS.NTCG(YMF/PB1THFEXP

Result:
DGPMEIAD_WEJQGZGGLRMLHPNGJUGUNARBWOGZFSS_YZZEON#JTOEKJ

Attempt Three

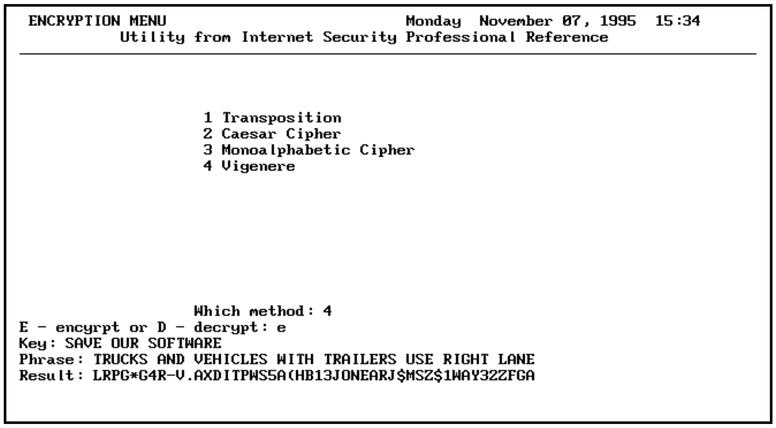
Key: OPPORTUNITIES

Phrase: RXPZ1MBR(GSAQTF1HZ-BAC.CTK1IUEBR8OCS.NTCG(YMF/PBLDNQIAP

Result: DIAL THE NKWYFQ"TI_HNU_UPS#TFQKY$BUZ&JBOR_KVM_CT

Notice the previous example. Although the guess to the key's identity is very close, the result is accurate only to the extent of the accuracy in the first occurrence of the guess. Nowhere else in the phrase are the correct characters decrypted, even though the guess is extremely close.

Figures 10.7 and 10.8 show examples of using the ENCRYPT utility to encrypt and then decrypt a message.

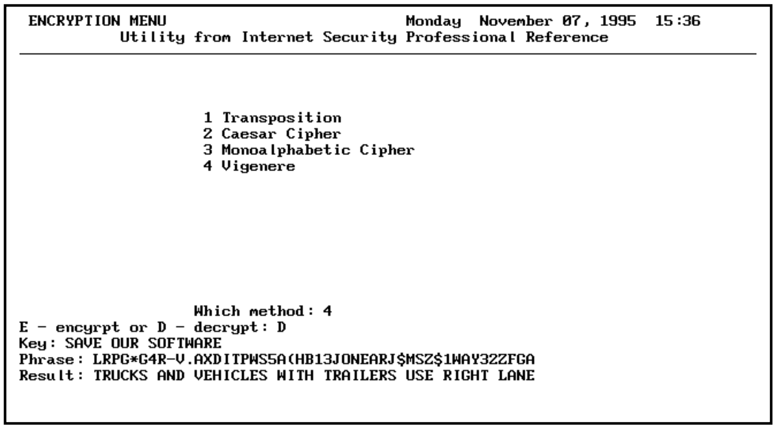


```
 ENCRYPTION MENU                              Monday  November 07, 1995  15:34
           Utility from Internet Security Professional Reference


                   1 Transposition
                   2 Caesar Cipher
                   3 Monoalphabetic Cipher
                   4 Vigenere


                  Which method: 4
E - encyrpt or D - decrypt: e
Key: SAVE OUR SOFTWARE
Phrase: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
Result: LRPG*G4R-V.AXDITPWS5A(HB13JONEARJ$MSZ$1WAY32ZFGA
```

Figure 10.7
Encrypting a message with Vigenere encryption.



```
 ENCRYPTION MENU                              Monday  November 07, 1995  15:36
           Utility from Internet Security Professional Reference


                   1 Transposition
                   2 Caesar Cipher
                   3 Monoalphabetic Cipher
                   4 Vigenere


                  Which method: 4
E - encyrpt or D - decrypt: D
Key: SAVE OUR SOFTWARE
Phrase: LRPG*G4R-V.AXDITPWS5A(HB13JONEARJ$MSZ$1WAY32ZFGA
Result: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
```

Figure 10.8
Decrypting a message with Vigenere encryption.

To make the message even more secure, encrypt the same message a number of times, using a different key each time, as illustrated in the following. Each of the keys here are coming from portions of newspaper headlines, making it easy for others to use the same keys.

```
Key:    SAVE OUR SOFTWARE
Phrase: TRUCKS AND VEHICLES WITH TRAILERS USE RIGHT LANE
Result: LRPG*G4R-V.AXDITPWS5A(HB13JONEARJ$MSZ$1WAY32ZFGA

Key:    MOB ACTIVITY
Phrase: LRPG*G4R-V.AXDITPWS5A(HB13JONEARJ$MSZ$1WAY32ZFGA
Result: XFQ&*IGZBDAYJRJ3PYL=V0AZ=AK.NGTZE,FQL226AAF:UNZY

Key:    SOCIAL PROBLEMS
Phrase: XFQ&*IGZBDAYJRJ3PYL=V0AZ=AK.NGTZE,FQL226AAF:UNZY
Result: PTS.*T&OSRBJNDBEDAT=G_PQKBV2ZYLNG4FB+ACDBLJFMFNA
```

To take the final result back to the original message, someone trying to break the code must now decrypt it three times, knowing three sets of keys. This is a very difficult task, indeed.

CHAPTER 11

PGP

PGP (Pretty Good Privacy) is a software encryption program that enables users to create secure messages and communicate securely over insecure communication links, such as e-mail and netnews. PGP uses various forms of encryption and combines messages with a simple packet format to provide a simple and efficient security mechanism for the transmission of messages over the Internet and other networks.

This chapter explains PGP 2.6.2, gives a little history and background, and talks about the different security methods PGP provides. The chapter explains the use of PGP "keys" and discusses security concerns with PGP and known attacks against PGP.

It is important for a system administrator to understand the security requirements and implications of using PGP. Because it is such a popular program, many users may want to use it. Instead of each user having his or her own copy, it is worthwhile to make it available system-wide. Understanding how the program works and how it needs to be maintained will help an administrator perform the job in an effective, educated manner.

PGP Overview

The PGP program has become the de facto standard for public key cryptography and message security worldwide. The program has evolved since its first release, and is now very popular. Hundreds of thousands of copies of PGP are known to have been distributed from public sites worldwide. Although it is impossible to know exactly how many copies of PGP exist, or how many people use PGP on a regular basis, it is easy to see that PGP is the most widely used security program on the Internet. On many Usenet newsgroups or popular e-mail lists, for example, a high number of posts use PGP. This section describes the history of PGP and explains other background information about the program.

Security is a trade-off between the cost of the data being protected and the cost it takes an attacker to get that data. Protecting data worth only a dime with a security system that costs a million dollars to break is obviously a bad investment. On the other hand, the protection of data worth a million dollars with a security system that costs a dime to crack is a serious problem. The trade-off is to find the balance—the cost to an attacker to compromise the security protecting data, and the worth of the data to its owner. The cost to break a security system can be measured in many ways, which includes the amount of computer time necessary to perform the security break.

PGP is currently believed by many to be the best and most cost-effective security program available. It uses some of the best known encryption technology, and provides security that, it is believed, governments cannot break. Moreover, because the source code is available many people have looked at the program in search of bugs and security flaws; all of them have been corrected as they have been found.

History of PGP

PGP version 1.0 was first released in the summer of 1991 in the United States through an ftp site and a Usenet news posting. This program used a home-brewed secret-key encryption scheme called Bass-O-Matic and implemented the Rivest, Shamir, and Adelmen (RSA) public-key encryption system. Unfortunately the Bass-O-Matic system was less than secure. The idea, however—to provide a simple program that provides the user with strong encryption, and to make it available to everyone—was genius.

The original PGP was created by Philip Zimmermann, a political activist turned programmer and cryptographer. Philip started his work on PGP when the United States Congress started considering restricting freedoms on computers. As a result, Philip decided to write a program that could protect the privacy of electronic communications to thwart the draconian laws that were under consideration. The result was PGP 1.0.

In September 1992, PGP 2.0 was released in Europe. A group of programmers took the ideas from the earlier 1.0 release, added some features, put in a real cryptographic system, and released a new version of PGP. The 2.0 release also replaced the Bass-O-Matic encryption scheme with IDEA, a professionally developed cryptosystem. The IDEA Cipher is a block cipher similar to the Data Encryption Standard (DES), except that it has a larger key and is believed to be more secure.

With the release of PGP 2.0, the program started gaining popularity. Computer users around the world started using PGP to protect their communications from electronic eavesdroppers or would-be counterfeiters. The program's simplicity and ease-of-use made the program popular with many different skill levels of computer users.

One of the problems that held back the use of PGP is that patents exist on the RSA cryptosystem in the United States. Because PGP did not have a license, it was claimed that PGP violated the RSA patents. One solution was found by a company called ViaCrypt, which started selling a commercial version of PGP. ViaCrypt was licensed to sell software that uses the RSA patent, so they could legally sell PGP.

Another problem holding back the use of PGP is the United Stated International Traffic and Arms Regulations, or ITAR. The ITAR rules limit the exportability of military munitions, such as guns and nuclear weapons. Unfortunately, cryptographic systems, encryption systems, and so forth, are also considered munitions under ITAR. Therefore, exporting programs that use cryptography, such as PGP, could be considered arms smuggling.

Finally, in June 1994, the Massachusetts Institute of Technology released a free version of PGP for United States citizens that had an RSA license, thereby freeing PGP from its "forbidden-ware" status and allowing anyone in the United States to use it. Since this time, PGP's acceptance and use has grown dramatically. Many say that it has become the de facto standard for public key cryptography in the world.

Why Use PGP?

People use PGP for a variety of reasons. Most people use PGP because they want to protect their electronic files and communications. These reasons might include:

- If you do not want your messages to fall into the hands of other companies
- If you want to keep your files private from crackers
- If you believe you have the right to private conversations
- If you want a simple method to authenticate messages

PGP is also easier to use than any current alternative. The command-line interface and time it takes to start using the program is practically zero. The commercially available Privacy-Enhanced Mail (PEM), for example, requires a user to generate a key and then wait to get it signed by a Certification Authority before the key can be used in communications. PGP, however, needs to only generate a key and then the user can immediately start using PGP features.

PGP has a large and growing number of users worldwide. If you want to encrypt your communications, it is most useful if your intended correspondents are also using the same encryption programs. Because PGP has become the de facto standard in electronic privacy, you should use the same technology to ensure that files will be compatible.

Short Encryption Review

Although the science of encryption is explained in Chapter 10, a few definitions in this chapter help you understand how PGP works.

Secret Key Encryption

Secret Key Encryption (SKE), also called Conventional Encryption, is defined as a cryptosystem in which the same key is used to encrypt and decrypt a message. In other words, a key turns a message into a seemingly random stream of bits. Later, some other user uses that same key to turn the random stream of bits back into the original message. SKE systems are fast and provide a high degree of security for the number of bits in the key.

Public Key Encryption

Public Key Encryption (PKE) defines a set of encryption schemes (cryptosystems) in which two keys are involved. When a user encrypts a message in one key to create an output ciphertext, decryption of that ciphertext requires the use of the second key to obtain the original message. The two keys are created to form a mathematical relationship; part of this relationship is that knowledge of one key, the secret key, is computationally infeasible to obtain by possession of the other key, the public key.

The term *computationally infeasible* means a process that is not time-invariant. In general, this means that it is difficult to perform the operation in question. However, what is difficult in the year 1996 may not be difficult in the year 2000. When an algorithm is computationally infeasible to break, it means that it is computationally infeasible today, and is expected to be easier to break in the future.

Current Public Key cryptosystems are based on difficult mathematical problems. The RSA cryptosystem, for example, is based on the difficulty of factoring a large number that is the product of two large prime numbers. In such a system as RSA, creating the private key from the public key is only known to be as difficult as factoring the public key modulus into the two prime numbers. An RSA public key is made of the following two parts:

- Modulus
- Exponent

The *modulus* is the product of two large primes and is the basis for a mathematical system called a group. The *exponent* is chosen at key creation time to fit a particular mathematical relationship with the secret key.

The public key can be safely given to anyone who wants it. It can be published, and knowledge of that key does not break the security of the system. PGP keeps two key rings, a public key ring and a secret key ring, to maintain a cache of known public and secret keys. More on this later in the section "PGP Key rings."

The biggest problems with PKE systems are that Public Key systems are slow, cumbersome, and require large keys to maintain decent levels of security. As of this writing, the time it would take to brute-force a 128-bit IDEA key is about as long as it would take to factor a 3,000-bit RSA key. To "brute-force" a key, every possible key is tried to find the correct one. Moreover, a single RSA encryption, which can only be performed over data as large as the keysize, can take many orders of magnitude more time than a conventional encryption system with a much smaller key.

PGP How-To

This section contains a step-by-step example showing how to use PGP. It assumes that the user already has a working PGP program. Many of the details are left for later sections; this is only a quick explanation of what to do and in what order.

Before You Use PGP

The first step in using PGP is obtaining a PGP binary. Binary distributions are available for platforms such as DOS and Mac. However PGP is only available in source code for Unix and some other systems. As a result, users must compile PGP themselves before it can be used. PGP 2.6.2 has been ported to many operating systems, and it builds cleanly on most Unix systems.

A few items should be collected before PGP is used. You will be able to use PGP after you have done the following:

- Obtained a PGP binary
- Created the PGPPATH directory

- Set the PGPPATH variable
- Chosen a pass phrase

The PGP program depends on some system state to operate. On most platforms, the system state is an environment variable—the PGPPATH environment variable—that tells PGP where to look for its other files.

An environment variable is usually set upon system startup or user initialization depending upon the system in use. On a DOS system, for example, the PGPPATH variable can be set in CONFIG.SYS as follows:

```
set PGPPATH=C:\PGP
```

When using a Unix system, the means to set the environment variable is dependent upon the shell in use. When using the Bourne Shell, PGPPATH is set as follows:

```
PGPPATH=/home/user/.pgp; export PGPPATH
```

When using the C Shell, PGPPATH can be set using setenv, as follows:

```
setenv PGPPATH /home/user/.pgp
```

If you want to keep PGP special files in a special directory, you need to set PGPPATH to point to that directory. PGP will look in the appropriate place for its configuration and data files. By default, PGP will use the current working directory to hold all data files unless PGPPATH is set.

Note The exception is in Unix: PGP will use the .pgp subdirectory of the user's home directory, $HOME/.pgp. This directory must be created by the user. PGP will not create this directory and will print an error if it does not exist.

Next, you need to decide on a pass phrase to use. The pass phrase should be hard to forget and difficult to guess. This pass phrase will be your key to PGP, and knowledge of the pass phrase allows others to create and access messages as if they were you. Think of the pass phrase as the PIN on a bank card; access to a bank account depends entirely on the security of the PIN and access to the bank card. The difference is that there are many more ways to obtain the PGP equivalent of the PIN and bank card than there are in the physical version, but the threat to the user data security can be the same.

The best pass phrases are relatively long and complex. They should contain uppercase and lowercase letters, and each pass phrase should contain some numeric or punctuation characters. A sentence of 8–10 words is long enough to be impossible to guess but short enough that most people should be able to remember it. One way to come up with a sentence is to look in the dictionary at random and combine 8–10 words with articles and punctuation to make a coherent sentence. Once created, this sentence then becomes the pass phrase.

A long pass phrase should be used because it is more difficult to guess a long pass phrase and it is also more difficult to create a program to try every possible pass phrase. For example, if a pass phrase were only eight characters long, it would be simple to write a program to try all eight-character pass phrases. Moreover, this cracking program would run in a reasonably short amount of time.

Assume, for example, an eight-character pass phrase using only letters and numbers. This would mean there are 2×10^{14} possible pass phrases. Assuming one million checks per second, it would take 2×10^{8} seconds, or just under seven years. However, it is still best to choose longer pass phrases to help protect against these attacks.

Generate a PGP Key

The first step in using PGP is to generate a PGP Key. PGP uses these keys when performing operations to secure messages. It is important that you choose an appropriate name for your PGP key and that you choose a memorable pass phrase when creating the key.

Warning When generating a key it is important to remember the pass phrase. The pass phrase is used to lock the secret portion of the key when it is created, and is later used to unlock the key. A forgotten pass phrase cannot be recovered by anyone.

```
~> mkdir .pgp
~> pgp -kg
Pretty Good Privacy(tm) 2.6.2 - Public-Key Encryption for the Masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/14 08:12 GMT
Pick your RSA key size:
1) 512 bits- Low commercial grade, fast but less secure
2) 768 bits- High commercial grade, medium speed, good security
3) 1024 bits- "Military" grade, slow, highest security
Choose 1, 2, or 3, or enter desired number of bits: 3
Generating an RSA Key with a 1024-Bit Modulus.

You need a user ID for your public key.  The desired form for this
user ID is your name, followed by your E-mail address enclosed in
<angle brackets>, if you have an E-mail address.
For example:  John Q. Smith <12345.6789@compuserve.com>
Enter a user ID for your public key: Ruth Thomas <tara@mail.Free.NET>

You need a pass phrase to protect your RSA secret key. Your pass
phrase can be any sentence or phrase and may have many
words, spaces, punctuation, or any other printable characters.

Enter pass phrase:
```

```
Enter same pass phrase again:
Note that key generation is a lengthy process.

We need to generate 784 random bits.  This is done by measuring the
time intervals between your keystrokes.  Please enter some random text
on your keyboard until you hear the beep:
0 * -Enough, thank you.
.......**** .....................................................
..****
Key generation completed.
```

Listing the public key ring would give this output:

```
Type bits/keyID    Date       User ID
pub  1024/D0C6326D 1995/11/14 Ruth Thomas <tara@mail.Free.NET>
```

Distributing the Public Key

After a key has been created, you should obtain the fingerprint and extract the key so that it can be sent to others. Only when someone else has the public key can it effectively be used to sign messages. Moreover, only with the public key can messages be encrypted.

The fingerprint verifies the key.

```
~? pgp -kvc tara
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, that is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/19 05:01 GMT
Key ring: '/tmp/pubring.pgp', looking for user ID "tara".
Type bits/keyID    Date       User ID
pub  1024/D0C6326D 1995/11/14 Ruth Thomas <tara@mail.Free.NET>
Key fingerprint =  B0 22 D9 02 16 25 ED 6E  89 EF 0F 9D A5 5F 9A 1B
```

When a key is extracted, it is copied out of the public key ring into a keyfile that can then be sent to others.

```
~> pgp -kxa tara /tmp/keys.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/19 05:00 GMT

Extracting from key ring: 'pubring.pgp', userid "tara".

Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14
```

```
Transport armor file: /tmp/keys.asc

Key extracted to file '/tmp/keys.asc'.
```

You can distribute the key either via e-mail, finger, the public keyservers, or a number of other means. The keyfile contains your public key certificate . It looks like this:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQCNAzCoUC0AAAEEAM65mYnk+d0i67fpDZbhTT2/xOOKL7umESWW4zYyna22dhHP
BX6mE4oHqy5h1SkXtA4VinQvQWaNWhlTXLo46sEnzNTQKr3hXD5P7O08F4oMjMjT
n5QTG+4Zq6BT1Nh0qN/Fv1rl6JgWEk4bZrBS6sx9JAg1mHjnQkj/XP7QxjJtAAUR
tCBSdXRoIFRob21hcyA8dGFyYUBtYWlsLkZyZWUuTkVUPg==
=327B
-----END PGP PUBLIC KEY BLOCK-----
```

Signing a Message

After your key has been distributed, you can sign messages. A *signature* is a digital stamp on a message that shows others that you have processed that message. A signature can have many meanings, but the important point is that if the message is modified at all, the signature will no longer be valid. For example, a signature can mean that the signer created the message, or it can mean that the signer saw the message, such as a digital notary. A signature enables you to check whether a message is authentic and has not been tampered with in transit.

Another use of a signature is called *non-repudiation*, in which a recipient of a message can prove that the sender actually sent the message. This is useful for signatures on digital contracts, for example, to show that a signature is really valid, rather than forged, on a document.

The following example shows how to sign a file.

```
~> pgp -sat message
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/19 05:17 GMT

A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key. Key for user ID "Ruth Thomas
<tara@mail.Free.NET>"

Enter pass phrase:
Pass phrase is good.
Key for user ID: Ruth Thomas <tara@mail.Free.NET> \\
1024-bit key, Key ID D0C6326D, created 1995/11/14
Just a moment....
Clear signature file: message.asc
```

This message can then be sent to another user:

```
-----BEGIN PGP SIGNED MESSAGE-----

This is a signed message. Actually, this is a clearsigned message.
You can read the contents of the message and also verify the
signature. It should be noted that although you can read this
message, it is not really a plain-text message; it is a PGP file.

The PGP header and footer should not be removed by hand, since the
message may have been quoted by PGP. For example, lines that begin
with a dash (-) or lines that begin with the string: "From " will be
quoted by PGP.

- - this line originally had a leading dash, but PGP added a second one.
------------------------------------------------------------------
Messages should be input to PGP and only the output from PGP, which
is the original message, should be used as input to other processors.
Moreover, only the output of PGP should be trusted to be the signed
message.

-----BEGIN PGP SIGNATURE-----
Version: 2.6.2

iQCVAwUBMK69z0j/XP7QxjJtAQFzJAP/ejfuughrVs7CRGDdRQWEW1QLk4l12qs9
4lvxbGqRcitbfNd/RG98sb1LMsgtmFqFAit+Wi7L5P6P4NHyTTwhvoYtruQ999Hi
cBUoQrT3Lna6q+FElIE7ulH79alaKE9quTq6d3fsW+SghowoMpnTejUUnV+q1DXO
cZ17Jg9fhpY=
=HwTy
-----END PGP SIGNATURE-----
```

Adding Someone Else's Key

Before you can encrypt a message for someone else, you first need to have all the recipient's keys on the public key ring. Through various key management methods, you can obtain other users' keys and, after they are obtained, the keys can be added to the public key ring. For example, a user could add this key block:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQBmAirCXPAAAAECxRpUPos8OENoVWYEkpaZm4YXKu1khXZi/+6UfqPqkMXXASQX
7gqilRqTEMDM1sdq9+n4VWpvXZAktYPmZb3VOBbCmL3JLKDGbCexjjqb62yoMDh0
K1zBsGrxAAURtB5EZXJlayBBdGtpbnMgPHdhcmxvcmATUlULkVEVT6JAJUCBRAu
Y2P/xS1HbQ2/kG0BAahpA/0Zh4oLeYMLFcijLltTo6FuDuPas6eGy+da5lHOPUft
7lgDZ0AdjvEDGiQdAGsIfRjcrKlITQBxjolUZegN9T/C+iPbx6ui3fz8ymeG2yxL
vcl3/neq3mvkzhqLPPjqF9AWLYDBP0Z6l43IpAKpPTtwsoU+lY8L0Qk0mJZSuaef
nYkAVQIFEC4DyWVVBWb6TQxO4QEBSJ8B/jjZ5HTyh3erVBTZ+GuPE7clIfs5YEH/
g2j8eMLTk0gWirUKfwL61RZaD8oIObahsjT0YknEm98py8gvI2tiAXmJAJUCBRAt
yxNVZXmEuMepZt0BATVSA/wLyVgn7mCDITuhT9771JHFMwkUaW7s2hb888Wi4P8u
+tUpoQl9vkmNBQtk/iH5uGBBJIKBLAW5NgA6ixUPDgudXPfDx/G3XG6pHfiH2Sjo
AUVzjHdXUa4+9+Sx5lsx/ZKyg2b6w9eg01iCnHpoEBPIW6l4NbuzI3k7ysbZ9mUd
```

```
sQ==
=KKAa
-----END PGP PUBLIC KEY BLOCK-----
```

Assume that this is in a file called warlord.asc. This file can then be added to the public key ring using the command:

```
~> pgp -ka warlord.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government. \
Current time: 1995/11/19 05:36 GMT

Looking for new keys...
pub   709/C1B06AF1 1992/09/25  Derek Atkins <warlord@MIT.EDU>

Checking signatures...


Keyfile contains:
   1 new key(s)

One or more of the new keys are not fully certified.
Do you want to certify any of these keys yourself (y/N)? No
```

Now the public key ring looks like this:

```
Type bits/keyID    Date       User ID
pub   709/C1B06AF1 1992/09/25 Derek Atkins <warlord@MIT.EDU>
pub  1024/D0C6326D 1995/11/14 Ruth Thomas <tara@mail.Free.NET>
```

Encrypting a Message

When someone else's key is on the key ring, it is simple to encrypt a message to that user. If the file message, for example, contains the plaintext to encrypt, you can encrypt the file to the user "warlord" by using this command:

```
~> pgp -eat message warlord
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/19 05:39 GMT

Recipients' public key(s) will be used to encrypt.
Key for user ID: Derek Atkins <warlord@MIT.EDU>
709-bit key, Key ID C1B06AF1, created 1992/09/25
```

```
WARNING:  Because this public key is not certified with a trusted
signature, it is not known with high confidence that this public key
actually belongs to: "Derek Atkins <warlord@MIT.EDU>".

Are you sure you want to use this public key (y/N)? yes
.
Transport armor file: message.asc
```

The dot (.) on a line by itself is printed by PGP to inform the user that the RSA encryption has proceeded. Because RSA is a slow operation, PGP prints the dot to inform you that it is still processing the message. Otherwise, users might incorrectly believe that PGP is not working. After PGP has finished with the RSA encryption, it writes the output file, message.asc, which can be sent to your recipients:

```
-----BEGIN PGP MESSAGE-----
Version: 2.6.2

hGUDOHQrXMGwavEBAsMEKW8MfmgAA+wLjeQMbWBlQtVTMo9xR/eo3bRODbqcJsZ8
mkNfbGFAXibtP165WI+xNAwjFSYNVZdaH7nFURDd00Aw4wNUzMhEGHQzTjTpYfI6
dnPfurDTjqYAAABwiNTwYTHzmuXJLWUEQSIWIvxfG48uCPgBYQXrSlmf8eRl5RME
F7K8SRs09opqZQwUyLxGEVkwffIiMuvdpezvr4QCSPtBl9OT/Yj34HwYTKQcDOJw
rrAKdtXmU0PglMn8vmudo8VcaRcVL2OpY1aB9g==
=Vmuz
-----END PGP MESSAGE-----
```

Decrypting and Verifying a Message

When a PGP message is received, it must be decrypted and verified before another user can read it. To decrypt a message that you received, you must possess at least one of the secret keys for which the public key was used to encrypt the message. To verify a message, you must have the public key of the signatory on the public key ring.

For example, assume that Ruth Thomas received the following message in the mail. She saved it to the file message.asc.

```
-----BEGIN PGP MESSAGE-----
Version: 2.6.2

hIwDSP9c/tDGMm0BBADB9Yp9oHlgSyt2LM5EcMd6ZWF3MX+qyHX3eQEr3fhAcc2M
PoN+98nldVnmpF5pthU1u/Rj00+t8BaSrho0Qa6in1pyV+2nDR32WUU3wgjmmKCe
Mg1fi+4uD/6bv3TiKEKGJDhtg5YY3NFsirDJ0g6eP+qcX0dApxnbHAYBcAuuIqYA
AAFIm9tP8K9xRqVPeMJfMGpDwhUlZRlFea4906Os24+f7K90YNLwMRs/Kz/QNhYr
8i5LL7ZGs+SjxmqI4FZ8gZkUj7EbuIif4xc8HNbrKXO094TUvvCwBkRXZ4Lv0ukf
5n5032vTgZssTDezRncq2w4OvqcqlmUpMolDWhCFR4zJ7TnC7dpGPIW7/MmxZI+k
Yx4Ov06515Zngj3MgEVEdwu3ATrzkiz/jmP6q+MoSJEP7a4/G87MLHLGgki/hf60
yoLdcG6AQAuIJd4QR3jFpM0wixuUuprJdPM9A2elsLdzZZBqhmNaSACRKPe2y8O1
1GE6pwz+GOE9varZBbehWSXrjj771xMhNOqGUAlXcK938+wX0Cpxu88vFPAeLNyS
7O+GaKpxjg6H2pJ57xeBd+Ozkcyi2YgQiewUtiS0ki6rjA7CwopCyFMoJA==
=TPGz
-----END PGP MESSAGE-----
```

Ruth can try to decrypt this message and print it on-screen:

```
~> pgp -m message.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/19 05:47 GMT

File is encrypted.  Secret key is required to read it.
Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14

You need a pass phrase to unlock your RSA secret key.
Enter pass phrase: Pass phrase is good.  Just a moment.
File has signature.  Public key is required to check signature.
Good signature from user "Derek Atkins <warlord@MIT.EDU>".
Signature made 1995/11/19 05:45 GMT

WARNING:  Because this public key is not certified with a trusted
signature, it is not known with high confidence that this public key
actually belongs to: "Derek Atkins <warlord@MIT.EDU>".
But you previously approved using this public key anyway.


Plaintext message follows...
------------------------------

This message has been signed by Derek Atkins, and is encrypted to the
user Ruth Thomas.  If you are reading this message, then you must have decrypted
it using Ruth's Secret Key.  You can verify this message using Derek's Public Key.

Done...hit any key
Save this file permanently (y/N)? no
```

If Ruth does not plan to read the message right away, but instead wants to decrypt the message onto the disk, she does not have to use the option shown earlier. Instead, the command to decrypt it to disk is as follows:

```
pgp message.asc
```

This command decrypts the contents of the file message.asc and places the output into a file called message. Ruth can read the file later; it has the contents of the original text file.

PGP Keys

Keys are probably the most important concept in PGP. A PGP key is a public keypair that is created by a user for a specific purpose. In general, a user creates a keypair for use as a general contact with the rest of the world. All outgoing messages are signed using this key, and all incoming messages are encrypted using this key. Key management can be a little confusing at first. The following sections will clarify the use and purpose of keys.

What's in a Name?

The previous examples show how easy it is to generate a key. It also shows how easy it is to put any name on a key. The example shows a key being generated in the name of "Ruth Thomas <tara@mail.Free.NET>." It would be just as easy to generate a key in the name of "William Clinton <President@Whitehouse.GOV>." This is not a joke; a key was actually created with this name on it. Of course, it does not belong to the President, but others may not know this if they just see the key on the network.

PGP provides you with a number of ways to name a key. You need to understand how each of the different names can and should be used. You can generate a key with any name on it; this name is called the *userid* on the key.

A key can have many userids on it. In general, a userid has the form Real Name <email@mail.site>, combining the user's real name and e-mail address in a single, compact string. For example, Ruth Thomas created a keypair for herself for use with her Internet address at free.net. As shown earlier, she created a 1,024-bit key on November 14, 1995.

Because the same key can be used with multiple addresses, you might want to have multiple names on the same key to denote its use at multiple sites. You can add userids to your own key by using PGP to edit the key ring. If Ruth wants to use the same key at her other e-mail address, <rthomas@school.edu>, she can add it as a secondary userid on her key.

PGP keys each have another name that you cannot control: the keyid. The *keyid* of a key is a numerical string that is obtained from the key parameters and is used internally by PGP to access the key in question. By design, the keyid is supposed to resemble slightly the actual key, but in reality the keyid differs for each key.

The keyid is a 64-bit quantity, although only 32 bits are printed to the user in hex format. Whenever a userid is required by PGP, the keyid can be used in its place. To specify to PGP that a string is a keyid, it should be prepended with the string "0x", to denote a hex string. Ruth's key can also be called 0xD0C6326D.

The problem with the keyid is that it is currently the lowest 64 bits of the public key modulus. There is a known attack in which someone could generate another keypair of a different size, but with identical keyid and userid. By cursory examination it is difficult to tell which key you are using, and it becomes impossible to tell PGP which key you want because PGP can only index off of the userid and keyid.

Unfortunately there is no defense against this attack at this time. Future versions of PGP may try to handle this case. Because it is relatively easy to create a new key with the same keyID, the need arose for a cryptographically secure fingerprint of a key. This key fingerprint is unique and cannot be easily forged. This value can be used as a key verification string; if the userid, keyid, keysize, and fingerprint all match then a user is sure he or she has the correct key. Key fingerprints can be trusted because they are made with the same hash algorithm, MD5, that PGP uses for message integrity.

However, matching the numeric values on a key is not good enough to trust that key. It also becomes important to check the name on the key. Anyone can create a key that says that it belongs to the President, however it is highly unlikely that any of those keys actually belong to the Commander in Chief. Therefore, you, the user, must use other means to validate the name on a key. How to validate a key is covered in the section, "The Web of Trust."

PGP Key Rings

PGP requires users to keep a local cache of keys. This cache is called the user's key ring. Each user has at least two key rings: a public key ring and a secret key ring. Each key ring is used to store a set of keys that are used for specific purposes. It is important to keep both key rings secure, however; tampering with a public key ring can cause you to incorrectly verify signatures or encrypt messages to the wrong recipients.

Public Key Rings

The *public key ring* stores all the public keys, userids, signatures, and trust parameters for all the parties with whom you communicate. Whenever PGP looks for a key to verify a signature or encrypt a message, it looks in your public key ring. This means that you have to keep their public key ring up to date, either by frequently asking communiques to update your keys, or by accessing the PGP Public Keyservers.

Trust parameters are stored in the public key ring, so it is not feasible to share key rings between people. Moreover, PGP does not handle multiple key rings properly, so creating a site-wide key ring to store keys is not easy to do with the current releases. This is a known bug in PGP. Until multiple key rings are supported in a future version, the best way to distribute keys is to use a keyserver. One security concern with public key rings is that a compromised public key ring can lead to false positive signature verification or, worse, encrypted messages for the wrong parties. An attacker could change the trust parameters that are stored in the public key ring, or change the actual key material stored therein. These attacks are described in detail in the section, "Public Key Ring Attacks."

When it was designed, the key rings were meant to hold only a few keys of close friends and associates. Unfortunately, it is clear from current usage that this design assumption is limited. Many people keep their key ring full of keys for people whom they have never met and with whom they have never communicated. Unfortunately this can cause problems, mostly due to replication of information and the time required to access the key ring. The recommended procedure is to keep the key ring as small as possible, and fetch required keys as necessary from a keyserver or site-wide key ring.

Secret Key Rings

The secret key ring is where personal secrets are stored for PGP. When you generate a key, the parts that you must not divulge are stored in the secret key ring. The data that needs to be kept private is encrypted, so access to the secret key ring does not automatically grant use of its secrets. However, if an attacker can gain access to the secret key ring, he or she has one less obstacle in the way to forge signatures and decrypt messages.

Because secret keys are not transmitted between people, the only keys that are supposed to be on a user's secret key ring are his or her own secret keys. Because secret keys are protected by a pass phrase, simple transmission of the contents of a secret key ring will not allow access to the key material.

It is not recommended to share a secret key between parties, although at times it might be required. In particular, when you have a secret key that belongs to an organization, it might be worthwhile for multiple members of that organization to have access to the secret key. This means that any single individual can act fully on behalf of that organization, however.

Sometimes it might be useful to have a secret key without a pass phrase. For example, it might be worthwhile to have a server with a secret key acting on behalf of a group of people. In particular, you could run an encrypted mailing list in which the mailserver has its own key, and has the public keys for all list members. List members encrypt messages in the mailserver's key and mail it to the list. The list processor decrypts the message and then re-encrypts it for each list member using his or her public keys. At this point the list server could sign the message with the list key, but that is not necessary. In such a situation, where a server process needs access to a secret key, it is desirable to have no pass phrase on the key.

Because it is possible to have multiple secret keys on a secret key ring, PGP has an option to specify the userid of the secret key you want to use. Whenever PGP needs to choose a secret key to use, it will choose the first key on the key ring, which is usually the most recent key to be created. You can override this by supplying the userid to PGP using the -u option, and it will use the secret key that has the appropriate userid.

The Web of Trust

It is said that, using the appropriate intermediaries, it takes six handshakes to get from any one person on earth to any other person on earth. This is a web of introducers, where each person acts as an introducer to the next person in the chain. PGP uses a similar method to introduce new keys, using key signatures as a form of introduction. When someone signs a key, he or she become a potential introducer for that key. For example, suppose Alice signs Bob's key, and Bob signs Charlie's key. Alice now has a certification path to Charlie. Alice now has a means of knowing that Charlie's key really is Charlie's because it has a signature of Bob on it, and Alice knows that Bob's key really belongs to Bob. This is a way to provide transitive trust in keys.

There is clearly a problem in this design. What happens if someone is acting as an introducer but does not really know the person he claims to know? For example, what if Bob is completely careless and signed Doug's key, even though it claimed to be Charlie's. Not only would Bob think that this key belongs to Charlie (even though it is Doug claiming to be Charlie), but if there were no measurement of trust, Alice would believe it, too.

This is where the PGP Web of Trust comes into play. With the Web of Trust, users define the amount of trust they put into a key to act as an introducer for them. In the preceding example, Alice can put as much trust as she wants in Bob's key, and should only trust a key if she trusts Bob to sign other's keys correctly. If Alice knows that Bob is lax about verifying keys, she would clearly not trust Bob to act as an introducer. As a result Alice would not trust the key that Bob signed for Doug, claiming to be Charlie.

Of course, the Web of Trust is not foolproof. If someone is fooled into signing a wrong key, it can cause others to believe it incorrectly. The PGP Web of Trust can be thought of as a reputation system, where people are reputed to give good signatures, and others are reputed to give bad signatures. The system can fail when false positive reputations exist.

Degrees of Trust

The Web of Trust starts with a user's own keypair. PGP assumes that if you have the secret key for a keypair, you can trust it. This is because you can verify the key at any time by creating a signature and verifying it. This is called *Ultimate Trust*. Any keys signed by an Ultimately Trusted key are trusted to be valid keys.

For each valid key, the user is asked to assign a level of trust in that key. This trust value defines how much the user trusts that key as an introducer. This can get confusing because PGP uses the same terms to define trust in a key's validity as it uses to define the amount of trust as an introducer. There are four levels of trust:

- Complete trust
- Marginal trust
- No trust
- Unknown trust

In addition to defining trust in keys as introducers, users define the number of "completes" and "marginals" needed to trust the validity in a key. By default, PGP requires one complete or two marginal signatures, where a complete signature is a signature by a key that is completely trusted as an introducer, and a marginal signature is a signature by a key that is marginally trusted as an introducer. These values can be set by the user to define how many complete and marginal signatures are required to trust the validity of a key.

This process continues until a user-defined level is reached. The default value is four levels of recursion, or nesting, in the search of the key ring. If Alice signs Bob, Bob signs Charlie, Charlie signs Dave, Dave signs Elena, and Elena signs Frank, Alice could only get as far as Elena, and could not trust Frank because there are too many steps. Moreover, this all depends on the trust that Alice has in all of the signers in the line. In general, it is not recommended to put trust in keys belonging to users you do not know.

Key Management

To manage keys, PGP has developed an extensive set of key management functions. Many would say that this is the most confusing part of PGP, which is probably right. However, PGP key management is not so complicated that it takes a Unix guru to understand it. With some time exploring and with some careful explanations, anyone can understand it.

The important point regarding key management is that all PGP key management functions are invoked by PGP command lines that begin with the -k option. The arguments listed in table 11.1 follow this option and tell PGP which key management function is requested. Arguments listed with brackets are optional.

Table 11.1
Key Management Functions

Option	Description
pgp -kg [length] [ebits] [-u userid]	Generates your own unique public/secret key pair
pgp -ka keyfile [key ring]	Adds a key file's contents to your public or secret key ring
pgp -kx userid keyfile [key ring]	Extracts (copies) a key from your public or secret key ring
pgp -ks her_userid [-u your_userid] [key ring]	Signs someone else's public key on your public key ring
pgp -kv[v] [userid] [key ring]	Views the contents of your public key ring
pgp -kc [userid] [key ring]	Checks signatures on your public key ring
pgp -kr userid [key ring]	Removes a key or a user ID from your public or secret key ring
pgp -krs userid [key ring]	Removes selected signatures from a userid on a key ring

Option	Description
pgp -kvc [userid] [key ring]	Views fingerprints for keys on your key ring
pgp -kd userid [key ring]	Disables or revokes a key
pgp -ke your_userid [key ring]	Edits your user ID or pass phrase

Key Generation

The first thing any PGP user needs to do is create a keypair. When you generate a key (that is, an RSA keypair), you are asked for the keysize, the name on the key, a pass phrase, and then for some random keystrokes. The key parameters are used to generate the actual bits that will be your PGP key.

The keysize is directly proportional to the security of the key, and indirectly proportional to the time it takes to use that key. Larger keys are more secure, but they require more time to use. Because the time differential affects only the key owner, a key owner who wants a longer key will pay the penalty himself, whereas everyone else who uses that key will see a marginal penalty.

The name on the key is the userid. It is the printable string that is supposed to tell others who owns this key. By convention, the userid is a name and an e-mail address, such as the string Derek Atkins <warlord@MIT.EDU>. A key can have multiple names, which means that its owner has different names.

After the key parameters have been defined, PGP will ask the user for a pass phrase. This pass phrase will later be used to unlock the secret key. This provides an extra level of security when the secret key is used because the pass phrase is required to sign or decrypt messages using that keypair. Through the pass phrase, an attacker who obtains the on-disk portion of the secret key ring cannot use its contents because they are encrypted using the pass phrase. An attacker needs to have the contents of the secret key ring and the pass phrase in which is it encrypted to steal the secret key.

After the pass phrase, PGP asks for random keystrokes. These keystrokes are timed, and the inter-keystroke timing is used to generate random numbers. These random numbers are used to generate the primes that comprise the RSA keypair. The longer the keypair, the more random data that is required to generate it, and the more keystrokes are required.

To generate a key, use the -kg option to PGP. The first example is a repeat of the first example in this chapter, but each step is explained. First, the user must create the directory to hold the keypair. PGP uses the PGPPATH environment variable to hold the name of this directory. If PGPPATH is not set, PGP will use a reasonable default. In DOS PGP will use the current working directory; in Unix it will use the .pgp directory in the user's home directory. Because PGP does not make this directory, the user needs to create it first:

```
~> mkdir .pgp
```

After the PGPPATH directory is created, the user can generate a key. PGP will prompt for all the information that is required. A key is generated by using the -kg option:

```
~> pgp -kg
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/14 08:12 GMT
Pick your RSA key size:
1) 512 bits- Low commercial grade, fast but less secure
2) 768 bits- High commercial grade, medium speed, good security
3) 1024 bits- "Military" grade, slow, highest security
Choose 1, 2, or 3, or enter desired number of bits: 3
```

At this point, PGP wants to know the size of the key to generate. PGP will present you with three built-in sizes: 512, 768, and 1,024 bits. The larger the keysize, the more secure the key will be but the longer it will take the user to actually use the key. Although it is technically feasible to use arbitrarily large keys, the time it would take to actually perform various options using very large keys far outweighs the security benefit of the use of the larger key.

In the preceding sample command list, the user has chosen the built-in keysize of 1,024 bits by choosing option 3. Alternatively, the user could have typed in the actual size of the key to generate. PGP will generate keys of any length between 384 and 2,048 bits in length. A user need only type the number of bits requested instead of the built-in values.

```
Generating an RSA key with a 1024-bit modulus.

You need a user ID for your public key.  The desired form for this
user ID is your name, followed by your E-mail address enclosed in
<angle brackets>, if you have an E-mail address.
For example:  John Q. Smith <12345.6789@compuserve.com>
Enter a user ID for your public key:
Ruth Thomas <tara@mail.Free.NET>
```

Creating the PGP userid

The next piece of information is the userid on the key. The userid should be a string that contains the name of the user of the key as well as an electronic address where that user can be reached. The suggested format appears in the preceding sample command list: the user's name followed by the e-mail address in angle-brackets.

Next, PGP will ask for a pass phrase. The pass phrase is used by PGP to encrypt the secret key before it is written to disk. Later, the user will be required to type the pass phrase before he or she can use the secret key to sign or decrypt messages. A lost pass phrase cannot be recovered; for this reason, it is imperative that users choose a pass phrase that is easy to remember. Never write down the pass phrase.

Warning Choose a pass phrase that is easy to remember and hard to guess. PGP accepts pass phrases over 100 characters long, which provides you with enough space to make pass phrases as long as you want. The longer the pass phrase, the harder it is to brute force by trying all possible keys. Good pass phrases consist of both upper- and lowercase letters and some punctuation and numeric characters. A medium-length sentence with capitalization and punctuation usually makes a good pass phrase.

It is important that users do not forget their pass phrases. A secret key cannot be recovered if the pass phrase is lost. Nothing in the world can be done for a user who forgets his or her pass phrase. Make sure that pass phrases can be remembered. One of the benefits of having such a long pass phrase is that it can be English words in a meaningful English sentence; which makes remembering the phrase much simpler.

```
You need a pass phrase to protect your RSA secret key.
Your pass phrase can be any sentence or phrase and may have many
words, spaces, punctuation, or any other printable characters.

Enter pass phrase:
Enter same pass phrase again:
Note that key generation is a lengthy process.

We need to generate 784 random bits.  This is done by measuring the
time intervals between your keystrokes.  Please enter some random text on your
keyboard until you hear the beep:
   0 * -Enough, thank you.
............**** ......................................................
..****
Key generation completed.
```

After the pass phrase is entered, PGP will ask for a lot of random keystrokes. While the user types the keystrokes, it measures the inter-keystroke timings to get random data. Because people type at an inconsistent speed, PGP can use the time between each keystroke and use the variance as a source of randomness. It then uses that randomness to generate two large prime numbers, which become the RSA keypair. A user can specify almost all the appropriate data on the command-line. The following example will generate a key of only 512 bits, which is a relatively insecure length. The name of the President is used as the userid to show how easy it is to create a key in someone else's name.

```
~> pgp -kg 512 -u 'William Clinton <President@Whitehouse.GOV>'
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
```

```
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/14 08:16 GMT
Generating an RSA key with a 512-bit modulus.
Generating RSA key-pair with UserID "William Clinton <President@Whitehouse.GOV>".
\\

You need a pass phrase to protect your RSA secret key.
Your pass phrase can be any sentence or phrase and may have many
words, spaces, punctuation, or any other printable characters.

Enter pass phrase:
Enter same pass phrase again:
Note that key generation is a lengthy process.

We need to generate 576 random bits.  This is done by measuring the
time intervals between your keystrokes.  Please enter some random text on your
keyboard until you hear the beep:
   0 * -Enough, thank you.
............**** .......................................................
..****
Key generation completed.
```

This key ring now looks like this:

```
Type bits/keyID    Date       User ID
pub   512/97D45291 1995/11/14 William Clinton <President@Whitehouse.GOV>
pub   709/C1B06AF1 1992/09/25 Derek Atkins <warlord@MIT.EDU>
pub  1024/D0C6326D 1995/11/14 Ruth Thomas <tara@mail.Free.NET>
```

Adding Keys to the Public Key Ring

To use a key to encrypt a message or verify a signature, it must be on a public key ring. There are a number of ways to acquire a key, and they all involve just a few steps. The first step is to obtain the key. This can be done via e-mail, ftp, a keyserver, a floppy, or by typing the key. After you have a key, you tell PGP to add it to your key ring using the -ka option.

When you first use PGP, it is helpful to add the keys that are in the PGP release to your personal key ring. One reason is that the PGP release is signed by at least one of these keys, usually; adding the key to the public key ring enables users to check the signature on the PGP distribution. The keys are held in a file called keys.asc:

```
~> pgp -ka keys.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 18:01 GMT
```

```
Looking for new keys...
pub  1024/0DBF906D 1994/08/27  Jeffrey I. Schiller <jis@mit.edu>
pub   512/4D0C4EE1 1992/09/10  Jeffrey I. Schiller <jis@mit.edu>
pub  1024/0778338D 1993/09/17  Philip L. Dubois <dubois@csn.org>
pub  1024/FBBB8AB1 1994/05/07  Colin Plumb <colin@nyx.cs.du.edu>
pub  1024/C7A966DD 1993/05/21  Philip R. Zimmermann <prz@acm.org>
pub  1024/8DE722D9 1992/07/22  Branko Lankester  <branko@hacktic.nl>
pub  1024/9D997D47 1992/08/02  Peter Gutmann <pgut1@cs.aukuni.ac.nz>
pub  1019/7D63A5C5 1994/07/04  Hal Abelson <hal@mit.edu>

Checking signatures...
pub  1024/0DBF906D 1994/08/27  Jeffrey I. Schiller <jis@mit.edu>
sig!      C7A966DD 1994/08/28  Philip R. Zimmermann <prz@acm.org>
sig!      C1B06AF1 1994/08/29  Derek Atkins <warlord@MIT.EDU>
sig!      4D0C4EE1 1994/08/27  Jeffrey I. Schiller <jis@mit.edu>
pub   512/4D0C4EE1 1992/09/10  Jeffrey I. Schiller <jis@mit.edu>
sig!      4D0C4EE1 1994/06/27  Jeffrey I. Schiller <jis@mit.edu>
sig!      C1B06AF1 1994/06/19  Derek Atkins <warlord@MIT.EDU>
sig!      C7A966DD 1994/05/07  Philip R. Zimmermann <prz@acm.org>
pub  1024/0778338D 1993/09/17  Philip L. Dubois <dubois@csn.org>
sig!      C7A966DD 1993/10/19  Philip R. Zimmermann <prz@acm.org>
pub  1024/FBBB8AB1 1994/05/07  Colin Plumb <colin@nyx.cs.du.edu>
sig!      C7A966DD 1994/05/07  Philip R. Zimmermann <prz@acm.org>
sig!      FBBB8AB1 1994/05/07  Colin Plumb <colin@nyx.cs.du.edu>
pub  1024/C7A966DD 1993/05/21  Philip R. Zimmermann <prz@acm.org>
sig!      0DBF906D 1994/08/30  Jeffrey I. Schiller <jis@mit.edu>
sig!      4D0C4EE1 1994/05/26  Jeffrey I. Schiller <jis@mit.edu>
sig!      C7A966DD 1994/05/07  Philip R. Zimmermann <prz@acm.org>
pub  1024/8DE722D9 1992/07/22  Branko Lankester  <branko@hacktic.nl>
sig!      C7A966DD 1994/05/07  Philip R. Zimmermann <prz@acm.org>
sig!      8DE722D9 1993/11/06  Branko Lankester  <branko@hacktic.nl>
pub  1024/9D997D47 1992/08/02  Peter Gutmann <pgut1@cs.aukuni.ac.nz>
sig!      C7A966DD 1994/02/06  Philip R. Zimmermann <prz@acm.org>
pub  1019/7D63A5C5 1994/07/04  Hal Abelson <hal@mit.edu>
sig!      0DBF906D 1994/09/03  Jeffrey I. Schiller <jis@mit.edu>
sig!      C7A966DD 1994/07/28  Philip R. Zimmermann <prz@acm.org>
pub   709/C1B06AF1 1992/09/25  Derek Atkins <warlord@MIT.EDU>
sig!      0DBF906D 1994/08/30  Jeffrey I. Schiller <jis@mit.edu>
sig!      4D0C4EE1 1994/06/19  Jeffrey I. Schiller <jis@mit.edu>
sig!      C7A966DD 1994/05/07  Philip R. Zimmermann <prz@acm.org>


Keyfile contains:
   8 new key(s)

One or more of the new keys are not fully certified.
Do you want to certify any of these keys yourself (y/N)? No
```

After a new key is added, you usually are asked if you want to certify it, or sometimes how much trust should be put in a key to sign other keys. When you sign a key, you make a statement about the authenticity of that key. A signature states that you believe that the userid on the key actually names the user or group who has the secret key.

Users should never sign arbitrary keys. You should never sign a key without first verifying its authenticity by using the key fingerprint and talking to the key's owner. Whether a key should be trusted as an introducer is really a question in your trust in the key and the owner of the key. Do you believe that this key really belongs to the person whose userid is on the key? Do you know this person? Do you trust this person to sign other keys properly? Do you know if the user is easily spoofed? How much do you trust him or her to sign keys consistently? Ask yourself these questions before trusting a key as an introducer.

Extracting Keys from the Public Key Ring

To exchange PGP Public keys, you exchange PGP keyfiles. A keyfile is similar to a key ring, except that it has no trust information or other bits that might be considered personal or confidential.

When extracting keys from a key ring into a keyfile, PGP will extract exactly one key, or every key, into the keyfile. PGP will extract the first key that matches the userid. For example, Ruth could extract her key using the following command:

```
~> pgp -kxa tara tara.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 18:10 GMT

Extracting from key ring: '/tmp/pubring.pgp', userid "tara".

Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14

Transport armor file: tara.asc

Key extracted to file 'tara.asc'.
```

Ruth now has a file called tara.asc that contains her public key. She can send this key to other people using e-mail, netnews, the keyservers, or any other key distribution mechanism. Sometimes it is useful to extract the whole key ring into a keyfile. For example, a key ring can be extracted to move it from one location to another. To extract the whole key ring into a keyfile, use a null userid, which can be obtained by using two sets of quotes:

```
~> pgp -kxa "" mykeys.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 18:13 GMT
```

```
Transport armor file: mykeys.asc

Key extracted to file 'mykeys.asc'.
```

The file mykeys.asc now contains the full contents of the key ring and can be sent to anyone just like in the previous example. In both of these examples, it was known that the key or keys were being extracted so that they could be sent elsewhere. Sometimes, however, it is necessary to extract a key in a form on which PGP can operate.

PGP can treat keyfiles like key rings because the formats are the same. Usually, this distinction is clear and important. At times, however, the distinction between key rings and keyfiles should be overlooked. When you want to treat a keyfile as a key ring, it must be in binary format. This means that you cannot use the -a option when generating the keyfile/key ring.

If you want to send a subset of a key ring in a single keyfile, for example, you need to extract each key, one-by-one, into a keyfile. PGP treats this keyfile as a key ring and all the keys can be extracted into another keyfile to send. The following example shows a simple Unix shell script that extracts a set of keys into a file named keys.asc, which can be sent via e-mail to someone else. First PGP extracts the requested keys into a keyfile called keyfile.pgp. Next, PGP treats that keyfile as a key ring and extracts the keys into an armored keyfile called keys.asc.

```
#!/bin/sh
rm -f keyfile.pgp
for user in user1 user2 user3; do
        pgp -kx $user keyfile.pgp;
        done
pgp -kxa " keys.asc keyfile.pgp
rm -f keyfile.pgp
```

You can now e-mail the output file, keys.asc, to the intended recipients so that they can add it to their key rings using pgp -ka.

Signing Keys

A signature on a key is an important statement that a user can make about that key. In general, a signature on a key means that the signer has verified, to some degree, that the key actually belongs to the user whose userid is on the key. PGP uses signatures to build up trust in a key. In general, the more signatures on a key, the more likely that it will be trusted. The mere existence of signatures on a key, however, is not enough to force PGP to trust the key as valid.

A key signature is a binding between the key parameters (the RSA modulus and exponent, in the RSA case) with the userid that is being signed. If userids are added or changed, the signature will fail. Users should never sign a key without first verifying it. Methods of verification are discussed in the section, "Key Fingerprints and Verifying Keys."

When a key has been verified, a user may choose to sign it. Signing a key involves using a secret key to sign the public key parameters and the userid of the public key to be signed. To sign a key, use the -ks option in PGP. For example, Ruth could sign the key of the userid warlord in this manner:

```
~> pgp -ks warlord
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 18:42 GMT

Looking for key for user 'warlord':

Key for user ID: Derek Atkins <warlord@MIT.EDU>
709-bit key, Key ID C1B06AF1, created 1992/09/25
          Key fingerprint =  A0 9A 7E 2F 97 31 63 83  C8 7B 9C 8E DE 0E 8D F9


READ CAREFULLY: Based on your own direct first-hand knowledge, are
you absolutely certain that you are prepared to solemnly certify that
the above public key actually belongs to the user specified by the
above user ID (y/N)? yes

You need a pass phrase to unlock your RSA secret key.
Key for user ID "Ruth Thomas <tara@mail.Free.NET>"

Enter pass phrase: Pass phrase is good.  Just a moment....
Key signature certificate added.
```

Next, PGP will go through the key ring and validate the trust parameters of the keys. Because Ruth's own key is ultimately trusted, Ruth's signature implies that warlord's key is valid to Ruth. In other words, PGP makes the assertion that a user who signs keys will not fool him- or herself into signing false keys. With this method, a signature by a user's own key is enough to trust its validity.

When keys become trusted as valid, the keys can then act as introducers. PGP examines the key ring and asks you to place a trust on valid keys. How much do you trust a key to sign other keys? For each valid key, PGP will ask this question. Using these answers, more keys can become trusted as valid, and so on. This is how the web of trust is built.

For each valid key, PGP enables you to specify four trust values that specify how much you trust the key as an introducer. A value of one (1) means that you do not know how much trust to place in the key. Therefore that key is not used to compute validity trust values. A trust value of two (2) means that you do not trust the key as an introducer. When these values are used on a valid key, PGP ignores signatures on other keys made by this one, so these values apply nothing towards the trust in another key.

The trust value of three (3) denotes marginal trust in a key acting as an introducer; a value of four (4) denotes complete trust in a key acting as an introducer. PGP will add together the number of completely trusted signatures and marginally trusted signatures and compare the values to the number of completes and marginals needed to fully trust a key as valid. By default, PGP requires one completely trusted signature or two marginally trusted signatures to validate a key. These numbers can be changed through two configuration file options: COMPLETESNEEDED and MARGINALS_NEEDED.

```
Make a determination in your own mind whether this key actually
belongs to the person whom you think it belongs to, based on available
evidence.  If you think it does, then based on your estimate of
that person's integrity and competence in key management, answer
the following question:

Would you trust "Derek Atkins <warlord@MIT.EDU>"
to act as an introducer and certify other people's public keys to you?
(1=I don't know. 2=No. 3=Usually. 4=Yes, always.) ? 4

Make a determination in your own mind whether this key actually
belongs to the person whom you think it belongs to, based on available evidence.
If you think it does, then based on your estimate of that person's integrity and
competence in key management, answer the following question:

Would you trust "Jeffrey I. Schiller <jis@mit.edu>"
to act as an introducer and certify other people's public keys to you?
(1=I don't know. 2=No. 3=Usually. 4=Yes, always.) ? 4
```

Sometimes users are known personally and they can be trusted to sign keys properly. When this is the case, you can assign a trust value on that key to always sign keys properly. In general, this trust value should be used on keys for which you have validated the owner and when you know the other user to be trustworthy. For example, Ruth could have visited MIT and met both Derek and Jeff. During this meeting, she determined that both are completely trustworthy and decided that they will always sign keys properly.

Occasionally PGP will ask whether a key can be used as an introducer even when you do not know the owner. In this case, you should choose how much trust you have in the key owner, even though you haven't met him or her. In general, it is best not to put complete trust in a key of an unknown individual. If Ruth had never met Phil Zimmermann, and if she never had the chance to learn his signing habits, she might only have marginal trust in the key, which she can indicate by choosing the value of trust she wants to place on the key. The next part of this example outlines the trust settings for the individual Ruth has never met:

```
Make a determination in your own mind whether this key actually
belongs to the person whom you think it belongs to, based on
available evidence.  If you think it does, then based on your
estimate of that person's integrity and competence in key management,
answer the following question:

Would you trust "Philip R. Zimmermann <prz@acm.org>"
to act as an introducer and certify other people's public keys to you?
(1=I don't know. 2=No. 3=Usually. 4=Yes, always.) ? 3
```

```
Make a determination in your own mind whether this key actually
belongs to the person whom you think it belongs to, based on available evidence.
If you think it does, then based on your estimate of that person's integrity and
competence in key management, answer the following question:

Would you trust "Jeffrey I. Schiller <jis@mit.edu>"
to act as an introducer and certify other people's public keys to you?
(1=I don't know. 2=No. 3=Usually. 4=Yes, always.) ? 2
```

The choices of trust are personal value judgments based both on the key and the key's owner. Sometimes you may have multiple keys but only one of them would be useful to someone else. In the example you've followed in this chapter, Ruth should assign no trust to Jeff's second key because it is an old key that has been replaced by a new one. Unfortunately, PGP does not inform you that a key is a duplicate of another key with the same name, so you need to be aware of situations that may have multiple keys with the same name on them.

Viewing the Contents of a Key Ring

Many times it is useful to see what keys exist on a key ring. PGP enables users to view key rings in multiple formats. The first format, -kv, is a short format, where only the key information and userids are printed. The second format, -kvv, is the long format, and it also shows signatures on keys.

```
~> pgp -kv
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 19:02 GMT

Key ring: '/tmp/pubring.pgp'
Type bits/keyID    Date       User ID
pub  1024/0DBF906D 1994/08/27 Jeffrey I. Schiller <jis@mit.edu>
pub   512/4D0C4EE1 1992/09/10 Jeffrey I. Schiller <jis@mit.edu>
pub  1024/0778338D 1993/09/17 Philip L. Dubois <dubois@csn.org>
pub  1024/FBBB8AB1 1994/05/07 Colin Plumb <colin@nyx.cs.du.edu>
pub  1024/C7A966DD 1993/05/21 Philip R. Zimmermann <prz@acm.org>
pub  1024/8DE722D9 1992/07/22 Branko Lankester  <branko@hacktic.nl>
pub  1024/9D997D47 1992/08/02 Peter Gutmann <pgut1@cs.aukuni.ac.nz>
pub  1019/7D63A5C5 1994/07/04 Hal Abelson <hal@mit.edu>
pub   512/97D45291 1995/11/14 William Clinton <President@Whitehouse.GOV>
pub   709/C1B06AF1 1992/09/25 Derek Atkins <warlord@MIT.EDU>
pub  1024/D0C6326D 1995/11/14 Ruth Thomas <tara@mail.Free.NET>
11 matching keys found.
```

One interesting quirk you should know about the user interface is that PGP will print out all keys that match the userid, whereas most other functions will choose the first key that matches the userid. In other words, the userid is treated as a substring that is matched against the keys in the key ring. This capability lets you print out a set of keys. For example, you can print out all the keys for people at mit.edu.

```
~> pgp -kvv mit.edu
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 19:05 GMT

Key ring: '/tmp/pubring.pgp', looking for user ID "mit.edu".
Type bits/keyID    Date       User ID
pub  1024/0DBF906D 1994/08/27  Jeffrey I. Schiller <jis@mit.edu>
sig       C7A966DD             Philip R. Zimmermann <prz@acm.org>
sig       C1B06AF1             Derek Atkins <warlord@MIT.EDU>
sig       4D0C4EE1             Jeffrey I. Schiller <jis@mit.edu>
pub   512/4D0C4EE1 1992/09/10  Jeffrey I. Schiller <jis@mit.edu>
sig       4D0C4EE1             Jeffrey I. Schiller <jis@mit.edu>
sig       C1B06AF1             Derek Atkins <warlord@MIT.EDU>
sig       C7A966DD             Philip R. Zimmermann <prz@acm.org>
pub  1019/7D63A5C5 1994/07/04  Hal Abelson <hal@mit.edu>
sig       0DBF906D             Jeffrey I. Schiller <jis@mit.edu>
sig       C7A966DD             Philip R. Zimmermann <prz@acm.org>
pub   709/C1B06AF1 1992/09/25  Derek Atkins <warlord@MIT.EDU>
sig       D0C6326D             Ruth Thomas <tara@mail.Free.NET>
sig       0DBF906D             Jeffrey I. Schiller <jis@mit.edu>
sig       4D0C4EE1             Jeffrey I. Schiller <jis@mit.edu>
sig       C7A966DD             Philip R. Zimmermann <prz@acm.org>
4 matching keys found.
```

You can also list every key in a key ring other than the default. Leaving off the userid works for the default key ring. However, if an alternate key ring is supplied, you need to supply a userid. A NULL userid, “ ”, will match all keys, which will list the full contents.

Removing Keys and Signatures

Occasionally an extra key will be added to a key ring, or keys will have unverifiable signatures on them. Although these data on the key ring cannot cause any problems, it is sometimes useful to remove extraneous keys and signatures to reduce the size of data sent to others.

Fortunately, PGP lets you remove keys and signatures from keys in a key ring. The key management function -kr removes a key; the function -krs lets you remove the signatures on a key. PGP will first ask if you want to proceed to make sure you really want to remove the data. At times, PGP will walk you through to the appropriate key to find the exact data you want to remove.

When you remove a key, specify the userid of the key you want to remove. For example, two keys exist for Jeffrey I. Schiller on the key ring and Ruth wants only the most recent key. She wants to remove his second key. Unfortunately, both keys have the same name, so she needs to specify the keyid of the key to remove:

```
~> pgp -kr 0x4D0C4EE1
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 23:08 GMT

Removing from key ring: '/tmp/pubring.pgp', userid "0x4D0C4EE1".

Key for user ID: Jeffrey I. Schiller <jis@mit.edu>
512-bit key, Key ID 4D0C4EE1, created 1992/09/10

Are you sure you want this key removed (y/N)? yes

Key removed from key ring.
```

When you remove a signature, specify the userid of the key that incorporates the signature. For each signature on that key, PGP will ask whether it should be removed. When Ruth removed Jeffrey's key, some unknown signatures were left on the key ring. She now needs to remove the extraneous signatures on the keys. For example, an extra signature exists on the key for Derek Atkins; Ruth needs to remove this extra signature:

```
~> pgp -krs warlord
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 23:13 GMT

Removing signatures from userid 'warlord' in key ring
'/tmp/pubring.pgp'

Key for user ID: Derek Atkins <warlord@MIT.EDU>
709-bit key, Key ID C1B06AF1, created 1992/09/25

Key has 4 signature(s):
sig      D0C6326D             Ruth Thomas <tara@mail.Free.NET>
Remove this signature (y/N)? <Enter>
sig      0DBF906D             Jeffrey I. Schiller <jis@mit.edu>
Remove this signature (y/N)? no
sig      4D0C4EE1             (Unknown signator, can't be checked)
Remove this signature (y/N)? yes
sig      C7A966DD             Philip R. Zimmermann <prz@acm.org>
Remove this signature (y/N)? <Enter>

1 key signature(s) removed.
```

When you remove signatures from a key, PGP will ask you whether each signature, in turn, should be removed. The default answer is no; press Enter to move to the next signature.

Key Fingerprints and Verifying Keys

The most important part of the key verification process is knowing whether the person or entity behind the userid actually has the secret key of this keypair. This is an important concept, and should not be taken lightly. It is not important that the name on the key be the actual name of the person who uses the key; what is important is that the person using the key can be reached using the name on the key, and has the secret part of the key.

The best way to know whether a key is correct is to watch it being created. This remedy, however, isn't that realistic. The next best way to verify a key and its owner is to have the key owner give you the key in person, on a floppy disk. This process requires that you know the person, can meet him or her in person, or can match the key to the individual by name. These methods are called *in-band key verification*, in which you get the key and verification information at the same time using the same key distribution methods.

PGP provides another way to verify a key *out of band*. You can use any key distribution method to obtain the key, such as by downloading it from an untrusted keyserver, and then verify the key using the trusted information. This way you can obtain key verification out of band, either through a phone call, a letter, or some other means of communicating with the other party, regardless of what key distribution method is used.

Sometimes a key is validated inappropriately. Either a key was changed in transit, or a user was fooled by social engineering to validate a key. Social engineering is where an attacker uses social means, such as posing as someone else, in order to gain the desired results. In such cases, the falsely validated key can wreak havoc among users who trust the signer's fooled owner. Unfortunately, there is no automatic means to verify the verification.

The most secure way to get this information is when the userid on a key matches the real name of a person. It is possible for that person to supply documents verifying his identity, and then provide a means to verify the key he is presenting as his own. The way to verify a key is through the key fingerprint.

A *key fingerprint* is a cryptographic hash of the key parameters of a public key, printed in a form that is easy to write down, copy, or speak. To obtain a key fingerprint, PGP is called with the -kvc option.

```
~> pgp -kvc warlord
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 23:26 GMT

Key ring: '/tmp/pubring.pgp', looking for user ID "warlord".
Type bits/keyID    Date       User ID
pub   709/C1B06AF1 1992/09/25 Derek Atkins <warlord@MIT.EDU>
```

```
          Key fingerprint =  A0 9A 7E 2F 97 31 63 83  C8 7B 9C 8E DE 0E 8D F9
1 matching key found.
```

First, the key owner obtains the fingerprint when the key is created and writes it down. Then, when anyone wants to verify the key, he or she contacts the key owner who transfers the fingerprint. Then the end user can check the fingerprint, keysize, key creation date, and userid against the information obtained from the key's owner. If everything matches, then the key has been verified and it is OK to sign it.

Revoking Your Key

When you know that your key has been compromised, you should revoke it. A key has been compromised when an attacker has the opportunity to access the full key. This can happen when you are careless with the secret key ring and pass phrase, or if the attacker has spent enough computer time to derive the secret key from the public key.

Warning You should never type a pass phrase in clear-text over the network. Pass phrases should always be typed at a keyboard that is directly connected to the CPU running PGP. Unfortunately, a pass phrase might be typed in the wrong window, at the wrong time, or even in the wrong program.

No matter what the cause of a compromised key, a key compromise, or revocation, certificate should be issued and sent to everyone who might be using the key. A revocation certificate behaves like a signature on the user's own key, which tells PGP not to use the key for any security methods. A revoked key will remain on the key ring, and it can be viewed, extracted, and e-mailed just like a normal key.

```
~> pgp -kd president
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/21 23:29 GMT

Key for user ID: William Clinton <President@Whitehouse.GOV>
512-bit key, Key ID 97D45291, created 1995/11/14

Do you want to permanently revoke your public key
by issuing a secret key compromise certificate
for "William Clinton <President@Whitehouse.GOV>" (y/N)? yes
```

When you ask PGP to revoke a key, it first asks you to verify your decision. You should revoke a key only when you think the key has been compromised or when you never want that key to be used again.

When you verify this revocation, PGP asks for the pass phrase on the secret key. You need the secret key to create a revocation certificate, which means that the pass phrase on the key is required.

```
You need a pass phrase to unlock your RSA secret key.
Key for user ID "William Clinton <President@Whitehouse.GOV>"

Enter pass phrase:
Pass phrase is good.  Just a moment....
Key compromise certificate created.
```

Finally, the compromise certificate is created and added to the secret key ring. You can later extract the key and send it to others to propagate the revocation certificate. Only when other users obtain the revocation certificate will they actually know not to use the key.

Basic Message Operations

PGP can perform a number of security operations on files and messages. The most interesting operations are message encryption and digital signatures, which are listed in table 11.2.

Table 11.2
Message Encryption and Digital Signatures

Operation Parameters	Message Operations
pgp -c text file	Encrypts with conventional encryption only
pgp -s text file [-u your_userid]	Signs a plaintext file with your secret key (produces text file.pgp)
pgp -e text file her_userid [other userids]	Encrypts a plaintext file with recipient's public key (produces text file.pgp)
pgp -es text file her_userid [other userids] [-u your_userid]	Signs a plaintext file with your secret key, and then encrypts it with recipient's public key, producing a .pgp file
pgp ciphertext file [plaintext file]	Decrypts or checks a signature for a ciphertext (.pgp) file

PGP: Program or Filter?

PGP is a program that takes input files, performs a set of operations, and writes an output file. Although this process resembles the functions of a program, PGP can also be thought of as a filter—you give it some input, it processes it and gives you some output. By looking at PGP this way you can see how easily it can be integrated into other programs.

Because PGP 2.6.2 is only distributed as an application program, not as a library, this chapter describes only the application user interface. Some applications that use PGP as a filter are mentioned at the end of this chapter, but most of the effort is spent in explaining the PGP user interface.

To use PGP in filter-mode, PGP should be run with the -f option. This tells PGP to use standard input and standard output for its main functional I/O. The use of filter-mode can change the arguments to various PGP functions because input and/or output files are no longer required. Command examples in this chapter try to explain what happens when PGP is used as a filter.

Compressing the Message

Whenever possible, PGP attempts to compress a message before sending it. This reduces the size of most messages sent by PGP. Of course, PGP compresses messages inside encryption, although it compresses outside a signature, thereby nesting the various operations on a message in the best possible order.

In other words, a PGP signed message first is signed, and then compressed; a PGP encrypted message first is compressed and then encrypted. When PGP combines signatures and encryption, compression happens between the two operations, after the signature is created but before the encryption takes place.

Compression is turned on by default and can be turned off using the COMPRESS option in the configuration file or by using the command-line option:

```
+compress=off
```

Processing Text and Binary Files

Files PGP creates are inherently in binary format, although PGP can process both binary and text files. Binary files are easy to work with because PGP can process the file byte-by-byte. When a text file must be processed, PGP needs to process the file with some special operations for it to transfer properly.

PGP has a canonical format for text files using a special character set and line ending convention. When processing a text file, PGP automatically converts messages from the local character set to ISO Latin-1, an international standard character set. It also uses a carriage return and newline at the end of each line. These text transformations are done before other processing can proceed.

When the PGP file is decrypted and verified, PGP converts the canonical message back into the local character set and local line ending convention. This way a message will never lose its characteristics across various platforms and interoperability can be achieved.

PGP requires you to specify when a text file is the desired file to process and which text-filtering options should be performed. PGP attempts to verify that a file is actually a text file and not a binary file by reading a few bytes of data and testing it. Therefore, it is safe to turn on textmode for non-text files.

To turn on textmode, you add the -t option to PGP. This option specifies that PGP should attempt to process the input file as a text file. If the input is binary, PGP will treat it as binary without the textmode filters.

The TEXTMODE configuration option can also be turned on in the configuration file so that PGP always attempts to use textmode when possible. When this setup is used, you can turn off textmode on the command line:

```
+textmode=off
```

Sending PGP Messages via E-Mail

The files that PGP produces are generally binary files because the PGP protocol is inherently a binary protocol. However, PGP provides a mechanism to encode its binary output in ASCII armor, to protect it from transmission over links that require ASCII data, such as e-mail and netnews. This armor protects a PGP file during transport so that it will not be modified in transit.

Whenever PGP is asked to output PGP data, be it a message or a key, and the -a option is used, PGP will encode the output in ASCII armor. Usually you should use the -a option when creating messages for transmission to other users. Whenever you use ASCII armor, you should remember to use a MIME Transfer-Encoding of 7 bits.

Armor mode can be turned on by default using the ARMOR option in the configuration file. When this is done, Armor mode will always be used. To get binary output, you can turn off Armor mode on the command line:

```
+armor=off
```

You can also control the number of lines of armor that will be put in a single file. Because armorlines are 64 characters wide, you can effectively control the size of the output files. This is useful because some mailer software refuses to allow large messages through; large data need to be broken into multiple files to be sent successfully.

The number of lines of ASCII armor is controlled by the ARMORLINES configuration option. By default, armorlines is set to 720 lines per file. Users can set the number of armorlines to any non-negative integer value. A value of zero (0) will force PGP to output into a single armor file no matter how large the data size. Sometimes it is useful to set the ARMORLINES value in the configuration file to a useful size (if 720 lines does not suffice) and specify zero lines on the command line when a single output file is required:

```
+armorlines=0
```

Conventional Encryption

Sometimes you need to encrypt a message in a pass phrase using conventional encryption. This approach does not provide any key management because PGP converts the pass phrase into an IDEA key and uses that key to encrypt the message. IDEA is a secret key cipher that uses a 128-bit key and encrypts in 8-byte blocks. In general, this mode of operation is not used because it requires manual, out-of-band key distribution. It is useful, however, as a more secure version of crypt, a Unix encryption tool, at times when you want to encrypt messages to yourself using some chosen pass phrase independent of your private key.

Warning Do not use the same pass phrase that is used on the secret key. The new pass phrase should be chosen especially for this file, and a different pass phrase should be used for each file encrypted using conventional encryption. The following sample command lines show the setup and use of a new pass phrase:

```
~> pgp -c message
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 18:58 GMT

You need a pass phrase to encrypt the file.
Enter pass phrase:
Enter same pass phrase again: Just a moment....
Ciphertext file: message.pgp
```

When you use conventional encryption, PGP asks for a pass phrase twice. The second query is to ensure that the pass phrase has been typed properly by the user. The pass phrase is then used to encrypt the message.

In the sample file used throughout the chapter, the output file, message.pgp, contains the encrypted file. It is in binary format, so it cannot be sent to someone else. To send the file to someone else, it should have been wrapped in ASCII armor, using the -a option:

```
pgp -ca filename
```

Signing a Message

To sign a message, you use your secret key to encrypt a digital hash of the message. The signature is attached to the message, and other users can later verify the signature. Using the Web of Trust, a recipient can be assured that the message originated from the appropriate user by using the trust of validity on a key/userid pair.

In general, a message is signed to protect it from tampering when it is transmitted to someone else. For this reason, signatures are usually created with ASCII armor for protection during transmission. The following commands show the process:

```
~> pgp -sa message
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 19:05 GMT

A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key.
Key for user ID "Ruth Thomas <tara@mail.Free.NET>"

Enter pass phrase: Pass phrase is good.
Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14
Just a moment....
Transport armor file: message.asc
```

The file message.asc now contains the signed message encoded in ASCII armor. This file can then be transmitted to recipients who can verify that the message originated with the appropriate user and that the message was not changed in any way during transmission. The following output is the message that was just signed:

```
-----BEGIN PGP MESSAGE-----
Version: 2.6.2
 owHrZJjKzMpgsIv7j8f/mH8XjhnlMjI2cjD/L7yjyviXo/Hgn+77rjx7ihc/ytFi
0jWKqz6/hYP/HM+SdPXak2asUXcj7Yqr2qrqr0/2f6C84eXknSmqm7ri3hw2iDzv
LiG6L477NYcXr/7VhqeyD6fXdHMdcFSwLLWzEeSrvBC88t1LK9ENPAlL9btORq88
KpHik3B1g7fCr5qHGvHLSzPWHE1iz00tLk5MT2UAgpCMzGIFIEpUKM5Mz0tNUYDK
6SkoeJaAJFLzkvNTgOKZeQqOwc6engqJRbn5RQol+QoFRfklqcklXJklCmlF+bkK
JUWJecW5mcXFmfl5EO3J+aU5KQoZiWWpCkmpqXkKxal5JQqlxZl56chm6SgklZZw
lWQAFSSmJwLtKQG5CV1vokJSZl5iUaVCWmZOqiIXAA==
=UK7f
-----END PGP MESSAGE-----
```

Encrypting a Message Using Public Key

In general, whenever someone mentions that a message is “PGP-encrypted,” he or she means that the message was encrypted using Public Key Encryption. A message of this form is actually encrypted using a secret-key cipher, such as IDEA, using a randomly generated key. PGP takes that key and uses Public Key Encryption to transmit that key to all the intended recipients.

When PGP is told to encrypt using Public Key via the -e option, PGP takes the list of recipients, finds their public keys in the public key ring, generates the random session key, and encrypts the session key in each public key. Finally, PGP encrypts the message in the session key.

When the session key is encrypted, PGP adds random padding. Even if you use the same public key twice, the data that is sent will differ. If random padding did not occur, a message encrypted to multiple people would be vulnerable to a mathematical derivation of the session key used in the message. To eliminate this risk, PGP never creates the same output twice when encrypting a message. Not only does it defeat the math attack against the session key, but users also have plausible deniability about even encrypting a message because they cannot re-create the same ciphertext from the same plaintext. They can plausibly deny the fact that they created the message because they cannot create the exact same ciphertext output more than once.

```
~> pgp -ea message warlord tara jis
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 19:41 GMT


Recipients' public key(s) will be used to encrypt.
Key for user ID: Derek Atkins <warlord@MIT.EDU>
709-bit key, Key ID C1B06AF1, created 1992/09/25

Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14

Key for user ID: Jeffrey I. Schiller <jis@mit.edu>
1024-bit key, Key ID 0DBF906D, created 1994/08/27
.
Transport armor file: message.asc
```

When you encrypt messages, it is important to know who the recipient will be. PGP tries to use the key you specify, but it works only if you can specify a unique key. If the name requested matches multiple keys on the key ring, only the first matching key will be used. PGP does not prompt you to choose, nor does it even mention that there was an ambiguity. It is up to you to read the PGP output and recognize when the wrong key is being used. This minor problem should be fixed in a future release.

Signing and Encrypting Messages

Various PGP options can be combined to perform multiple operations on a single message. The signing and encryption of a message can easily be performed in a single step by combining options on the command line. When options are combined on the command line, a hierarchy is used to determine which option is executed first. In this example, PGP first signs the message, and then encrypts the signed message.

```
~> pgp -sea message prz
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 19:45 GMT
```

```
A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key.
Key for user ID "Ruth Thomas <tara@mail.Free.NET>"

Enter pass phrase:
```

Because a signature is involved, PGP asks for a pass phrase, which must be the pass phrase of the secret key. PGP then uses this pass phrase to unlock the secret key and generate the signature on the message.

```
Pass phrase is good.
Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14
Just a moment....

Recipients' public key(s) will be used to encrypt.
Key for user ID: Philip R. Zimmermann <prz@acm.org>
1024-bit key, Key ID C7A966DD, created 1993/05/21

WARNING:  Because this public key is not certified with a trusted
signature, it is not known with high confidence that this public key
actually belongs to: "Philip R. Zimmermann <prz@acm.org>".

Are you sure you want to use this public key (y/N)? yes
.
Transport armor file: message.asc
```

Finally, PGP notifies you which keys are being used to encrypt the message and places the output into the appropriate file. This file can subsequently be transferred to someone else who must first decrypt the message before verifying the signature and reading its contents.

Decrypting and Verifying Messages

When you receive a PGP message, you usually want to use PGP to unpackage it and retrieve the data. This might involve decrypting the message, or verifying the signature on the message. This is the default operation with PGP. It will try to decode the PGP message and decrypt and/or verify the message as necessary and capable.

```
~>pgp message.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 19:52 GMT

File is encrypted.  Secret key is required to read it.
Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14
```

```
You need a pass phrase to unlock your RSA secret key.
Enter pass phrase:
```

PGP first attempts to decrypt the example message because it is encrypted. The message was encrypted for Ruth; she can enter her secret key pass phrase to decrypt the message. The pass phrase opens the secret key, and the secret key opens the message. With a successful pass phrase, PGP can continue processing the message.

```
Pass phrase is good.  Just a moment......
File has signature.  Public key is required to check signature. .
Good signature from user "Derek Atkins <warlord@MIT.EDU>".
Signature made 1995/11/27 19:52 GMT

Plaintext filename: message
```

The message was signed, so PGP attempts to verify the signature with the private key, assuming it is on the public key ring. In this case, the message was signed by Derek Atkins, and the message was not modified during transport. PGP will report the validity of the signature as best it can.

Finally, PGP deposits the decrypted, validated message into the output file. In this case, the file message contains the original message that was encrypted and signed. You can then read, process, or use the file.

Sometimes it is not possible to read a message. The message may have been encrypted using a key or set of keys you don't have. In this case, PGP tries to tell you who can decrypt the message.

```
~> pgp -m message.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/29 19:01 GMT

File is encrypted.  Secret key is required to read it.
This message can only be read by:
  Philip R. Zimmermann <prz@acm.org>
  Jeffrey I. Schiller <jis@mit.edu>
  Derek Atkins <warlord@MIT.EDU>

You do not have the secret key needed to decrypt this file.

For a usage summary, type:  pgp -h
For more detailed help, consult the PGP User's Guide.
```

Another situation in which you might not be able to decrypt a message is when the message is signed by a key that is not on the key ring. In this case, PGP asks for an alternate key ring, and if one is not supplied PGP will not verify the signature. It still attempts to output the message, if possible.

```
~> pgp -m message.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/29 19:10 GMT
.
File has signature.  Public key is required to check signature.
Key matching expected Key ID 82FF3459 not found in file '/tmp/pubring.pgp'.
Enter public key filename: <Enter>

WARNING: Can't find the right public key--can't check signature integrity.


Plaintext message follows...
------------------------------

This is a signed message which is signed by an unknown key

Done...hit any key
Save this file permanently (y/N)? <Enter>
```

PGP tries to report any error conditions, although it is not perfect. It probably will inform you of an invalid signature, a signature by an untrusted key, or reveal other types of problems. To understand the cause of encryption and decryption problems, you need to be aware of the types of messages PGP supplies.

Advanced Message Operations

Some functions of PGP are slightly more advanced and intricate. Although the concepts mentioned in this section might be simple, their application and implication are much more difficult to grasp. The most important thing to remember is that whenever PGP operates on a file, the output is a PGP file. The following table lists most of the useful advanced commands.

Command Parameters	Description
pgp -sat text file	Clearsigns a text message
pgp -sb text file	Creates a separate signature for a file
pgp -m	For Her Eyes Only mode
pgp -w filename	Wipes file clean

Clearsigning

Clearsigning a message is the addition of a digital signature to a message that has been left in text form so that it can be read without the need for PGP. In the future, PGP should be combined with Multimedia Internet Mail Enhancements (MIME) to sign messages, but at this time PGP has its own method. Clearsigning can only sign text files. If a binary file is chosen, PGP will revert to a normal signature on the file instead of clearsigning it.

When PGP clearsigns a message, the output is a PGP file that is partially protected in ASCII armor. Clearsigning does not armor the message itself, only the signature on the message. The message must be capable of being transported without armor protection. Although PGP does not wrap the clearsigned message in armor, it may quote parts of the message. In particular, PGP will quote lines that have a leading dash, or start with the string "From ." When PGP quotes a line, it adds a leading dash (-) followed by a space.

Note that the output of clearsigning a message is a PGP file, not a text file. Even though the output is readable using a text editor or mail reader, the actual text may be modified by the signer (that is, quoted), so anything that depends on the text itself should be used only on the output from PGP. For example, a clearsigned PostScript file may not execute on the remote side due to the clearsigned quoting until PGP is used to retrieve the original text.

It is important that you understand the distinction between a PGP file and a text file. Though a clearsigned message is readable, it is not necessarily the original message sent. You should always run PGP on clearsigned messages and use the output from PGP as the original message; never use the contents of a clearsigned message and run PGP just to verify the signature. Instead, you should always use PGP to unquote the clearsigned message before running the text file through any other processor.

```
~> pgp -sat message
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/19 05:17 GMT

A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key.
Key for user ID "Ruth Thomas <tara@mail.Free.NET>"

Enter pass phrase:
```

Because a signature is requested, PGP asks for the pass phrase of the secret key. By default, PGP uses the first secret key on the secret key ring, which is the most recently created key. As an alternative, you can also specify the secret key used to sign the message by using the -u command-line option or by specifying the MYNAME variable in the configuration file.

```
Pass phrase is good.
Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14
Just a moment....
Clear signature file: message.asc
```

Detached Signatures

A *detached signature* is a signature that is stored separately from the file it is meant to protect—the original file is unmodified. This scenario is usually used to sign files in-place, such as package distributions and system programs. Any time you want to sign a message but not require the recipient to use PGP on the original file, you probably want to use detached signatures.

A detached signature has the same information as a normal signature: who signed the file, when it was signed, and signature data. The difference is that the signature file and signed file must be transmitted separately. If the signed file is an executable program, this may be the most useful way to verify the program. For example, you could sign the PGP binary using a separate signature so that someone can later verify the signature on the binary. To generate a separate signature, use the -sb option to PGP:

```
~> pgp -sba text file
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 20:09 GMT

A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key.
Key for user ID "Ruth Thomas <tara@mail.Free.NET>

Enter pass phrase:
```

PGP will ask for the pass phrase of the current secret key. This pass phrase will open the key so that it can be used to generate the signature. When the key is successfully opened, PGP will put the signature in a separate file, leaving the original file intact.

```
Pass phrase is good.
Key for user ID: Ruth Thomas <tara@mail.Free.NET>
1024-bit key, Key ID D0C6326D, created 1995/11/14
Just a moment....
Transport armor file: text file.asc
```

The output message is just the signature on the original file. The original file can be a text file, a binary file, an executable, or anything else. A signature looks like this:

```
-----BEGIN PGP MESSAGE-----
Version: 2.6.2
```

```
iQCVAwUAMLoa4kj/XP7QxjJtAQEzMgP/ZlQRGio1xYPxJnTaflxhmX5s5b66WN6Z
PMZo3LcO/K6HwFuunL0u0qt6rwKOHd5gm83GEv6Xic8MwraYT347hY86QWYFbw7A
aEAXQPY1PNK8YD6ZPm38ChXXjAzqEEYHYO10KBA5FGKuEpv1GhpYAuau0FwftZVN
r/e1rB6/2A8=
=15Fb
-----END PGP MESSAGE-----
```

For Her Eyes Only

PGP contains an internal pager that can be used to view the program's output. When you decrypt a message, you can use this option to send the output to the screen, rather than save it to a file. When PGP finishes showing the plaintext, it asks you whether you want to save the message to a file. In this manner, you can decrypt and read a message without saving it off to a file. The -m option tells PGP to use the pager to view the output.

Sometimes the message that is being sent is so sensitive that the sender believes it should only be displayed on-screen and not saved to a file. In other words, the message is meant to be read only. When encrypting a message, you can set a flag in the message to tell PGP to print out the message on the recipient's screen without allowing him or her to save it to a file. To mark a message For Her Eyes Only, the -m option should be given to PGP when the message is encrypted. On decryption, PGP will only use the pager and will not enable the user to save the output to a file.

Note Although PGP tries to prevent recipients from saving messages encoded For Her Eyes Only, it cannot prohibit it. The reader can work around this limitation by using screen dump programs or other text collection means that vary from system to system. For Her Eyes Only is meant as a hint for the recipient and should be used as a means to keep a user from accidentally saving a message to a file.

Wiping Files

PGP can also wipe files clean. In the file systems of most machines, a directory contains a list of pointers to files. When a file is removed, the pointer to the data on disk is removed from the list of files and the space that the file occupies is marked as unused. The actual file, however, still sits on the disk and remains there until another file writes over the same spot on the disk.

Sometimes data encrypted with PGP is so important that you might not want it to remain on the disk in clear text. Fortunately, PGP lets you wipe the file off the disk. When the -w option is used, PGP wipes the source file before removing it from the directory list. The result is data on the disk appears as pseudo-random numbers before it is deleted, thwarting would-be crackers who might be looking for the original file on the disk.

When pgp -w is used alone, this option will wipe and remove a file. When used in conjunction with other options, -w will wipe and remove the original file:

```
~> ls -l
total 1
-rw-rw-r--   1 warlord  users         26 Nov 27 13:35 origin
~> pgp -w origin
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1995/11/27 21:35 GMT

File origin wiped and deleted.
~> ls -l
total 0
```

The PGP Configuration File

When you want to configure PGP, you can use a file to specify options other than the defaults for various values that PGP uses. Each user is allowed to have a configuration file that PGP will read on startup to define how it behaves for that user. The configuration file specifies items such as the default number of lines of armor or the default key to use.

The default configuration file is called config.txt and is located in the directory in the PGPPATH environment variable. On Unix systems, the default PGPPATH is the .pgp directory in the user's home directory, $HOME/.pgp. Various OS systems have various options for the configuration filename. In Unix, for example, you can use the file .pgprc in the PGPPATH directory. When using DOS, you can use the file named pgp.ini.

PGP also supports a system-wide configuration file, which can be used to set up defaults for all users of a system. The user's local configuration file will override the options set in the system configuration file. The system configuration file location is set at compile-time. In Unix, the default location is /usr/local/lib/pgp; in VMS, the default is PGP$LIBRARY.

Three types of values are required by configuration variables: Boolean, integer, and string. A *Boolean* is a yes/no value, and is denoted either by "true" and "false," or "on" and "off." An *integer* value is a number; some numbers must be non-negative. A *string* is a series of characters up to the next newline.

Table 11.3 contains configuration keywords that PGP supports. These keywords can be put into the configuration file, which is normally the file config.txt in the PGPPATH directory. PGP also accepts these configuration values on the command line by preceding the configuration option with a plus (+) and following it with an equal sign (=) and its value. This is described in more detail later in this chapter.

Table 11.3
Configuration Keywords for PGP Startup

Name	Type	Default	Effect
ARMOR	Boolean	off	When this option is on, data is output encoded in ASCII armor.
ARMORLINES	integer	720	The number of lines to put in a single ASCII armor block. If there are more than this number of lines, PGP will break up the message into multiple output files.
BAKRING	string		The directory in which PGP should store backup key rings. In general, this is used to keep a backup key ring on a floppy disk. PGP will then compare the data on the normal key ring with the data in the backup key ring and report errors when they do not match.
CERT_DEPTH	integer	4	The maximum depth for which certification is valid in the web of trust. This is the maximum level of recursion that PGP will allow.
CHARSET	string	noconv	The character set to use when displaying messages locally. PGP internally uses the Latin-1 charset and converts to external character sets as appropriate. By default, no conversion is done except for MS-DOS, which uses the default charset cp850, not noconv.
CLEARSIG	Boolean	on	When possible, clearsign text messages. If this is off, never clearsign messages. Clearsigning is only possible on text messages when signing with ASCII armor.
COMMENT	string		When defined, this string will be put in the headers of ASCII armor.
COMPLETES_NEEDED	integer	1	The number of completely trusted key certifications needed to trust the validity of a public key.

Name	Type	Default	Effect
COMPRESS	Boolean	on	When turned on, try to compress all messages when possible. Clearsigned and separate-signature messages are not compressed, but any normal operation will be compressed.
ENCRYPTTOSELF	Boolean	off	Automatically add the originator to the list of recipients when using public key encryption.
INTERACTIVE	Boolean	off	Interactively add keys to the system. By default PGP will add keys in a lump to the key ring. This option allows users to interactively decide which keys to add and which not to add.
KEEPBINARY	Boolean	off	Keep a binary version of the file around. When decrypting an ASCII armor file, PGP will save the binary contents of the ASCII armor to a file.
LANGUAGE	string	en	What language to use when printing messages to the user. By default the program uses English.
MARGINALS_NEEDED	integer	2	The number of marginally trusted key certifications needed to trust the validity of a key.
MYNAME	string		The name of the key to use when signing messages. By default, PGP will use the first key on the secret key ring, which is usually the most recently generated key.
PAGER	string		The pager program to use when printing messages in For Her Eyes Only mode. This option will override the environment variable, PAGER, which in turn overrides the default pager. The default pager is the internal pager except under VMS, which uses Type/Page. Set the PAGER configuration variable to

continues

Table 11.3, Continued
Configuration Keywords for PGP Startup

Name	Type	Default	Effect
			"pgp" to override the environment variable and use the internal pager.
PUBRING	string		Specifies the location of the public key ring. By default, PGP will look in the PGPPATH directory for the file pubring.pgp. This variable will override the file $PGPPATH/ pubring.pgp; PGP will use this file instead.
RANDSEED	string		Specifies the location of the random number seed file. By default, PGP will look in the PGPPATH directory for randseed.bin. As with PUBRING, PGP will use this file instead of looking in PGPPATH.
SECRING	string		Specifies the location of the secret key ring file. By default, this option looks in the PGPPATH directory for the file secring.pgp. PGP will use this file instead of looking in PGPPATH.
SHOWPASS	Boolean	off	When on, show the pass phrase as it is being typed. By default, this option is off to protect your pass phrase from being read while you type it.
TEXTMODE	Boolean	off	When turned on, assume a file is a text file. PGP will always check to verify if it is a text file, and will turn off textmode if it is not.
TMP	string		The directory where temporary files are created. PGP will try to choose a reasonable default if it is not set in the configuration file. On Unix systems, PGP uses the contents of the TMP environment variable; on

Name	Type	Default	Effect
			VMS, PGP will use the contents of SYS$SCRATCH; on DOS, the current directory is used.
TZFIX	integer	0	The number of hours to add to the time to get GMT. This is needed only if the TZ environment variable does not work.
VERBOSE	integer	1	The verbosity level of PGP. The more verbose, the more debugging information and progress information is printed to the user. Verbose level 0 is quiet mode, and verbose level 2 provides extra runtime information.

PGP also supports a number of configuration options that only make sense on the command line. Table 11.4 lists these options. As you saw in table 11.3, these options are also used by putting a plus sign before the name, and following it with an equal sign and the value. For example, to turn off compression you can add `+compress=off` to the command line.

Table 11.4
Configuration Options for PGP

Name	Type	Default	Effect
BATCHMODE	Boolean	off	Process the current request as a batch request. This is useful for servers and to perform default operations without asking for user input.
FORCE	Boolean	off	When turned on, force PGP to answer questions using default values. This option forces PGP to perform the default actions instead of asking the user. In general, this is used with BATCHMODE for system servers that want to use PGP.
MAKERANDOM	integer		Output a file of random bytes, using the length of this variable.

If you want to use PGP as a random number generator, for instance, it can be configured to make a file of random numbers. You can specify this using the makerandom option. For example, to generate 1k of random data into a file named output.bin, you would use this command:

```
pgp +makerandom=1024 output.bin\\
```

The configuration options are best used by setting the preferred default options in the configuration file and then using the command-line options to change the defaults when necessary. For example, a suggested mode is to specify TEXTMODE and ARMOR to be true in the configuration file, and use +armor=off or +textmode=off on the command line when textmode or armor mode or both are not desired.

Security of PGP

The use of a security program does not ensure that your communications will be secure. You can have the most secure lock on the front door of your house, and a prowler can still crawl in through an open window. Similarly, your computer can be just as vulnerable, even when using PGP.

A number of known attacks exist against PGP; the next few sections cover many of them. However, this is by no means a complete list. Attacks may be found in the future that break all public key cryptography. This list tries to give you a taste of what you need to protect your communications.

The Brute Force Attack

The most direct attack against PGP is to brute force the keys that are used. Because PGP 2.6.2 uses two cryptographic algorithms, it is appropriate to look at the security of both algorithms. For public key cryptography, PGP uses the RSA algorithm; for secret key cryptography, it uses IDEA.

Brute Force on RSA Keys

For RSA keys, the best brute force attack known is to try to factor them. RSA keys are generated so that they are difficult to factor. Moreover, factoring large numbers is still a new art.

The most recent, and largest, RSA key to be factored is RSA-129 in April, 1994. RSA-129 is the original RSA challenge number that was created in 1977 when the RSA algorithm was devised. It is a 129-decimal digit RSA key, which is equivalent to about 425 bits. A worldwide effort to factor the number used the resources of 1,600 computers for over eight months of real time. This figures out to 4,600 MIPS-years; a MIPS-year is the amount of data a 1 MIPS machine could process in one year.

For example, a Pentium 100 is approximately 125 MIPS (according to Intel). If one Pentium 100 machine were to run full time for one full year on a problem, it would donate 125 MIPS-years. At this rate, it would take one machine just about 37 years to break RSA-129. Alternatively, 100 machines could break the code in just over 4 months, which is about half the time of the actual project.

A newer factoring algorithm exists than the one used in the RSA-129 project. This newer algorithm is much faster, and is believed to be able to factor RSA-129 in about a quarter of the time. It is uncertain how this new algorithm will perform, and there is currently a project underway to factor RSA-130, a sister challenge to RSA-129. As of this writing, many computers around the world are working on factoring this number. The results may not be known for some time.

Currently, PGP uses keys between 512 and 2,048 bits. The larger the key the harder it is to factor. At the same time, increasing the keysize increases the time it takes to use that key. To date, a 512-bit key is believed to give about one year of security; access to 100 Pentium 100 machines should take at least a year to crack a 512-bit RSA key. If that is true, then a 1,024-bit key, given today's newest algorithms, will be secure for the next 10,000 years, assuming no more increases in technology. If technology increases, less time will be required.

Brute Forcing IDEA Keys

There are no known attacks against IDEA keys at this time. The best that can be done is trying all 2^{128}, or 3.4×10^{38}, keys. Given the difficulty in performing this test, it is actually easier to try to break the RSA keys that are used to encrypt the IDEA keys in PGP. It has been estimated that the difficulty in breaking IDEA is about the same difficulty as factoring a 3,000-bit RSA key.

Secret Keys and Pass Phrases

The security of the PGP secret key ring is based on two things: access to the secret key ring data and knowledge of the pass phrase that is used to encrypt each secret key. Possession of both parts is needed to use the secret key. This also leads to a number of attacks, however.

If PGP is used on a multiuser system, access to the secret key ring is possible. Through cache files, network sniffing, or a multitude of other attacks, a secret key ring can be obtained just by watching the network or reading through the disks. This leaves only the pass phrase to protect the data in the secret key ring, which means an attacker needs to obtain only the pass phrase to break the security of PGP.

Moreover, on a multiuser system, the link between the keyboard and the CPU is probably insecure. Watching the keystrokes would be easy for anyone who has physical access to the network connecting the user's keyboard to the mainframe being used. For example, users might be logged in from a public cluster of client terminals, where the connecting network can be sniffed for pass phrases. Alternatively, users might be dialing up via modem, in which case an eavesdropper could listen in on their keystrokes. In either case, running PGP on a multiuser machine is insecure.

Of course, the most secure way to run PGP is on a personal machine that no one else uses and is not connected to the network; in other words, a laptop or home computer. Users must balance the cost of a secure environment with that of secure communications. The recommended way to use PGP is always on a secure machine in a secure environment, where the user has control over the machine.

The key to the best type of security is that the connection between the keyboard and the CPU be secure. This is accomplished either by encryption or better yet by a direct, uninterruptible connection. Workstations, PCs, Macs, laptops—all fit into the category of secure machines. The secure environment is much more difficult to show and is not explored here.

Public Key Ring Attacks

Because of the importance and dependence on the public key ring, PGP is susceptible to a number of attacks against the key ring. First, the key ring is checked only when it changes. When new keys or signatures are added, PGP will attempt to verify them. However it will flag the checked signatures on the key ring so it will not validate them again. If someone modifies the key ring and sets the bits appropriately on signatures, they will not be checked.

Another attack against the key ring focuses on the process PGP uses to set a bit for the validity trust in a key. When new signatures arrive on a key, PGP computes the validity of the key by using the Web of Trust values described earlier. PGP then caches the validity on the public key ring. An attacker could modify this bit on the key ring to force a user to trust the validity in an invalid key. For example, by setting this flag an attacker could make the user believe that a key belongs to Alice even though there are not enough signatures to prove that validity.

Another attack against PGP's public key ring may occur because the trust of a key as an introducer is also cached on the public key ring. This value defines how much trust is put in this key's signatures, so it is possible to force PGP to accept invalid keys as valid by signing them with the key with the invalid trust parameter. If a key were modified to be a fully trusted introducer, any keys that were signed by that key would be trusted as valid. Therefore, an attacker could force the user to believe that a forged key is valid by signing it with the modified key.

The biggest problem with the public key ring is that all of these bits are not only cached on the key ring, but they are not protected in any way on the key ring! Anyone who has read the PGP source code and has access to the public key ring can use a binary file editor to change any of these bits, and the key ring owner would never notice the change. Fortunately, PGP provides a way to recheck the keys on the key ring. By using the -kc and -km options together, a user can tell PGP to perform a key maintenance pass over the whole key ring. The former option tells PGP to check keys and signatures. It will go through the key ring and recheck every signature.

When all the signatures have been checked, PGP will perform a maintenance check (-km) and recompute the validity of all the keys.

Unfortunately there is no way to completely recheck all of the trust bytes on keys. This is a bug. There should be a command to tell PGP to ignore all trust bytes and ask the user for trust starting with the ultimate keys—those on the secret key ring. Perhaps a future version of PGP will fix this problem. If a key is modified to be a trusted introducer, there is no easy way for you to find the change and fix it. Running the key and maintenance checks will revert the validity of a key, but not the trust value. Only running pgp -ke on a key will enable you to edit the trust parameters, and this cannot be done automatically.

Program Security

If someone has access to the PGP binary, he or she can change it and do whatever they want it to do. If this meddler can replace your PGP binary from right underneath your nose, your trust in PGP would then be based on your trust in that person or your ability to actually verify the program. For example, an attacker with such access could change PGP to always validate signatures, even if the signature is invalid. PGP could be modified to always send a cleartext copy of all messages straight to the NSA. These kinds of attacks are difficult to detect and difficult to counteract. PGP needs to be a part of the trusted code base; if you cannot trust your PGP binary, then you cannot trust its output.

The best way to trust the PGP binary is to build it from sources yourself. That is not always possible, however. Alternatives involve watching it being built or getting it from a trusted source. It helps to look at the size and date of the binary. Using other trusted programs like md5sum can help. But this just pushes the problems down to another layer. If you cannot trust the PGP program, there is not much you can do.

Other Attacks Against PGP

Other attacks are possible against PGP, but they are not discussed here. It has never been proven that the cryptographic algorithms used in PGP are secure. It is possible that the mathematics used in PGP, which are believed to be secure, may be simple to break. Factoring attacks against RSA could improve, or someone could find a hold in IDEA.

Not enough is known about the mathematics behind cryptography to know what is and is not secure. In fact, it is known that nothing can be completely secure. Given enough computer power it is possible to break any form of cryptography. The question is if the cost of the time and effort to break the code is worth the cost of the data that is being protected. Note that the cost of the effort to break a code will only decrease as time moves on because the computer power keeps increasing and costs continue to decrease. For now, the cryptographer is still ahead of the cryptanalyist.

PGP Add-Ons

PGP is an extremely useful program, but unfortunately it still is very difficult to use. It provides so much functionality that it has become cumbersome and confusing to new users. The current release of PGP is definitely not something that this author's mother could use. However, there are a number of add-ons that can help.

Many people have written front-end applications or programs that provide additional features to make PGP easier to use, easier to integrate, or provide PGP with some useful additional functions. This chapter cannot include an exhaustive list, but does mention many of the most recent and most useful add-ons.

PGP Public Keyservers

One problem with PGP is that it is difficult to find the public key for a person without first contacting him or her. If people who use PGP aren't signing public postings, such as on Usenet, you need to be able to obtain public keys without interacting with everyone involved. The Public Keyservers serve this purpose.

The Public Keyservers are a network of machines that contain a list of all the published PGP public keys. You can publish your public key by sending it to any one of the keyservers. Because all the keyservers talk to each other, new keys and key updates are propagated to all the keyservers. When you want to obtain a key, you can access a keyserver and be sure the published key that matches the query will be there.

To update a key, the only thing you need to do is extract and send in the new key. The keyservers will merge the existing and new keys together. New signatures will be added to the existing key, and new userids will be prepended. Key revocations are treated the same way. Just send in the key with the revocation certificate and it will be propagated to all the keyservers, thereby revoking the key.

Warning Keep in mind that keyservers are not trusted machines. You should never trust a key just because it came from a keyserver. Trust should be based solely on the signatures on the key, not on the basis of the keyserver.

Keyservers support only a few commands: Add, Get, MGet, Index, Verbose Index, and Help. All keyserver commands are sent in the subject of an e-mail message; the message body is ignored for all commands except Add. For the Add command, you must send your public keys (extracted using pgp -kxa, sent as plain text) as the message body. You can use the Get command to obtain a key from the keyserver by supplying an argument: "get userid". Mget lets you request a number of keys using a regular expression. Index and Verbose Index let you search for keys that are available.

The easiest way to learn more about the public keyservers is to ask them for assistance. You can send a message to the keyserver network using the address <pgp-public-keys@keys.pgp.net>. Send an e-mail with a subject of "help" to obtain a full help message in response.

PGPMenu: A Menu Interface to PGP for Unix

Because PGP can be so difficult to use for beginner users, PGPMenu was written to help people use PGP and to minimize the steep learning curve. PGPMenu is a menu-based interface for PGP's message handling, key management, and configuration options.

The program was implemented for a Unix-based system. It is written in PERL, and is the only TTY-based interface—not a graphical interface. It might not be pretty, but PGPMenu provides an easy way for novices to start using PGP.

When the program starts, it reads in your PGP Configuration file and presents a menu of options. The main menu enables you to use the PGP message security operations. Most PGP operations are supported on this menu. You can also call up the key management or configuration menus.

The key management menu enables you to maintain key rings. You can add, sign, extract keys, send keys to the keyserver network, and even get keys signed via MITSign if it is available. Of course, PGPMenu can help you generate keys, and will even help select an appropriate username. The interface allows you to access the PGP functions without requiring you to remember the nuances of the PGP command-line interface.

The PGPMenu configuration menu also lets you control some of the values that can be stored in the config.txt file. You can change a number of configuration options and even save them to the config.txt file for later use.

More information about PGPMenu can be found on the World Wide Web via the following URL:

```
http://www.mit.edu:8001/people/warlord/pgpmenu.html
```

MITSign: A Kerberized PGP Key Signer

One major problem with PGP is that the Web of Trust does not easily scale. One feature of Privacy Enhanced Mail (PEM, one of the secure mail standards) is that it has a certification hierarchy, where certification authorities (CA) sign keys to validate them. When a key is signed by a CA, other users can verify the key by following a certification path down to the CA and then to the key in question.

When a site has an existing Kerberos installation, MITSign lets the existing security infrastructure provide a certification authority for PGP. Kerberos is a network authentication system that was developed at MIT's Project Athena. It uses DES to encrypt network authentication tickets, which, in turn, are used to authenticate a client to a server. A trusted server, the Kerberos server, acts as an introducer between all clients and servers on the network.

The keysigner accepts a Kerberos authentication from a user and compares the authentication to the userid on the PGP key. Using a set of rules, the keysigner decides whether to sign the PGP key based on the authentication, Kerberos name, PGP userid, site specifics, and other rules.

In this manner, the creation of a PGP CA is simplified by using the existing Kerberos infrastructure at a site. For example, both MIT and Stanford have keysigners running. If there is a path between the MIT and Stanford keysigner keys, then it becomes possible to validate keys between both sites because MIT users are signed by the MIT key, and Stanford users are signed by the Stanford key. This reduces the number of trusted keys necessary to validate user keys.

Note See Chapter 9 for more information about Kerberos.

More information about MITSign can be found on the World Wide Web via the following URL:

```
http://www.mit.edu:8001/people/warlord/mitsign.html
```

Windows Front-Ends

A number of front-end applications are available for Microsoft Windows that provide various interfaces to PGP. Unfortunately, there is no native Windows PGP application, so the front-end programs are the best interfaces for Windows users.

All windows front-ends are built on top of the DOS PGP executable. They read the text output messages from PGP and interpret them for the user. These messages can then be presented more graphically. This approach is a simple way to use PGP and to interface it with other programs.

So many Windows front-ends to PGP exist that it would take another chapter to describe them. A helpful list of PGP utilities is available on the World Wide Web through the following URL address:

```
http://world.std.com/~franl/pgp/utilities.html
```

Unix Mailers

PGP has been integrated into a number of mailers for various flavors of Unix. There are too many variations to go into all of them here, but suffice it to say that someone has either completed or is working on an integration tool for most major popular mailers.

As of this writing, it is known that interfaces exist and work for emacs mailers and the elm mailer agent(2.4pl24). Scripts that tie into pine and mh are also available. More information about these can also be found at the previous page on the World Wide Web:

```
http://world.std.com/~franl/pgp/utilities.html
```

Mac PGP

For Macintosh users, a native MacPGP program can be used. Unlike the Windows front-end applications, MacPGP is a native PGP application with a Macintosh interface. This program enables you to directly operate on files. The best part about the recent versions of MacPGP is that it can interface to other programs using Apple Events. One such program is the Eudora mailer for Macintosh. Using Apple Events, Eudora can ask PGP to sign, verify, encrypt, or decrypt messages. This way the functions of PGP can be added to other applications.

More information about MacPGP can be found on the World Wide Web via the following URL:

```
http://web.mit.edu/network/pgp.html
```