Phil's Pretty Good Software Presents
PGP Pretty Good Privacy
Public Key Encryption for the Masses

By Philip Zimmermann

PGP Version 2.62 -11Oct 94 Software by Philip Zimmermann, and Many others.
This hypertext version of the manual compiled by Jeff Sheets (Xanthur@aol.com) by permission from Philip Zimmermann, & Revised 30 April 95 and will be considered freeware insomuch as it is distributed freely...If it is contained in any shareware or commercial product then please contact me by Email for authorization--Jeff

Installation Guide
Volume I -- Essential Topics
Volume II --Special Topics
New Features Per Version
File Formats Used by PGP 2.6
PGP Quick Reference

Synopsis: PGP(tm) uses public-key encryption to protect E-mail and data files. Communicate securely with people you've never met, with no secure channels needed for prior exchange of keys. PGP is well featured and fast, with sophisticated key management, digital signatures, data compression, and good ergonomic design.

"Whatever you do will be insignificant, but it is very important that you do it." --Mahatma Gandhi

Volume I -- Essential Topics

Quick Overview
Why Do You Need PGP?
How it Works
Installing PGP
How to Use PGP
Managing Keys
Advanced Topics
Legal Issues
Acknowledgments
About the Author

Volume II - Special Topics

Separating Signatures from Messages
Decrypting the Message and Leaving the Signature on it
Handling of Text
Using PGP as a Better Uuencode
Keyrings and Key Management
Using PGP as a Unix Style Filter
BATCHMODE
Setting Configuration Parameters : CONFIG.TXT
A Peek Under the Hood
Vulnerabilities
Legal Issues
Where to get PGP
Reporting PGP Bugs
Computer Related Political Groups
Recommended Reading
To Contact The Author

Quick summary of PGP v2.3 commands.

Basic Commands
Key Management Commands
Esoteric Commands
Combination Command Options

Basic Commands

To encrypt a plaintext file with the recipient's public key:

```
pgp -e textfile her_userid
```

To sign a plaintext file with your secret key:

```
pgp -s textfile [-u your_userid]
```

To sign a plaintext file with your secret key, and then encrypt it with the recipient's public key:

```
pgp -es textfile her_userid [-u your_userid]
```

To encrypt a plaintext file with just conventional cryptography, type:

```
pgp -c textfile
```

To decrypt an encrypted file, or to check the signature integrity of a signed file:

```
pgp ciphertextfile [-o plaintextfile]
```

To encrypt a message for any number of multiple recipients:

```
pgp -e textfile userid1 userid2 userid3
```

--- Command options that can be used in combination with other command options (sometimes even spelling interesting words!):

To produce a ciphertext file in ASCII radix-64 format, just add the
-a option when encrypting or signing a message or extracting a key:

```
    pgp -sea textfile her_userid
or: pgp -kxa userid keyfile [keyring]
```

To wipe out the plaintext file after producing the ciphertext file,
just add the -w (wipe) option when encrypting or signing a message:

```
    pgp -sew message.txt her_userid
```

To specify that a plaintext file contains ASCII text, not binary, and
should be converted to recipient's local text line conventions, add
the -t (text) option to other options:

```
    pgp -seat message.txt her_userid
```

To view the decrypted plaintext output on your screen (like the
Unix-style "more" command), without writing it to a file, use
the -m (more) option while decrypting:

```
    pgp -m ciphertextfile
```

To specify that the recipient's decrypted plaintext will be shown
ONLY on her screen and cannot be saved to disk, add the -m option:

```
    pgp -steam message.txt her_userid
```

To recover the original plaintext filename while decrypting, add
the -p option:

```
    pgp -p ciphertextfile
```

To use a Unix-style filter mode, reading from standard input and
writing to standard output, add the -f option:

```
    pgp -feast her_userid <inputfile >outputfile
```

Esoteric Commands

To decrypt a message and leave the signature on it intact:

```
pgp -d ciphertextfile
```

To create a signature certificate that is detached from the document:

```
pgp -sb textfile [-u your_userid]
```

To detach a signature certificate from a signed message:

```
pgp -b ciphertextfile
```

Key Management Commands

To generate your own unique public/secret key pair:
    pgp -kg

To add a public or secret key file's contents to your public or secret key ring:
    pgp -ka keyfile [keyring]

To extract (copy) a key from your public or secret key ring:
    pgp -kx userid keyfile [keyring]
or: pgp -kxa userid keyfile [keyring]

To view the contents of your public key ring:
    pgp -kv[v] [userid] [keyring]

To view the "fingerprint" of a public key, to help verify it over the telephone with its owner:
    pgp -kvc [userid] [keyring]

To view the contents and check the certifying signatures of your public key ring:
    pgp -kc [userid] [keyring]

To edit the userid or pass phrase for your secret key:
    pgp -ke userid [keyring]

To edit the trust parameters for a public key:
    pgp -ke userid [keyring]

To remove a key or just a userid from your public key ring:
    pgp -kr userid [keyring]

To sign and certify someone else's public key on your public key ring:
    pgp -ks her_userid [-u your_userid] [keyring]

To remove selected signatures from a userid on a keyring:
    pgp -krs userid [keyring]

To permanently revoke your own key, issuing a key compromise certificate:
    pgp -kd your_userid

To disable or reenable a public key on your own public key ring:
    pgp -kd userid

Quick Overview

Pretty Good(tm) Privacy (PGP), from Phil's Pretty Good Software, is a high security cryptographic software application for MSDOS, Unix, VAX/VMS, and other computers. PGP allows people to exchange files or messages with privacy, authentication, and convenience. Privacy means that only those intended to receive a message can read it. Authentication means that messages that appear to be from a particular person can only have originated from that person. Convenience means that privacy and authentication are provided without the hassles of managing keys associated with conventional cryptographic software. No secure channels are needed to exchange keys between users, which makes PGP much easier to use. This is because PGP is based on a powerful new technology called "public key" cryptography.

PGP combines the convenience of the Rivest-Shamir-Adleman (RSA) public key cryptosystem with the speed of conventional cryptography, message digests for digital signatures, data compression before encryption, good ergonomic design, and sophisticated key management. And PGP performs the public-key functions faster than most other software implementations. PGP is public key cryptography for the masses. PGP does not provide any built-in modem communications capability. You must use a separate software product for that.

This document, "Volume I: Essential Topics", only explains the essential concepts for using PGP, and should be read by all PGP users. "Volume II: Special Topics" covers the advanced features of PGP and other special topics, and may be read by more serious PGP users. Neither volume explains the underlying technology details of cryptographic algorithms and data structures.

Why do you need PGP?

It's personal. It's private. And it's no one's business but yours. You may be planning a political campaign, discussing your taxes, or having an illicit affair. Or you may be doing something that you feel shouldn't be illegal, but is. Whatever it is, you don't want your private electronic mail (E-mail) or confidential documents read by anyone else. There's nothing wrong with asserting your privacy. Privacy is as apple-pie as the Constitution. Perhaps you think your E-mail is legitimate enough that encryption is unwarranted. If you really are a law-abiding citizen with nothing to hide, then why don't you always send your paper mail on postcards? Why not submit to drug testing on demand? Why require a warrant for police searches of your house? Are you trying to hide something? You must be a subversive or a drug dealer if you hide your mail inside envelopes. Or maybe a paranoid nut. Do law-abiding citizens have any need to encrypt their E-mail? What if everyone believed that law-abiding citizens should use postcards for their mail? If some brave soul tried to assert his privacy by using an envelope for his mail, it would draw suspicion. Perhaps the authorities would open his mail to see what he's hiding. Fortunately, we don't live in that kind of world, because everyone protects most of their mail with envelopes. So no one draws suspicion by asserting their privacy with an envelope. There's safety in numbers. Analogously, it would be nice if everyone routinely used encryption for all their E-mail, innocent or not, so that no one drew suspicion by asserting their E-mail privacy with encryption. Think of it as a form of solidarity.

Today, if the Government wants to violate the privacy of ordinary citizens, it has to expend a certain amount of expense and labor to intercept and steam open and read paper mail, and listen to and possibly transcribe spoken telephone conversation. This kind of labor-intensive monitoring is not practical on a large scale. This is only done in important cases when it seems worthwhile. More and more of our private communications are being routed through electronic channels. Electronic mail is gradually replacing conventional paper mail. E-mail messages are just too easy to intercept and scan for interesting keywords. This can be done easily, routinely, automatically, and undetectably on a grand scale. International cablegrams are already scanned this way on a large scale by the NSA.

We are moving toward a future when the nation will be crisscrossed with high capacity fiber optic data networks linking together all our increasingly ubiquitous personal computers. E-mail will be the norm for everyone, not the novelty it is today. The Government will protect our E-mail with Government-designed encryption protocols. Probably most people will acquiesce to that. But perhaps some people will prefer their own protective measures.

Senate Bill 266, a 1991 omnibus anti-crime bill, had an unsettling measure buried in it. If this non-binding resolution had become real law, it would have forced manufacturers of secure communications equipment to insert special "trap doors" in their products, so that the Government can read anyone's encrypted messages. It reads: "It is the sense of Congress that providers of electronic communications services and manufacturers of electronic communications service equipment shall insure that communications systems permit the Government to obtain the plain text contents of voice, data, and other communications when appropriately authorized by law." This measure was defeated after rigorous protest from civil libertarians and industry groups. In 1992, the FBI Digital Telephony wiretap proposal was introduced to Congress. It would require all manufacturers of communications equipment to build in special remote wiretap ports that would enable the FBI to remotely wiretap all forms of electronic communication from FBI offices. Although it never attracted any sponsors in Congress in 1992 because of citizen opposition, it was reintroduced in 1994.

Most alarming of all is the White House's bold new encryption policy initiative, under development at NSA since the start of the Bush administration, and unveiled April 16th,

1993. The centerpiece of this initiative is a Government-built encryption device, called the "Clipper" chip, containing a new classified NSA encryption algorithm. The Government is encouraging private industry to design it into all their secure communication products, like secure phones, secure FAX, etc. AT&T is now putting the Clipper into their secure voice products. The catch: At the time of manufacture, each Clipper chip will be loaded with its own unique key, and the Government gets to keep a copy, placed in escrow. Not to worry, though-- the Government promises that they will use these keys to read your traffic only when duly authorized by law. Of course, to make Clipper completely effective, the next logical step would be to outlaw other forms of cryptography.

If privacy is outlawed, only outlaws will have privacy. Intelligence agencies have access to good cryptographic technology. So do the big arms and drug traffickers. So do defense contractors, oil companies, and other corporate giants. But ordinary people and grassroots political organizations mostly have not had access to affordable "military grade" public-key cryptographic technology. Until now.

PGP empowers people to take their privacy into their own hands. There's a growing social need for it. That's why I wrote it.

How it works

It would help if you were already familiar with the concept of cryptography in general and public key cryptography in particular. Nonetheless, here are a few introductory remarks about public key cryptography.

First, some elementary terminology. Suppose I want to send you a message, but I don't want anyone but you to be able to read it. I can "encrypt", or "encipher" the message, which means I scramble it up in a hopelessly complicated way, rendering it unreadable to anyone except you, the intended recipient of the message. I supply a cryptographic "key" to encrypt the message, and you have to use the same key to decipher or "decrypt" it. At least that's how it works in conventional "single-key" cryptosystems.

In conventional cryptosystems, such as the US Federal Data Encryption Standard (DES), a single key is used for both encryption and decryption. This means that a key must be initially transmitted via secure channels so that both parties can know it before encrypted messages can be sent over insecure channels. This may be inconvenient. If you have a secure channel for exchanging keys, then why do you need cryptography in the first place?

In public key cryptosystems, everyone has two related complementary keys, a publicly revealed key and a secret key. Each key unlocks the code that the other key makes. Knowing the public key does not help you deduce the corresponding secret key. The public key can be published and widely disseminated across a communications network. This protocol provides privacy without the need for the same kind of secure channels that a conventional cryptosystem requires. Anyone can use a recipient's public key to encrypt a message to that person, and that recipient uses her own corresponding secret key to decrypt that message. No one but the recipient can decrypt it, because no one else has access to that secret key. Not even the person who encrypted the message can decrypt it.

Message authentication is also provided. The sender's own secret key can be used to encrypt a message, thereby "signing" it. This creates a digital signature of a message, which the recipient (or anyone else) can check by using the sender's public key to decrypt it. This proves that the sender was the true originator of the message, and that the message has not been subsequently altered by anyone else, because the sender alone possesses the secret key that made that signature. Forgery of a signed message is infeasible, and the sender cannot later disavow his signature.

These two processes can be combined to provide both privacy and authentication by first signing a message with your own secret key, then encrypting the signed message with the recipient's public key. The recipient reverses these steps by first decrypting the message with her own secret key, then checking the enclosed signature with your public key. These steps are done automatically by the recipient's software. Because the public key encryption algorithm is much slower than conventional single-key encryption, encryption is better accomplished by using a high-quality fast conventional single-key encryption algorithm to encipher the message. This original unenciphered message is called "plaintext". In a process invisible to the user, a temporary random key, created just for this one "session", is used to conventionally encipher the plaintext file. Then the recipient's public key is
used to encipher this temporary random conventional key. This public-key-enciphered conventional "session" key is sent along with the enciphered text (called "ciphertext") to the recipient. The recipient uses her own secret key to recover this temporary session key, and then uses that key to run the fast conventional single-key algorithm to decipher the large ciphertext message. Public keys are kept in individual "key certificates" that include the key owner's user ID (which is that person's name), a timestamp of when the key pair was generated, and the actual key material. Public key certificates contain the

public key material, while secret key certificates contain the secret key material. Each secret key is also encrypted with its own password, in case it gets stolen. A key file, or "key ring" contains one or more of these key certificates. Public key rings contain public key certificates, and secret key rings contain secret key certificates. The keys are also internally referenced by a "key ID", which is an "abbreviation" of the public key (the least significant 64 bits of the large public key). When this key ID is displayed, only the lower 32 bits are shown for further brevity. While many keys may share the same user ID, for all practical purposes no two keys share the same key ID.

PGP uses "message digests" to form signatures. A message digest is a 128-bit cryptographically strong one-way hash function of the message. It is somewhat analogous to a "checksum" or CRC error checking code, in that it compactly "represents" the message and is used to detect changes in the message. Unlike a CRC, however, it is computationally infeasible for an attacker to devise a substitute message that would produce an identical message digest. The message digest gets encrypted by the secret key to form a signature. Documents are signed by prefixing them with signature certificates, which contain the key ID of the key that was used to sign it, a secret-key-signed message digest of the document, and a timestamp of when the signature was made. The key ID is used by the receiver to look up the sender's public key to check the signature. The receiver's software automatically looks up the sender's public key and user ID in the receiver's public key ring. Encrypted files are prefixed by the key ID of the public key used to encrypt them. The receiver uses this key ID message prefix to look up the secret key needed to decrypt the message. The receiver's software automatically looks up the necessary secret decryption key in the receiver's secret key ring.

These two types of key rings are the principal method of storing and managing public and secret keys. Rather than keep individual keys in separate key files, they are collected in key rings to facilitate the automatic lookup of keys either by key ID or by user ID. Each user keeps his own pair of key rings. An individual public key is temporarily kept in a separate file long enough to send to your friend who will then add it to her key ring.

Installing PGP

The MSDOS PGP 2.6 release comes in a compressed archive file called PGP26.ZIP (each new release will have a name in the form "PGPxy.ZIP" for PGP version number x.y). The archive can be decompressed with the MSDOS shareware decompression utility PKUNZIP, or the Unix utility "unzip". The PGP release package contains a README.DOC file that you should always read before installing PGP. This README.DOC file contains late-breaking news on what's new in this release of PGP, as well as information on what's in all the other files included in the release. If you already have an earlier version of PGP, you should rename it or delete it, to avoid name conflicts with the new PGP. To install PGP on your MSDOS system, you just have to copy the compressed archive PGPxx.ZIP file into a suitable directory on your hard disk (like C:\PGP), and decompress it with PKUNZIP. For best results, you will also modify your AUTOEXEC.BAT file, as described elsewhere in this manual, but you can do that later, after you've played with PGP a bit and read more of this manual. If you haven't run PGP before, the first step after installation (and reading this manual) is to run the PGP key generation command "pgp -kg". Installing on Unix and VAX/VMS is generally similar to installing on MSDOS, but you may have to compile the source code first. A Unix makefile is provided with the source release for this purpose.

For further details on installation, see the separate PGP Installation Guide, in the file SETUP.DOC included with this release. It fully describes how to set up the PGP directory and your AUTOEXEC.BAT file and how to use PKUNZIP to install it.

How to use PGP

To see a Usage Summary
Encrypting a Message
Encrypting a Message to Multiple Recipients
Signing a Message
Signing and then Encrypting
UsingJust Conventional Encryption
Decrypting and Checking Signatures

To see a quick command usage summary for PGP, just type:

pgp -h

Encrypting a Message

To encrypt a plaintext file with the recipient's public key, type:

pgp -e textfile her_userid

This command produces a ciphertext file called textfile.pgp. A specific example is: pgp -e letter.txt Alice or: pgp -e letter.txt "Alice S"

The first example searches your public key ring file "pubring.pgp" for any public key certificates that contain the string "Alice" anywhere in the user ID field. The second example would find any user IDs that contain "Alice S". You can't use spaces in the string on the command line unless you enclose the whole string in quotes. The search is not case-sensitive. If it finds a matching public key, it uses it to encrypt the plaintext file "letter.txt", producing a ciphertext file called "letter.pgp".

PGP attempts to compress the plaintext before encrypting it, thereby greatly enhancing resistance to cryptanalysis. Thus the ciphertext file will likely be smaller than the plaintext file.

If you want to send this encrypted message through E-mail channels, convert it into printable ASCII "radix-64" format by adding the -a option, as described later.

Encrypting a Message to Multiple Recipients

If you want to send the same message to more than one person, you may specify encryption for several recipients, any of whom may decrypt the same ciphertext file. To specify multiple recipients, just add more user IDs to the command line, like so:

```
pgp -e letter.txt Alice Bob Carol
```

This would create a ciphertext file called letter.pgp that could be decrypted by Alice or Bob or Carol. Any number of recipients may be specified.

Signing a Message

To sign a plaintext file with your secret key, type:

pgp -s textfile [-u your_userid]

Note that [brackets] denote an optional field, so don't actually type real brackets. This command produces a signed file called textfile.pgp. A specific example is:

pgp -s letter.txt -u Bob

This searches your secret key ring file "secring.pgp" for any secret key certificates that contain the string "Bob" anywhere in the user ID field. Your name is Bob, isn't it? The search is not case-sensitive. If it finds a matching secret key, it uses it to sign the plaintext file "letter.txt", producing a signature file called "letter.pgp". If you leave off the user ID field, the first key on your secret key ring is used as the default secret key for your signature.

PGP attempts to compress the message after signing it. Thus the signed file will likely be smaller than the original file, which is useful for archival applications. However, this renders the file unreadable to the casual human observer, even if the original message was ordinary ASCII text. It would be nice if you could make a signed file that was still directly readable to a human. This would be particularly useful if you want to send a signed message as E-mail. For signing E-mail messages, where you most likely do want the result to be human-readable, it is probably most convenient to use the CLEARSIG feature, explained later. This allows the signature to be applied in printable form at the end of the text, and also disables compression of the text. This means the text is still human-readable by the recipient even if the recipient doesn't use PGP to check the signature. This is explained in detail in the section entitled "CLEARSIG - Enable Signed Messages to be Encapsulated as Clear Text", in the Special Topics volume. If you can't wait to read that section of the manual, you can see how an E-mail message signed this way would look, with this example:

pgp -sta message.txt

This would create a signed message in file "message.asc", comprised of the original text, still human-readable, appended with a printable ASCII signature certificate, ready to send through an E-mail system. This example assumes that you are using the normal settings for enabling the CLEARSIG flag in the config file.

Signing and then Encrypting

To sign a plaintext file with your secret key, and then encrypt it with the recipient's public key:

```
pgp -es textfile her_userid [-u your_userid]
```

Note that [brackets] denote an optional field, so don't actually type real brackets.

This example produces a nested ciphertext file called textfile.pgp. Your secret key to create the signature is automatically looked up in your secret key ring via your_userid. Her public encryption key is automatically looked up in your public key ring via her_userid. If you leave off her user ID field from the command line, you will be prompted for it. If you leave off your own user ID field, the first key on your secret key ring is be used as the default secret key for your signature. Note that PGP attempts to compress the plaintext before encrypting it. If you want to send this encrypted message through E-mail channels, convert it into printable ASCII "radix-64" format by adding the -a option, as described later. Multiple recipients may be specified by adding more user IDs to the command line.

Using Just Conventional Encryption

Sometimes you just need to encrypt a file the old-fashioned way, with conventional single-key cryptography. This approach is useful for protecting archive files that will be stored but will not be sent to anyone else. Since the same person that encrypted the file will also decrypt the file, public key cryptography is not really necessary.

To encrypt a plaintext file with just conventional cryptography, type:

```
pgp -c textfile
```

This example encrypts the plaintext file called textfile, producing a ciphertext file called textfile.pgp, without using public key cryptography, key rings, user IDs, or any of that stuff. It prompts you for a pass phrase to use as a conventional key to encipher the file. This pass phrase need not be (and, indeed, SHOULD not be) the same pass phrase that you use to protect your own secret key. Note that PGP attempts to compress the plaintext before encrypting it. PGP will not encrypt the same plaintext the same way twice, even if you used the same pass phrase every time.

Decrypting and Checking Signatures

To decrypt an encrypted file, or to check the signature integrity of a signed file:

pgp ciphertextfile [-o plaintextfile]

Note that [brackets] denote an optional field, so don't actually type real brackets.

The ciphertext file name is assumed to have a default extension of ".pgp". The optional plaintext output file name specifies where to put processed plaintext output. If no name is specified, the ciphertext filename is used, with no extension. If a signature is nested inside of an encrypted file, it is automatically decrypted and the signature integrity is checked. The full user ID of the signer is displayed.

Note that the "unwrapping" of the ciphertext file is completely automatic, regardless of whether the ciphertext file is just signed, just encrypted, or both. PGP uses the key ID prefix in the ciphertext file to automatically find the appropriate secret decryption key on your secret key ring. If there is a nested signature, PGP then uses the key ID prefix in the nested signature to automatically find the appropriate public key on your public key ring to check the signature. If all the right keys are already present on your key rings, no user intervention is required, except that you will be prompted for your password for your secret key if necessary. If the ciphertext file was conventionally encrypted without public key cryptography, PGP recognizes this and prompts you for the pass phrase to conventionally decrypt it.

Managing Keys

Since the time of Julius Caesar, key management has always been the hardest part of cryptography. One of the principal distinguishing features of PGP is its sophisticated key management.

RSA Key Generation
Adding a Key to Your Key Ring
Removing a Key or User ID from Your Key Ring
Extracting (copying) a Key from Your Key Ring
Viewing the Contents of Your Key Ring
How to Protect Public Keys from Tampering
How Does PGP Keep Track of Which Keys are Valid?
How to Protect Secret Keys from Disclosure
Revoking a Public Key
What If You Lose Your Secret Key?

RSA Key Generation

To generate your own unique public/secret key pair of a specified size, type:

pgp -kg

PGP shows you a menu of recommended key sizes (low commercial grade, high commercial grade, or "military" grade) and prompts you for what size key you want, up to more than a thousand bits. The bigger the key, the more security you get, but you pay a price in speed.

It also asks for a user ID, which means your name. It's a good idea to use your full name as your user ID, because then there is less risk of other people using the wrong public key to encrypt messages to you. Spaces and punctuation are allowed in the user ID. It would help if you put your E-mail address in <angle brackets> after your name, like so:

Robert M. Smith <rms@xyzcorp.com>

If you don't have an E-mail address, use your phone number or some other unique information that would help ensure that your user ID is unique.

PGP also asks for a "pass phrase" to protect your secret key in case it falls into the wrong hands. Nobody can use your secret key file without this pass phrase. The pass phrase is like a password, except that it can be a whole phrase or sentence with many words, spaces, punctuation, or anything else you want in it. Don't lose this pass phrase-- there's no way to recover it if you do lose it. This pass phrase will be needed later every time you use your secret key. The pass phrase is case-sensitive, and should not be too short or easy to guess. It is never displayed on the screen. Don't leave it written down anywhere where someone else can see it, and don't store it on your computer. If you don't want a pass phrase (You fool!), just press return
(or enter) at the pass phrase prompt.

The public/secret key pair is derived from large truly random numbers derived mainly from measuring the intervals between your keystrokes with a fast timer. The software will ask you to enter some random text to help it accumulate some random bits for the keys. When asked, you should provide some keystrokes that are reasonably random in their timing, and it wouldn't hurt to make the actual characters that you type irregular in content as well. Some of the randomness is derived from the unpredictability of the content of what you type. So don't just type repeated sequences of characters.

Note that RSA key generation is a lengthy process. It may take a few seconds for a small key on a fast processor, or quite a few minutes for a large key on an old IBM PC/XT. PGP will visually indicate its progress during key generation. The generated key pair will be placed on your public and secret key rings. You can later use the -kx command option to extract (copy) your new public key from your public key ring and place it in a separate public key file suitable for distribution to your friends. The public key file can be sent to your friends for inclusion in their public key rings. Naturally, you keep your secret key file to yourself, and you should include it on your secret key ring. Each secret key on a key ring is individually protected with its own pass phrase.

Never give your secret key to anyone else. For the same reason, don't make key pairs for your friends. Everyone should make their own key pair. Always keep physical control of your secret key, and don't risk exposing it by storing it on a remote timesharing computer. Keep it on your own personal computer.

If PGP complains about not being able to find the PGP User's Guide on your computer, and

refuses to generate a key pair without it, read the explanation of the NOMANUAL parameter in the section "Setting Configuration Parameters" in the Special Topics volume.

Adding a Key to Your Key Ring

Sometimes you will want to add to your keyring a key provided to you by someone else, in the form of a keyfile. To add a public or secret key file's contents to your public or secret key ring (note that [brackets] denote an optional field):

```
pgp -ka keyfile [keyring]
```

The keyfile extension defaults to ".pgp". The optional keyring file name defaults to "pubring.pgp" or "secring.pgp", depending on whether the keyfile contains a public or a secret key. You may specify a different key ring file name, with the extension defaulting to ".pgp". If the key is already on your key ring, PGP will not add it again. All of the keys in the keyfile are added to the keyring, except for duplicates.

Later in the manual, we will explain the concept of certifying keys with signatures. If the key being added has attached signatures certifying it, the signatures are added with the key. If the key is already on your key ring, PGP just merges in any new certifying signatures for that key that you don't already have on your key ring.

PGP was originally designed for handling small personal keyrings. If you want to handle really big keyrings, see the section on "Handling Large Public Keyrings" in the Special Topics volume.

Removing a Key or User ID from Your Key Ring

To remove a key or a user ID from your public key ring:

```
pgp -kr userid [keyring]
```

This searches for the specified user ID in your key ring, and removes it if it finds a match. Remember that any fragment of the user ID will suffice for a match. The optional keyring file name is assumed to be literally "pubring.pgp". It can be omitted, or you can specify "secring.pgp" if you want to remove a secret key. You may specify a different key ring file name. The default key ring extension is ".pgp".

If more than one user ID exists for this key, you will be asked if you want to remove only the user ID you specified, while leaving the key and its other user IDs intact.

Extracting (copying) a Key from Your Key Ring

To extract (copy) a key from your public or secret key ring:

```
pgp -kx userid keyfile [keyring]
```

This non-destructively copies the key specified by the user ID from your public or secret key ring to the specified key file. This is particularly useful if you want to give a copy of your public key to someone else. If the key has any certifying signatures attached to it on your key ring, they are copied off along with the key.

If you want the extracted key represented in printable ASCII characters suitable for email purposes, use the -kxa options.

Viewing the Contents of Your Key Ring

To view the contents of your public key ring:

pgp -kv[v] [userid] [keyring]

This lists any keys in the key ring that match the specified user ID substring. If you omit the user ID, all of the keys in the key ring are listed. The optional keyring file name is assumed to be "pubring.pgp". It can be omitted, or you can specify "secring.pgp" if you want to list secret keys. If you want to specify a different key ring file name, you can. The default key ring extension is
".pgp".

Later in the manual, we will explain the concept of certifying keys with signatures. To see all the certifying signatures attached to each key, use the -kvv option:

pgp -kvv [userid] [keyring]

If you want to specify a particular key ring file name, but want to see all the keys in it, try this alternative approach:

pgp keyfile

With no command options specified, PGP lists all the keys in keyfile.pgp, and also attempts to add them to your key ring if they are not already on your key ring.

How to Protect Public Keys from Tampering

In a public key cryptosystem, you don't have to protect public keys from exposure. In fact, it's better if they are widely disseminated. But it is important to protect public keys from tampering, to make sure that a public key really belongs to whom it appears to belong to. This may be the most important vulnerability of a public-key cryptosystem. Let's first look at a potential disaster, then at how to safely avoid it with PGP.

Suppose you wanted to send a private message to Alice. You download Alice's public key certificate from an electronic bulletin board system (BBS). You encrypt your letter to Alice with this public key and send it to her through the BBS's E-mail facility. Unfortunately, unbeknownst to you or Alice, another user named Charlie has infiltrated the BBS and generated a public key of his own with Alice's user ID attached to it. He covertly substitutes his bogus key in place of Alice's real public key. You unwittingly use this bogus key belonging to Charlie instead of Alice's public key. All looks normal because this bogus key has Alice's user ID. Now Charlie can decipher the message intended for Alice because he has the matching secret key. He may even re-encrypt the deciphered message with Alice's real public key and send it on to her so that no one suspects any wrongdoing. Furthermore, he can even make apparently good signatures from Alice with this secret key because everyone will use the bogus public key to check Alice's signatures.

The only way to prevent this disaster is to prevent anyone from tampering with public keys. If you got Alice's public key directly from Alice, this is no problem. But that may be difficult if Alice is a thousand miles away, or is currently unreachable.

Perhaps you could get Alice's public key from a mutual trusted friend David who knows he has a good copy of Alice's public key. David could sign Alice's public key, vouching for the integrity of Alice's public key. David would create this signature with his own secret key. This would create a signed public key certificate, and would show that Alice's key had not been tampered with. This requires you have a known good copy of David's public key to check his signature. Perhaps David could provide Alice with a signed copy of your public key also. David is thus serving as an "introducer" between you and Alice.

This signed public key certificate for Alice could be uploaded by David or Alice to the BBS, and you could download it later. You could then check the signature via David's public key and thus be assured that this is really Alice's public key. No impostor can fool you into accepting his own bogus key as Alice's because no one else can forge signatures made by David.

A widely trusted person could even specialize in providing this service of "introducing" users to each other by providing signatures for their public key certificates. This trusted person could be regarded as a "key server", or as a "Certifying Authority". Any public key certificates bearing the key server's signature could be trusted as truly belonging to whom they appear to belong to. All users who wanted to participate would need a known good copy of just the key server's public key, so that the key server's signatures could be verified. A trusted centralized key server or Certifying Authority is especially appropriate for large impersonal centrally-controlled corporate or government institutions. Some institutional environments use hierarchies of Certifying Authorities.

For more decentralized grassroots "guerrilla style" environments, allowing all users to act as a trusted introducers for their friends would probably work better than a centralized key server. PGP tends to emphasize this organic decentralized non-institutional approach. It better reflects the natural way humans interact on a personal social level, and allows people to better choose who they can trust for key management.

This whole business of protecting public keys from tampering is the single most difficult

problem in practical public key applications. It is the Achilles' heel of public key cryptography, and a lot of software complexity is tied up in solving this one problem.

You should use a public key only after you are sure that it is a good public key that has not been tampered with, and actually belongs to the person it claims to. You can be sure of this if you got this public key certificate directly from its owner, or if it bears the signature of someone else that you trust, from whom you already have a good public key. Also, the user ID should have the full name of the key's owner, not just her first name.

No matter how tempted you are-- and you will be tempted-- never, NEVER give in to expediency and trust a public key you downloaded from a bulletin board, unless it is signed by someone you trust. That uncertified public key could have been tampered with by anyone, maybe even by the system administrator of the bulletin board.

If you are asked to sign someone else's public key certificate, make certain that it really belongs to that person named in the user ID of that public key certificate. This is because your signature on her public key certificate is a promise by you that this public key really belongs to her. Other people who trust you will accept her public key because it bears your signature. It may be ill- advised to rely on hearsay-- don't sign her public key unless you have independent firsthand knowledge that it really belongs to her. Preferably, you should sign it only if you got it directly from her.

In order to sign a public key, you must be far more certain of that key's ownership than if you merely want to use that key to encrypt a message. To be convinced of a key's validity enough to use it, certifying signatures from trusted introducers should suffice. But to sign a key yourself, you should require your own independent firsthand knowledge of who owns that key. Perhaps you could call the key's owner on the phone and read the key file to her to get her to confirm that the key you have really is her key-- and make sure you really are talking to the right person. See the section called "Verifying a Public Key Over the Phone" in the Special Topics volume for further details.

Bear in mind that your signature on a public key certificate does not vouch for the integrity of that person, but only vouches for the integrity (the ownership) of that person's public key. You aren't risking your credibility by signing the public key of a sociopath, if you were completely confident that the key really belonged to him. Other people would accept that key as belonging to him because you signed it (assuming they trust you), but they wouldn't trust that key's owner. Trusting a key is not the same as trusting the key's owner.

Trust is not necessarily transferable; I have a friend who I trust not to lie. He's a gullible person who trusts the President not to lie. That doesn't mean I trust the President not to lie. This is just common sense. If I trust Alice's signature on a key, and Alice trusts Charlie's signature on a key, that does not imply that I have to trust Charlie's signature on a key.

It would be a good idea to keep your own public key on hand with a collection of certifying signatures attached from a variety of "introducers", in the hopes that most people will trust at least one of the introducers who vouch for your own public key's validity. You could post your key with its attached collection of certifying signatures on various electronic bulletin boards. If you sign someone else's public key, return it to them with your signature so that they can add it to their own collection of credentials for their own public key.

PGP keeps track of which keys on your public key ring are properly certified with signatures from introducers that you trust. All you have to do is tell PGP which people you trust as introducers, and certify their keys yourself with your own ultimately trusted key. PGP can

take it from there, automatically validating any other keys that have been signed by your designated introducers. And of course you may directly sign more keys yourself. More on this later.

Make sure no one else can tamper with your own public key ring. Checking a new signed public key certificate must ultimately depend on the integrity of the trusted public keys that are already on your own public key ring. Maintain physical control of your public key ring, preferably on your own personal computer rather than on a remote timesharing system, just as you would do for your secret key. This is to protect it from tampering, not from disclosure. Keep a trusted backup copy of your public key ring and your secret key ring on write-protected media.

Since your own trusted public key is used as a final authority to directly or indirectly certify all the other keys on your key ring, it is the most important key to protect from tampering. To detect any tampering of your own ultimately-trusted public key, PGP can be set up to automatically compare your public key against a backup copy on write-protected media. For details, see the description of the "-kc" key ring check command in the Special Topics volume.

PGP generally assumes you will maintain physical security over your system and your key rings, as well as your copy of PGP itself. If an intruder can tamper with your disk, then in theory he can tamper with PGP itself, rendering moot the safeguards PGP may have to detect tampering with keys.

One somewhat complicated way to protect your own whole public key ring from tampering is to sign the whole ring with your own secret key. You could do this by making a detached signature certificate of the public key ring, by signing the ring with the "-sb" options (see the section called "Separating Signatures from Messages" in the PGP User's Guide, Special Topics volume). Unfortunately, you would still have to keep a separate trusted copy of your own public key around to check the signature you made. You couldn't rely on your own public key stored on your public key ring to check the signature you made for the whole ring, because that is part of what you're trying to check.

How Does PGP Keep Track of Which Keys are Valid?

Before you read this section, be sure to read the above section on "How to Protect Public Keys from Tampering".

PGP keeps track of which keys on your public key ring are properly certified with signatures from introducers that you trust. All you have to do is tell PGP which people you trust as introducers, and certify their keys yourself with your own ultimately trusted key. PGP can take it from there, automatically validating any other keys that have been signed by your designated introducers. And of course you may directly sign more keys yourself.

There are two entirely separate criteria PGP uses to judge a public key's usefulness-- don't get them confused:

1) Does the key actually belong to whom it appears to belong?
   In other words, has it been certified with a trusted signature?
2) Does it belong to someone you can trust to certify other keys?

PGP can calculate the answer to the first question. To answer the second question, PGP must be explicitly told by you, the user. When you supply the answer to question 2, PGP can then calculate the answer to question 1 for other keys signed by the introducer you designated as trusted.

Keys that have been certified by a trusted introducer are deemed valid by PGP. The keys belonging to trusted introducers must themselves be certified either by you or by other trusted introducers.

PGP also allows for the possibility of you having several shades of trust for people to act as introducers. Your trust for a key's owner to act as an introducer does not just reflect your estimation of their personal integrity-- it should also reflect how competent you think they are at understanding key management and using good judgment in signing keys. You can designate a person to PGP as unknown, untrusted, marginally trusted, or completely trusted to certify other public keys. This trust information is stored on your key ring with their key, but when you tell PGP to copy a key off your key ring, PGP will not copy the trust information along with the key, because your private opinions on trust are regarded as confidential.

When PGP is calculating the validity of a public key, it examines the trust level of all the attached certifying signatures. It computes a weighted score of validity-- two marginally trusted signatures are deemed as credible as one fully trusted signature. PGP's skepticism is adjustable-- for example, you may tune PGP to require two fully trusted signatures or three marginally trusted signatures to judge a key as valid.

Your own key is "axiomatically" valid to PGP, needing no introducer's signature to prove its validity. PGP knows which public keys are yours, by looking for the corresponding secret keys on the secret key ring. PGP also assumes you ultimately trust yourself to certify other keys.

As time goes on, you will accumulate keys from other people that you may want to designate as trusted introducers. Everyone else will each choose their own trusted introducers. And everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people, with the expectation that anyone receiving it will trust at least one or two of the signatures. This will cause the emergence of a decentralized fault-tolerant web of confidence for all public keys.

This unique grass-roots approach contrasts sharply with Government standard public key

management schemes, such as Internet Privacy Enhanced Mail (PEM), which are based on centralized control and mandatory centralized trust. The standard schemes rely on a hierarchy of Certifying Authorities who dictate who you must trust. PGP's decentralized probabilistic method for determining public key legitimacy is the centerpiece of its key management architecture. PGP lets you alone choose who you trust, putting you at the top of your own private certification pyramid. PGP is for people who prefer to pack their own parachutes.

How to Protect Secret Keys from Disclosure

Protect your own secret key and your pass phrase carefully. Really, really carefully. If your secret key is ever compromised, you'd better get the word out quickly to all interested parties (good luck) before someone else uses it to make signatures in your name. For example, they could use it to sign bogus public key certificates, which could create problems for many people, especially if your signature is widely trusted. And of course, a compromise of your own secret key could expose all messages sent to you.

To protect your secret key, you can start by always keeping physical control of your secret key. Keeping it on your personal computer at home is OK, or keep it in your notebook computer that you can carry with you. If you must use an office computer that you don't always have physical control of, then keep your public and secret key rings on a write-protected removable floppy disk, and don't leave it behind when you leave the office. It wouldn't be a good idea to allow your secret key to reside on a remote timesharing computer, such as a remote dial-in Unix system. Someone could eavesdrop on your modem line and capture your pass phrase, and then obtain your actual secret key from the remote system. You should only use your secret key on a machine that you have physical control over.

Don't store your pass phrase anywhere on the computer that has your secret key file. Storing both the secret key and the pass phrase on the same computer is as dangerous as keeping your PIN in the same wallet as your Automatic Teller Machine bank card. You don't want somebody to get their hands on your disk containing both the pass phrase and the secret key file. It would be most secure if you just memorize your pass phrase and don't store it anywhere but your brain. If you feel you must write down your pass phrase, keep it well protected, perhaps even more well protected than the secret key file.

And keep backup copies of your secret key ring-- remember, you have the only copy of your secret key, and losing it will render useless all the copies of your public key that you have spread throughout the world.

The decentralized non-institutional approach PGP uses to manage public keys has its benefits, but unfortunately this also means we can't rely on a single centralized list of which keys have been compromised. This makes it a bit harder to contain the damage of a secret key compromise. You just have to spread the word and hope everyone hears about it.

If the worst case happens-- your secret key and pass phrase are both compromised (hopefully you will find this out somehow)-- you will have to issue a "key compromise" certificate. This kind of certificate is used to warn other people to stop using your public key. You can use PGP to create such a certificate by using the "-kd" command. Then you must somehow send this compromise certificate to everyone else on the planet, or at least to all your friends and their friends, et cetera. Their own PGP software will install this key compromise certificate on their public key rings and will automatically prevent them from accidentally using your public key ever again. You can then generate a new secret/public key pair and publish the new public key. You could send out one package containing both your new public key and the key compromise certificate for your old key.

Revoking a Public Key

Key/userid revocation certificates (PGP 2.62)

Suppose your secret key and your pass phrase are somehow both compromised. You have to get the word out to the rest of the world, so that they will all stop using your public key. To do this, you will have to issue a "key compromise", or "key revocation" certificate to revoke your public key.

To generate a certificate to revoke your own key, use the -kd command:

```
pgp -kd your_userid
```

This certificate bears your signature, made with the same key you are revoking. You should widely disseminate this key revocation certificate as soon as possible. Other people who receive it can add it to their public key rings, and their PGP software then automatically prevents them from accidentally using your old public key ever again. You can then generate a new secret/public key pair and publish the new public key.

You may choose to revoke your key for some other reason than the compromise of a secret key. If so, you may still use the same mechanism to revoke it.

What If You Lose Your Secret Key?

Normally, if you want to revoke your own secret key, you can use the "-kd" command to issue a revocation certificate, signed with your own secret key (see "Revoking a Public Key").

But what can you do if you lose your secret key, or if your secret key is destroyed? You can't revoke it yourself, because you must use your own secret key to revoke it, and you don't have it anymore. A future version of PGP will offer a more secure means of revoking keys in these circumstances, allowing trusted introducers to certify that a public key has been revoked. But for now, you will have to get the word out through whatever informal means you can, asking users to "disable" your public key on their own individual public key rings.

Other users may disable your public key on their own public key rings by using the "-kd" command. If a user ID is specified that does not correspond to a secret key on the secret key ring, the -kd command will look for that user ID on the public key ring, and mark that public key as disabled. A disabled key may not be used to encrypt any messages, and may not be extracted from the key ring with the -kx command. It can still be used to check signatures, but a warning is displayed. And if the user tries to add the same key again to his key ring, it will not work because the disabled key is already on the key ring. These combined features will help curtail the further spread of a disabled key.

If the specified public key is already disabled, the -kd command will ask if you want the key reenabled.

Sending Ciphertext Through E-mail Channels: Radix-64 Format

To use services like AOL and Prodigy keep in mind that all you need to do is make sure that you turn output into ASCII and the RADIX-64 format so that the line breaks don't get messed up in the email.

do not leave any keys or encrypted messages on any public message board as the services do tend to get a little jumpy about PGP and may cause trouble (also, to some people it looks like gibberish) exchange keys as a file transfer or as direct e-mail.

Many electronic mail systems only allow messages made of ASCII text, not the 8-bit raw binary data that ciphertext is made of. To get around this problem, PGP supports ASCII radix-64 format for ciphertext messages, similar to the Internet Privacy-Enhanced Mail (PEM) format, as well as the Internet MIME format. This special format represents binary data by using only printable ASCII characters, so it is useful for transmitting binary encrypted data through 7-bit channels or for sending binary encrypted data as normal E-mail text. This format acts as a form of "transport armor", protecting it against corruption as it travels through intersystem gateways on Internet. PGP also appends a CRC to detect transmission errors.

Radix-64 format converts the plaintext by expanding groups of 3 binary 8-bit bytes into 4 printable ASCII characters, so the file grows by about 33%. But this expansion isn't so bad when you consider that the file probably was compressed more than that by PGP before it was encrypted.

To produce a ciphertext file in ASCII radix-64 format, just add the "a" option when encrypting or signing a message, like so:

```
pgp -esa message.txt her_userid
```

This example produces a ciphertext file called "message.asc" that contains data in a PEM-like ASCII radix-64 format. This file can be easily uploaded into a text editor through 7-bit channels for transmission as normal E-mail on Internet or any other E-mail network.

Decrypting the radix-64 transport-armored message is no different than a normal decrypt. For example:

```
pgp message
```

PGP automatically looks for the ASCII file "message.asc" before it looks for the binary file "message.pgp". It recognizes that the file is in radix-64 format and converts it back to binary before processing as it normally does, producing as a by-product a ".pgp" ciphertext file in binary form. The final output file is in normal plaintext form, just as it was in the original file "message.txt".

Most Internet E-mail facilities prohibit sending messages that are more than 50000 bytes long. Longer messages must be broken into smaller chunks that can be mailed separately. If your encrypted message is very large, and you requested radix-64 format, PGP automatically breaks it up into chunks that are each small enough to send via E-mail. The chunks are put into files named with extensions ".as1", ".as2", ".as3", etc. The recipient must concatenate these separate files back together in their proper order into one big file before decrypting it. While decrypting, PGP ignores any extraneous text in mail headers that are not enclosed in the radix-64 message blocks.

If you want to send a public key to someone else in radix-64 format, just add the -a option while extracting the key from your keyring.

If you forgot to use the -a option when you made a ciphertext file or extracted a key, you may still directly convert the binary file into radix-64 format by simply using the -a option alone, without any

encryption specified. PGP converts it to a ".asc" file.

If you sign a plaintext file without encrypting it, PGP will normally compress it after signing it, rendering it unreadable to the casual human observer. This is a suitable way of storing signed files in archival applications. But if you want to send the signed message as E-mail, and the the original plaintext message is in text (not binary) form, there is a way to send it through an E-mail channel in such a way that the plaintext does not get compressed, and the ASCII armor is applied only to the binary signature certificate, but not to the plaintext message. This makes it possible for the recipient to read the signed message with human eyes, without the aid of PGP. Of course, PGP is still needed to actually check the signature. For further information on this feature, see the explanation of the "CLEARSIG" parameter in the section "Setting Configuration Parameters: CONFIG.TXT" in the Special Topics volume.

Sometimes you may want to send a binary data file through an E-mail channel without encrypting or signing it with PGP. Some people use the Unix uuencode utility for that purpose. PGP can also be used for that purpose, by simply using the -a option alone, and it does a better job than the uuencode utility. For further details, see the section on "Using PGP as a Better Uuencode" in the Special Topics volume.

Advanced Topics

Sending Ciphertext Through E-mail Channels: Radix-64 Format
Environmental Variable for Path Name
Setting Configuration Parameters: CONFIG.TXT
Beware of Snake Oil
Vulnerabilities

Environmental Variable for Path Name

PGP uses several special files for its purposes, such as your standard key ring files "pubring.pgp" and "secring.pgp", the random number seed file "randseed.bin", the PGP configuration file "config.txt", and the foreign language string translation file "language.txt". These special files can be kept in any directory, by setting the environmental variable "PGPPATH" to the desired pathname. For example, on MSDOS, the shell command:

```
SET PGPPATH=C:\PGP
```

makes PGP assume that your public key ring filename is "C:\PGP\pubring.pgp". Assuming, of course, that this directory exists. Use your favorite text editor to modify your MSDOS AUTOEXEC.BAT file to automatically set up this variable whenever you start up your system. If PGPPATH remains undefined, these special files are assumed to be in the current directory.

Setting Configuration Parameters: CONFIG.TXT

PGP has a number of user-settable parameters that can be defined in a special configuration text file called "config.txt", in the directory pointed to by the shell environmental variable PGPPATH. Having a configuration file enables the user to define various flags and parameters for PGP without the burden of having to always define these parameters in the PGP command line.

With these configuration parameters, for example, you can control where PGP stores its temporary scratch files, or you can select what foreign language PGP will use to display its diagnostics messages and user prompts, or you can adjust PGP's level of skepticism in determining a key's validity based on the number of certifying signatures it has.

For more details on setting these configuration parameters, see the appropriate section of the PGP User's Guide, Special Topics volume.

COMMANDS

Vulnerabilities

No data security system is impenetrable. PGP can be circumvented in a variety of ways. Potential vulnerabilities you should be aware of include compromising your pass phrase or secret key, public key tampering, files that you deleted but are still somewhere on the disk, viruses and Trojan horses, breaches in your physical security, electromagnetic emissions, exposure on multi-user systems, traffic analysis, and perhaps even direct cryptanalysis.

For a detailed discussion of these issues, see the "Vulnerabilities" section in the PGP User's Guide, Special Topics volume.

Beware of Snake Oil

When examining a cryptographic software package, the question always remains, why should you trust this product? Even if you examined the source code yourself, not everyone has the cryptographic experience to judge the security. Even if you are an experienced cryptographer, subtle weaknesses in the algorithms could still elude you.

When I was in college in the early seventies, I devised what I believed was a brilliant encryption scheme. A simple pseudorandom number stream was added to the plaintext stream to create ciphertext. This would seemingly thwart any frequency analysis of the ciphertext, and would be uncrackable even to the most resourceful Government intelligence agencies. I felt so smug about my achievement. So cock-sure.

Years later, I discovered this same scheme in several introductory cryptography texts and tutorial papers. How nice. Other cryptographers had thought of the same scheme. Unfortunately, the scheme was presented as a simple homework assignment on how to use elementary cryptanalytic techniques to trivially crack it. So much for my brilliant scheme.

From this humbling experience I learned how easy it is to fall into a false sense of security when devising an encryption algorithm. Most people don't realize how fiendishly difficult it is to devise an encryption algorithm that can withstand a prolonged and determined attack by a resourceful opponent. Many mainstream software engineers have developed equally naive encryption schemes (often even the very same encryption scheme), and some of them have been incorporated into commercial encryption software packages and sold for good money to thousands of unsuspecting users.

This is like selling automotive seat belts that look good and feel good, but snap open in even the slowest crash test. Depending on them may be worse than not wearing seat belts at all. No one suspects they are bad until a real crash. Depending on weak cryptographic software may cause you to unknowingly place sensitive information at risk. You might not otherwise have done so if you had no cryptographic software at all. Perhaps you may never even discover your data has been compromised.

Sometimes commercial packages use the Federal Data Encryption Standard (DES), a good conventional algorithm recommended by the Government for commercial use (but not for classified information, oddly enough-- hmmm). There are several "modes of operation" the DES can use, some of them better than others. The Government specifically recommends not using the weakest simplest mode for messages, the Electronic Codebook (ECB) mode. But they do recommend the stronger and more complex Cipher Feedback (CFB) or Cipher Block Chaining (CBC) modes.

Unfortunately, most of the commercial encryption packages I've looked at use ECB mode. When I've talked to the authors of a number of these implementations, they say they've never heard of CBC or CFB modes, and didn't know anything about the weaknesses of ECB mode. The very fact that they haven't even learned enough cryptography to know these elementary concepts is not reassuring. These same software packages often include a second faster encryption algorithm that can be used instead of the slower DES. The author of the package often thinks his proprietary faster algorithm is as secure as the DES, but after questioning him I usually discover that it's just a variation of my own brilliant scheme from college days. Or maybe he won't even reveal how his proprietary encryption scheme works, but assures me it's a brilliant scheme and I should trust it. I'm sure he believes that his algorithm is brilliant, but how can I know that without seeing it?

In all fairness I must point out that in most cases these products do not come from companies that specialize in cryptographic technology.

There is a company called AccessData (87 East 600 South, Orem, Utah 84058, phone 1-800-658-5199) that sells a package for $185 that cracks the built-in encryption schemes used by WordPerfect, Lotus 1-2-3, MS Excel, Symphony, Quattro Pro, Paradox, and MS Word 2.0. It doesn't simply guess passwords-- it does real cryptanalysis. Some people buy it when they forget their password for their own files. Law enforcement agencies buy it too, so they can read files they seize. I talked to Eric Thompson, the author, and he said his program only takes a split second to crack them, but he put in some delay loops to slow it down so it doesn't look so easy to the customer. He also told me that the password encryption feature of PKZIP files can often be easily broken, and that his law enforcement customers already have that service regularly provided to them from another vendor.

In some ways, cryptography is like pharmaceuticals. Its integrity may be absolutely crucial. Bad penicillin looks the same as good penicillin. You can tell if your spreadsheet software is wrong, but how do you tell if your cryptography package is weak? The ciphertext produced by a weak encryption algorithm looks as good as ciphertext produced by a strong encryption algorithm. There's a lot of snake oil out there. A lot of quack cures. Unlike the patent medicine hucksters of old, these software implementors usually don't even know their stuff is snake oil. They may be good software engineers, but they usually haven't even read any of the academic literature in cryptography. But they think they can write good cryptographic software. And why not? After all, it seems intuitively easy to do so. And their software seems to work okay. Anyone who thinks they have devised an unbreakable encryption scheme either is an incredibly rare genius or is naive and inexperienced.

I remember a conversation with Brian Snow, a highly placed senior cryptographer with the NSA. He said he would never trust an encryption algorithm designed by someone who had not "earned their bones" by first spending a lot of time cracking codes. That did make a lot of sense. I observed that practically no one in the commercial world of cryptography qualified under this criterion. "Yes", he said with a self assured smile, "And that makes our job at NSA so much easier." A chilling thought. I didn't qualify either.

The Government has peddled snake oil too. After World War II, the US sold German Enigma ciphering machines to third world governments. But they didn't tell them that the Allies cracked the Enigma code during the war, a fact that remained classified for many years. Even today many Unix systems worldwide use the Enigma cipher for file encryption, in part because the Government has created legal obstacles against using better algorithms. They even tried to prevent the initial publication of the RSA algorithm in 1977. And they have squashed essentially all commercial efforts to develop effective secure telephones for the general public.

The principal job of the US Government's National Security Agency is to gather intelligence, principally by covertly tapping into people's private communications (see James Bamford's book, "The Puzzle Palace"). The NSA has amassed considerable skill and resources for cracking codes. When people can't get good cryptography to protect themselves, it makes NSA's job much easier. NSA also has the responsibility of approving and recommending encryption algorithms. Some critics charge that this is a conflict of interest, like putting the fox in charge of guarding the hen house. NSA has been pushing a conventional encryption algorithm that they designed, and they won't tell anybody how it works because that's classified. They want others to trust it and use it. But any cryptographer can tell you that a well-designed encryption algorithm does not have to be classified to remain secure. Only the keys should need protection. How does anyone else really know if NSA's classified algorithm is secure? It's not that hard for NSA to design an encryption algorithm that only they can crack, if no one else can review the algorithm. Are

they deliberately selling snake oil?

I'm not as certain about the security of PGP as I once was about my brilliant encryption software from college. If I were, that would be a bad sign. But I'm pretty sure that PGP does not contain any glaring weaknesses. The crypto algorithms were developed by people at high levels of civilian cryptographic academia, and have been individually subject to extensive peer review. Source code is available to facilitate peer review of PGP and to help dispel the fears of some users. It's reasonably well researched, and has been years in the making. And I don't work for the NSA. I hope it doesn't require too large a "leap of faith" to trust the security of PGP.

Legal Issues

For detailed information on PGP(tm) licensing, distribution, copyrights, patents, trademarks, liability limitations, and export controls, see the "Legal Issues" section in the "PGP User's Guide, Volume II: Special Topics".

PGP uses a public key algorithm claimed by U.S. patent #4,405,829. The exclusive licensing rights to this patent are held by a California company called Public Key Partners, and you may be infringing the patent if you use PGP in the USA without a license. These issues are detailed in the Volume II manual, and in the RSAREF license that comes with the freeware version of PGP. PKP has licensed others to practice the patent, including a company known as ViaCrypt, in Phoenix, Arizona. ViaCrypt sells a fully licensed version of PGP. ViaCrypt may be reached at 602-944-0773.

PGP is "guerrilla" freeware, and I don't mind if you distribute it widely. Just don't ask me to send you a copy. Instead, you can look for it yourself on many BBS systems and a number of Internet FTP sites. But before you distribute PGP, it is essential that you understand the U.S. export
controls on encryption software.

Acknowledgements

Formidable obstacles and powerful forces have been arrayed to stop PGP. Dedicated peopleare helping to overcome these obstacles. PGP has achieved notoriety as "underground software", and bringing PGP "above ground" as fully licensed freeware has required patience and persistence. I'd especially like to thank Hal Abelson, Jeff Schiller, Brian LaMacchia, and Derek Atkins at MIT for their determined efforts. I'd also like to thank Jim Bruce and David Litster in the MIT administration and Bob Prior and Terry Ehling at the MIT Press. And I'd like to thank my entire legal defense team, whose job is not over yet. I used to tell a lot of lawyer jokes, before I encountered so many positive examples of lawyers in my legal defense team, most of whom work pro bono.

The development of PGP has turned into a remarkable social phenomenon, whose unique political appeal has inspired the collective efforts of an ever-growing number of volunteer programmers. Remember that children's story called "Stone Soup"?

I'd like to thank the following people for their contributions to the creation of Pretty Good Privacy. Although I was the author of PGP version 1.0, major parts of later versions of PGP were implemented by an international collaborative effort involving a large number of contributors, under my design guidance.

Branko Lankester, Hal Finney and Peter Gutmann all contributed a huge amount of time in adding features for PGP 2.0, and ported it to Unix variants.

Hugh Kennedy ported it to VAX/VMS, Lutz Frank ported it to the Atari ST, and Cor Bosman and Colin Plumb ported it to the Commodore Amiga.

Translation of PGP into foreign languages was done by Jean-loup Gailly in France, Armando Ramos in Spain, Felipe Rodriquez Svensson and Branko Lankester in The Netherlands, Miguel Angel Gallardo in Spain, Hugh Kennedy and Lutz Frank in Germany, David Vincenzetti in Italy, Harry Bush and Maris Gabalins in Latvia, Zygimantas Cepaitis in Lithuania, Peter Suchkow and Andrew Chernov in Russia, and Alexander Smishlajev in Esperantujo. Peter Gutmann offered to translate it into New Zealand English, but we finally decided PGP could get by with US English.

Jean-loup Gailly, Mark Adler, and Richard B. Wales published the ZIP compression code, and granted permission for inclusion into PGP. The MD5 routines were developed and placed in the public domain by Ron Rivest. The IDEA(tm) cipher was developed by Xuejia Lai and James L. Massey at ETH in Zurich, and is used in PGP with permission from Ascom-Tech AG.

Charlie Merritt originally taught me how to do decent multiprecision arithmetic for public key cryptography, and Jimmy Upton contributed a faster multiply/modulo algorithm. Thad Smith implemented an even faster modmult algorithm. Zhahai Stewart contributed a lot of useful ideas on PGP file formats and other stuff, including having more than one user ID for a key. I heard the idea of introducers from Whit Diffie. Kelly Goen did most of the work for the initial electronic publication of PGP 1.0.

Various contributions of coding effort also came from Colin Plumb, Derek Atkins, and Castor Fu. Other contributions of effort, coding or otherwise, have come from Hugh Miller, Eric Hughes, Tim May, Stephan Neuhaus, and too many others for me to remember right now. Zbigniew Fiedorwicz did a Macintosh port.

Since the release of PGP 2.0, many other programmers have sent in patches and bug fixes and porting adjustments for other computers. There are too many to individually thank here.

Just as in the "Stone Soup" story, it is getting harder to peer through the thick soup to see the stone at the bottom of the pot that I dropped in to start it all off.

About The Author

Philip Zimmermann is a software engineer consultant with 19 years experience, specializing in embedded real-time systems, cryptography, authentication, and data communications. Experience includes design and implementation of authentication systems for financial information networks, network data security, key management protocols, embedded real-time multitasking executives, operating systems, and local area networks.

Custom versions of cryptography and authentication products and public key implementations such as the NIST DSS are available from Zimmermann, as well as custom product development services. His consulting firm's address is:

Boulder Software Engineering 3021 Eleventh Street Boulder, Colorado 80304 USA Phone: 303-541-0140 (10:00am - 7:00pm Mountain Time) Fax: arrange by phone Internet: prz@acm.org

Keyrings and Key Management

Editing Your User ID or Pass Phrase
Editing the Trust Parameters for a Public Key
Checking if everything is OK on Your Public Key Ring
Verifying a Public Key Over the Phone
Handling Large Public Keyrings
Selecting Keys Via Key ID
Revoking Keys

BATCHMODE

Environmental Variable for Pass Phrase
Force "Yes" Answer to Confirmation Questions : FORCE
PGP Returns Exit Status to the Shell
Suppressing Unnecessary Questions

Setting Configuration Parameters:Config.txt

COMMANDS

PGP has a number of user-settable parameters that can be defined in a special configuration text file called "config.txt", in the directory pointed to by the shell environmental variable PGPPATH. Having a configuration file enables the user to define various flags and parameters for PGP without the burden of having to always define these parameters in the PGP command line.

Configuration parameters may be assigned integer values, character string values, or on/off values, depending on what kind of configuration parameter it is. A sample configuration file is provided with PGP, so you can see some examples. In the configuration file, blank lines are ignored, as is anything following the '#' comment character. Keywords are not case-sensitive.

Here is a short sample fragment of a typical configuration file:

```
# TMP is the directory for PGP scratch files, such as a RAM disk.
TMP = "e:\"      # Can be overridden by environment variable TMP.
Armor = on       # Use -a flag for ASCII armor whenever applicable.
# CERT_DEPTH is how deeply introducers may introduce introducers.
cert_depth = 3
```

If some configuration parameters are not defined in the configuration file, or if there is no configuration file, or if PGP can't find the configuration file, the values for the configuration parameters default to some reasonable value. Note that it is also possible to set these same configuration parameters directly from the PGP command line, by preceding the parameter setting with a "+" character. For example, the following two PGP commands produce the same effect:

pgp -e +armor=on message.txt smith or: pgp -ea message.txt smith

See "COMMANDS" at the top of this window for a summary of the various parameters than may be defined in the configuration file.

The filename "config.txt" has been in use for a long time by PGP, but some folks have pointed out that it may be at odds with naming conventions for configuration files for specific operating systems. Accordingly, PGP now tries to open this filename only after first trying to open the file ".pgprc" on Unix platforms, and "pgp.ini" on other platforms, in the same directory that PGP would look for "config.txt".

Handling of Text

Leaving No Traces of Plaintext on the Disk
Displaying Decrypted Plaintext on your Screen
Making a Message For her eyes only
Preserving the Original Plaintext Filename
Sending ASCII Text files Across Different Environments

Separating Signatures from Messages

Key/userid revocation certificates (PGP 2.62)

Normally, signature certificates are physically attached to the text they sign. This makes it convenient in simple cases to check signatures. It is desirable in some circumstances to have signature certificates stored separately from the messages they sign. It is possible to generate signature certificates that are detached from the text they sign. To do this, combine the 'b' (break) option with the 's' (sign) option. For example:

pgp -sb letter.txt

This example produces an isolated signature certificate in a file called "letter.sig". The contents of letter.txt are not appended to the signature certificate.

After creating the signature certificate file (letter.sig in the above example), send it along with the original text file to the recipient. The recipient must have both files to check the signature integrity. When the recipient attempts to process the signature file, PGP notices that there is no text in the same file with the signature and prompts the user for the filename of the text. Only then can PGP properly check the signature integrity. If the recipient knows in advance that the signature is detached from the text file, she can specify both filenames on the command line:

pgp letter.sig letter.txt or: pgp letter letter.txt

PGP will not have to prompt for the text file name in this case.

A detached signature certificate is useful if you want to keep the signature certificate in a separate certificate log. A detached signature of an executable program is also useful for detecting a subsequent virus infection. It is also useful if more than one party must sign a document such as a legal contract, without nesting signatures. Each person's signature is independent.

If you receive a ciphertext file that has the signature certificate glued to the message, you can still pry the signature certificate away from the message during the decryption. You can do this with the -b option during decrypt, like so:

pgp -b letter

This decrypts the letter.pgp file and if there is a signature in it, PGP checks the signature and detaches it from the rest of the message, storing it in the file letter.sig.

Decrypting the Message and Leaving the Signature on it

Usually, you want PGP to completely unravel a ciphertext file, decrypting it and checking the nested signature if there is one, peeling away the layers until you are left with only the original plaintext file.

But sometimes you want to decrypt an encrypted file, and leave the inner signature still attached, so that you are left with a decrypted signed message. This may be useful if you want to send a copy of a signed document to a third party, perhaps re-enciphering it. For example, suppose you get a message signed by Charlie, encrypted to you. You want to decrypt it, and, leaving Charlie's signature on it, you want to send it to Alice, perhaps re-enciphering it with Alice's public key. No problem. PGP can handle that.

To simply decrypt a message and leave the signature on it intact, type:

```
pgp -d letter
```

This decrypts letter.pgp, and if there is an inner signature, it is left intact with the decrypted plaintext in the output file. Now you can archive it, or maybe re-encrypt it and send it to someone else.

Sending ASCII Text files Across Different Machine Environments

You may use PGP to encrypt any kind of plaintext file, binary 8-bit data or ASCII text. Probably the most common usage of PGP will be for E-mail, when the plaintext is ASCII text. ASCII text is sometimes represented differently on different machines. For example, on an MSDOS system, all lines of ASCII text are terminated with a carriage return followed by a linefeed. On a Unix system, all lines end with just a linefeed. On a Macintosh, all lines end with
just a carriage return. This is a sad fact of life.

Normal unencrypted ASCII text messages are often automatically translated to some common "canonical" form when they are transmitted from one machine to another. Canonical text has a carriage return and a linefeed at the end of each line of text. For example, the popular KERMIT communication protocol can convert text to canonical form when transmitting it to another
system. This gets converted back to local text line terminators by the receiving KERMIT. This makes it easy to share text files across different systems. But encrypted text cannot be automatically converted by a communication protocol, because the plaintext is hidden by encipherment. To remedy this inconvenience, PGP lets you specify that the plaintext should be treated as ASCII text (not binary data) and should be converted to canonical text form before it gets encrypted. At the receiving end, the decrypted plaintext is automatically converted back to whatever text form is appropriate for the local environment.

To make PGP assume the plaintext is text that should be converted to canonical text before encryption, just add the "t" option when encrypting or signing a message, like so:

```
pgp -et message.txt her_userid
```

This mode is automatically turned off if PGP detects that the plaintext file contains what it thinks is non-text binary data. For PGP users that use non-English 8-bit character sets, when PGP converts text to canonical
form, it may convert data from the local character set into the LATIN1 (ISO 8859-1 Latin Alphabet 1) character set, depending on the setting of the CHARSET parameter in the PGP configuration file. LATIN1 is a superset of ASCII, with extra characters added for many European languages.

Preserving the Original Plaintext Filename

Normally, PGP names the decrypted plaintext output file with a name similar to the input ciphertext filename, but dropping the extension. Or, you can override that convention by specifying an output plaintext filename on the command line with the -o option. For most E-mail, this is a reasonable way to name the plaintext file, because you get to decide its name when you decipher it, and your typical E-mail messages often come from useless original plaintext filenames like "to_phil.txt".

But when PGP encrypts a plaintext file, it always saves the original filename and attaches it to the plaintext before it compresses and encrypts the plaintext. Normally, this hidden original filename is discarded by PGP when it decrypts, but you can tell PGP you want to preserve the original plaintext filename and use it as the name of the decrypted plaintext output file. This is useful if PGP is used on files whose names are important to preserve.

To recover the original plaintext filename while decrypting, add the -p option, like so:

```
pgp -p ciphertextfile
```

I usually don't use this option, because if I did, about half of my incoming E-mail would decrypt to the same plaintext filenames of "to_phil.txt" or "prz.txt".

Using PGP as a Unix Style Filter

Unix fans are accustomed to using Unix "pipes" to make two applications work together. The output of one application can be directly fed through a pipe to be read as input to another application. For this to work, the applications must be capable of reading the raw material from "standard input" and writing the finished output to "standard output". PGP can operate in this mode. If you don't understand what this means, then you probably don't need this feature.

To use a Unix-style filter mode, reading from standard input and writing to standard output, add the -f option, like so:

```
pgp -feast her_userid <inputfile >outputfile
```

This feature makes it easier to make PGP work with electronic mail applications.

When using PGP in filter mode to decrypt a ciphertext file, you may find it useful to use the PGPPASS environmental variable to hold the pass phrase, so that you won't be prompted for it. The PGPPASS feature is explained below.

A Peek Under the Hood

Random Numbers
PGP's Conventional Encryption Algorithm
Data Compression
Message Digests and Digital Signatures

Message Digests and Digital Signatures

To create a digital signature, PGP encrypts with your secret key. But PGP doesn't actually encrypt your entire message with your secret key-- that would take too long. Instead, PGP encrypts a "message digest".

The message digest is a compact (128 bit) "distillate" of your message, similar in concept to a checksum. You can also think of it as a "fingerprint" of the message. The message digest "represents" your message, such that if the message were altered in any way, a different message digest would be computed from it. This makes it possible to detect any changes made to the message by a forger. A message digest is computed using a cryptographically strong one- way hash function of the message. It would be computationally infeasible for an attacker to devise a substitute message that would produce an identical message digest. In that respect, a message digest is much better than a checksum, because it is easy to devise a different message that would produce the same checksum. But like a checksum, you can't derive the original message from its message digest. A message digest alone is not enough to authenticate a message. The message digest algorithm is publicly known, and does not require knowledge of any secret keys to calculate. If all we did was attach a message digest to a message, then a forger could alter a message and simply attach a new message digest calculated from the new altered message. To provide real authentication, the sender has to encrypt (sign) the message digest with his secret key.

A message digest is calculated from the message by the sender. The sender's secret key is used to encrypt the message digest and an electronic timestamp, forming a digital signature, or signature certificate. The sender sends the digital signature along with the message. The receiver receives the message and the digital signature, and recovers the original message digest from the digital signature by decrypting it with the sender's public key. The receiver computes a new message digest from the message, and checks to see if it matches the one recovered from the digital signature. If it matches, then that proves the message was not altered, and it came from the sender who owns the public key used to check the signature.

A potential forger would have to either produce an altered message that produces an identical message digest (which is infeasible), or he would have to create a new digital signature from a different message digest (also infeasible, without knowing the true sender's secret key). Digital signatures prove who sent the message, and that the message was not altered either by error or design. It also provides non-repudiation, which means the sender cannot easily disavow his signature on the message.

Using message digests to form digital signatures has other advantages besides being faster than directly signing the entire actual message with the secret key. Using message digests allows signatures to be of a standard small fixed size, regardless of the size of the actual message. It also allows the software to check the message integrity automatically, in a manner similar to using checksums. And it allows signatures to be stored separately from messages, perhaps even in a public archive, without revealing sensitive information about the actual messages, because no one can derive any message content from a message digest.

The message digest algorithm used here is the MD5 Message Digest Algorithm, placed in the public domain by RSA Data Security, Inc. MD5's designer, Ronald Rivest, writes this about MD5: "It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD5 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as

is the case with any new proposal of this sort. The level of security provided by MD5 should be sufficient for implementing very high security hybrid digital signature schemes based on MD5 and the RSA public-key cryptosystem."

Data Compression

PGP normally compresses the plaintext before encrypting it. It's too late to compress it after it has been encrypted; encrypted data is incompressible. Data compression saves modem transmission time and disk space and more importantly strengthens cryptographic security. Most cryptanalysis techniques exploit redundancies found in the plaintext to crack the cipher. Data compression reduces this redundancy in the plaintext, thereby greatly enhancing resistance to cryptanalysis. It takes extra time to compress the plaintext, but from a security point of view it seems worth it, at least in my cautious opinion. Files that are too short to compress or just don't compress well are not compressed by PGP.

If you prefer, you can use PKZIP to compress the plaintext before encrypting it. PKZIP is a widely-available and effective MSDOS shareware compression utility from PKWare, Inc. Or you can use ZIP, a PKZIP-compatible freeware compression utility on Unix and other systems, available from Jean-Loup Gailly. There is some advantage in using PKZIP or ZIP in certain cases, because unlike PGP's built-in compression algorithm, PKZIP and ZIP have the nice feature of compressing multiple files into a single compressed file, which is reconstituted again into separate files when decompressed. PGP will not try to compress a plaintext file that has already been compressed. After decrypting, the recipient can decompress the plaintext with PKUNZIP. If the decrypted plaintext is a PKZIP compressed file, PGP automatically recognizes this and advises the recipient that the decrypted plaintext appears to be a PKZIP file.

For the technically curious readers, the current version of PGP uses the freeware ZIP compression routines written by Jean-loup Gailly, Mark Adler, and Richard B. Wales. This ZIP software uses functionally-equivalent compression algorithms as those used by PKWare's new PKZIP 2.0. This ZIP compression software was selected for PGP mainly because of its free portable C source code availability, and because it has a really good compression ratio, and because it's fast. Peter Gutmann has also written a nice compression utility called HPACK, available for free from many Internet FTP sites. It encrypts the compressed archives, using PGP data formats and key rings. He wanted me to mention that here.

PGP's Conventional Encryption Algorithm

As described earlier, PGP "bootstraps" into a conventional single-key encryption algorithm by using a public key algorithm to encipher the conventional session key and then switching to fast conventional cryptography. So let's talk about this conventional encryption algorithm. It isn't the DES.

The Federal Data Encryption Standard (DES) used to be a good algorithm for most commercial applications. But the Government never did trust the DES to protect its own classified data, because the DES key length is only 56 bits, short enough for a brute force attack. Also, the full 16-round DES has been attacked with some success by Biham and Shamir using differential
cryptanalysis, and by Matsui using linear cryptanalysis.

The most devastating practical attack on the DES was described at the Crypto '93 conference, where Michael Wiener of Bell Northern Research presented a paper on how to crack the DES with a special machine. He has fully designed and tested a chip that guesses 50 million DES keys per second until it finds the right one. Although he has refrained from building the real chips so far, he can get these chips manufactured for $10.50 each, and can build 57000 of them into a special machine for $1 million that can try every DES key in 7 hours, averaging a solution in 3.5 hours. $1 million can be hidden in the budget of many companies. For $10 million, it takes 21 minutes to crack, and for $100 million, just two minutes. With any major government's budget for examining DES traffic, it can be cracked in seconds. This means that straight 56-bit DES is now effectively dead for purposes of serious data security applications.

A possible successor to DES may be a variation known as "triple DES", which uses two DES keys to encrypt three times, achieving an effective key space of 112 bits. But this approach is three times slower than normal DES. A future version of PGP may support triple DES as an option. PGP does not use the DES as its conventional single-key algorithm to encrypt messages. Instead, PGP uses a different conventional single-key block encryption algorithm, called IDEA(tm).

For the cryptographically curious, the IDEA cipher has a 64-bit block size for the plaintext and the ciphertext. It uses a key size of 128 bits. It is based on the design concept of "mixing operations from different algebraic groups". It runs much faster in software than the DES. Like the DES, it can be used in cipher feedback (CFB) and cipher block chaining (CBC) modes. PGP uses it in 64-bit CFB mode.

The IPES/IDEA block cipher was developed at ETH in Zurich by James L. Massey and Xuejia Lai, and published in 1990. This is not a "home-grown" algorithm. Its designers have a distinguished reputation in the cryptologic community. Early published papers on the algorithm called it IPES (Improved Proposed Encryption Standard), but they later changed the name to IDEA (International Data Encryption Algorithm). So far, IDEA has resisted attack much better than other ciphers such as FEAL, REDOC-II, LOKI, Snefru and Khafre. And recent evidence suggests that IDEA is more resistant than the DES to Biham & Shamir's highly successful differential cryptanalysis attack. Biham and Shamir have been examining the IDEA cipher for weaknesses, without success. Academic cryptanalyst groups in Belgium, England, and Germany are also attempting to attack it, as well as the military services from several European countries. As this new cipher continues to attract attack efforts from the most formidable quarters of the cryptanalytic world, confidence in IDEA is growing with the passage of time.

Every once in a while, I get a letter from someone who has just learned the awful truth that PGP does not use pure RSA to encrypt bulk data. They are concerned that the whole package is weakened if we use a hybrid public-key and conventional scheme just to speed

things up. After all, a chain is only as strong as its weakest link. They demand an explanation for this apparent "compromise" in the strength of PGP. This may be because they have been caught up in the public's reverence and awe for the strength and mystique of RSA, mistakenly believing that RSA is intrinsically stronger than any conventional cipher. Well, it's not.

People who work in factoring research say that the workload to exhaust all the possible 128-bit keys in the IDEA cipher would equal the factoring workload to crack a 3100-bit RSA key, which is quite a bit bigger than the 1024-bit RSA key size that most people use for high security applications. Given this range of key sizes, and assuming there are no hidden weaknesses in the conventional cipher, the weak link in this hybrid approach is in the public key algorithm, not the conventional cipher. It is not ergonomically practical to use pure RSA with large keys to encrypt and decrypt long messages. A 1024-bit RSA key would decrypt messages about 4000 times slower than the IDEA cipher. Absolutely no one does it that way in the real world. Many people less experienced in cryptography do not realize that the attraction of public key cryptography is not because it is intrinsically stronger than a conventional cipher-- its appeal is because it helps you manage keys more conveniently.

Not only is RSA too slow to use on bulk data, but it even has certain weaknesses that can be exploited in some special cases of particular kinds of messages that are fed to the RSA cipher. These special cases can be avoided by using the hybrid approach of using RSA to encrypt random session keys for a conventional cipher. So the bottom line is this: Using pure RSA on bulk data is the wrong approach, period. It's too slow, it's not stronger, and may even be weaker. If you find a software application that uses pure RSA on bulk data, it probably means the implementor does not understand these issues.

Random Numbers

PGP uses a cryptographically strong pseudorandom number generator for creating temporary conventional session keys. The seed file for this is called "randseed.bin". It too can be kept in whatever directory is indicated by the PGPPATH environmental variable. If this random seed file does not exist, it is automatically created and seeded with truly random numbers derived from timing your keystroke latencies.

This generator reseeds the disk file each time it is used by mixing in new key material partially derived with the time of day and other truly random sources. It uses the conventional encryption algorithm as an engine for the random number generator. The seed file contains both random seed material and random key material to key the conventional encryption engine for the random generator. This random seed file should be at least slightly protected from disclosure, to reduce the risk of an attacker deriving your next or previous session keys. The attacker would have a very hard time getting anything useful from capturing this random seed file, because the file is cryptographically laundered before and after each use. Nonetheless, it seems prudent to at least try to keep it from falling into the wrong hands.

If you feel uneasy about trusting any algorithmically derived random number source however strong, keep in mind that you already trust the strength of the same conventional cipher to protect your messages. If it's strong enough for that, then it should be strong enough to use as a source of random numbers for temporary session keys. Note that PGP still uses truly random numbers from physical sources (mainly keyboard timings) to generate long-term public/secret key pairs.

Vulnerabilities

No data security system is impenetrable. PGP can be circumvented in a variety of ways. In any data security system, you have to ask yourself if the information you are trying to protect is more valuable to your attacker than the cost of the attack. This should lead you to protecting yourself from the cheapest attacks, while not worrying about the more expensive attacks. Some of the discussion that follows may seem unduly paranoid, but such an attitude is appropriate for a reasonable discussion of vulnerability issues:

"Not Quite Deleted" Files
Viruses and Trojan Horses
Physical Security Breach
Tempest Attacks
Protecting against Bogus Timestamps
Exposure on Multi-user Systems
Cryptanalysis
Compromised Pass Phrase and Secret Key
Public Key Tampering
Traffic Analysis

Legal Issues

Trademarks, Copyrights, and Warranties
Patent Rights on the Algorithms
Licensing and Distribution
Export Controls
Philip Zimmermann's Legal Situation

Philip Zimmermann's Legal Situation

At the time of this writing, I am the target of a US Customs criminal investigation in the Northern District of California. My defense attorney has been told by the Assistant US Attorney that the area of law of interest to the investigation has to do with the export controls on encryption software. The federal mandatory sentencing guidelines for this offense are 41 to 51 months in a federal prison. US Customs appears to be taking the position that electronic domestic publication of encryption software is the same as exporting it. The prosecutor has issued a number of federal grand jury subpoenas. It may be months before a decision is reached on whether to seek indictment. This situation may change at any time, so this description may be out of date by the time you read it. Watch the news for further developments. If I am indicted and this goes to trial, it will be a major test case.

I have a legal defense fund set up for this case. So far, no other organization is doing the fundraising for me, so I am depending on people like you to contribute directly to this cause. The fund is run by my lead defense attorney, Phil Dubois, here in Boulder. Please send your contributions to:

Philip Dubois
2305 Broadway
Boulder, Colorado 80304 USA
Phone 303-444-3885
E-mail: dubois@csn.org

You can also phone in your donation and put it on Mastercard or Visa. If you want to be really cool, you can use Internet E-mail to send in your contribution, encrypting your message with PGP so that no one can intercept your credit card number. Include in your E-mail message your Mastercard or Visa number, expiration date, name on the card, and amount of donation. Then sign it with your own key and encrypt it with Phil Dubois's public key (his key is included in the standard PGP distribution package, in the "keys.asc" file). Put a note on the subject line that this is a donation to my legal defense fund, so that Mr. Dubois will decrypt it promptly. Please don't send a lot of casual encrypted email to him -- I'd rather he use his valuable time to work on my case. If you want to read some press stories about this case, see the following references:

1) William Bulkeley, "Cipher Probe", Wall Street Journal, Thursday April 28th, 1994, front page.
2) John Cary, "Spy vs. Computer Nerd: The Fight Over Data Security", Business Week, 4 Oct 1993, page 43.
3) Jon Erickson, "Cryptography Fires Up the Feds", Dr. Dobb's Journal, December 1993, page 6.
4) John Markoff, "Federal Inquiry on Software Examines Privacy Programs", New York Times, Tuesday 21 Sep 1993, page C1.
5) Kurt Kleiner, "Punks and Privacy", Mother Jones Magazine, Jan/Feb 1994, page 17.
6) John Markoff, "Cyberspace Under Lock and Key", New York Times, Sunday 13 Feb 1994.
7) Philip Elmer-DeWitt, "Who Should Keep the Keys", Time, 14 Mar 1994, page 90.

Leaving No Traces of Plaintext on the Disk

After PGP makes a ciphertext file for you, you can have PGP automatically overwrite the plaintext file and delete it, leaving no trace of plaintext on the disk so that no one can recover it later using a disk block scanning utility. This is useful if the plaintext file contains sensitive information that you don't want to keep around. To wipe out the plaintext file after producing the ciphertext file, just add the "w" (wipe) option when encrypting or signing a message, like so:

```
pgp -esw message.txt her_userid
```

This example creates the ciphertext file "message.pgp", and the plaintext file "message.txt" is destroyed beyond recovery. Obviously, you should be careful with this option. Also note that this will not wipe out any fragments of plaintext that your word processor might have created on the disk while you were editing the message before running PGP. Most word processors create backup files, scratch files, or both. Also, it overwrites the file only once, which is enough to thwart conventional disk recovery efforts, but not enough to withstand a determined and sophisticated effort to recover the faint magnetic traces of the data using special disk recovery hardware.

Displaying Decrypted Plaintext on your Screen

To view the decrypted plaintext output on your screen (like the Unix-style "more" command), without writing it to a file, use the -m (more) option while decrypting:

```
pgp -m ciphertextfile
```

This displays the decrypted plaintext display on your screen one screenful at a time.

Making a Message For her eyes only

To specify that the recipient's decrypted plaintext will be shown ONLY on her screen and will not be saved to disk, add the -m option:

```
pgp -sem message.txt her_userid
```

Later, when the recipient decrypts the ciphertext with her secret key and pass phrase, the plaintext will be displayed on her screen but will not be saved to disk. The text will be displayed as it would if she used the Unix "more" command, one screenful at a time. If she wants to read the message again, she will have to decrypt the ciphertext again. This feature is the safest way for you to prevent your sensitive message from being inadvertently left on the recipient's disk. This feature was added at the request of a user who wanted to send intimate messages to his lover, but was afraid she might accidentally leave the decrypted messages on her husband's computer.

Note that this feature will not prevent a clever and determined person from finding a way to save the decrypted plaintext to disk-- it's to help prevent a casual user from doing it inadvertently.

Where to get PGP

The following describes how to get the freeware public key cryptographic software PGP (Pretty Good Privacy) from an anonymous FTP site on Internet, or from other sources. PGP has sophisticated key management, an RSA/conventional hybrid encryption scheme, message digests for digital signatures, data compression before encryption, and good ergonomic design. PGP is well featured and fast, and has excellent user documentation. Source code is free.

The Massachusetts Institute of Technology is the distributor of PGP version 2.6, for distribution in the USA only. It is available from "net-dist.mit.edu," a controlled FTP site that has restrictions and limitations, similar to those used by RSA Data Security, Inc., to comply with export control requirements. The software resides in the directory /pub/PGP.

A reminder: Set mode to binary or image when doing an FTP transfer. And when doing a kermit download to your PC, specify 8-bit binary mode at both ends.

There are two compressed archive files in the standard release, with the file name derived from the release version number. For PGP version 2.6, you must get pgp26.zip which contains the MSDOS binary executable and the PGP User's Guide, and you can optionally get pgp26src.zip which contains all the source code. These files can be decompressed with the MSDOS shareware archive decompression utility PKUNZIP.EXE, version 1.10 or later. For Unix users who lack an implementation of UNZIP, the source code can also be found in the compressed tar file pgp26src.tar.Z.

If you don't have any local BBS phone numbers handy, here is a BBS you might try. The Catacombs BBS, operated by Mike Johnson in Longmont, Colorado, has PGP available for download by people in the US or Canada only. The BBS phone number is 303-772-1062. Mike Johnson's voice phone number is 303 772-1773, and his email address is mpj@csn.org. Mike also has PGP available on an Internet FTP site for users in the US or Canada only; the site name is csn.org, in directory /mpj/, and you must read the README.MPJ file to get it.

To get a fully licensed version of PGP for use in the USA or Canada, contact ViaCrypt in Phoenix, Arizona. Their phone number is 602-944-0773. ViaCrypt has obtained all the necessary licenses from PKP, Ascom-Tech AG, and Philip Zimmermann to sell PGP for use in commercial or Government environments. ViaCrypt PGP is every bit as secure as the freeware PGP, and is entirely compatible in both directions with the freeware version of PGP. ViaCrypt PGP is the perfect way to get a fully licensed version of PGP into your corporate or Government environment.

Source and binary distributions of PGP are available from the Canadian Broadcasting Corporation library, which is open to the public. It has branches in Toronto, Montreal, and Vancouver. Contact Max Allen, at +1 416 205-6017 if you have questions.

Here are a few people and their email addresses or phone numbers you can contact in some countries to get information on local PGP availability for versions earlier than 2.5:

Peter Gutmann
pgut1@cs.aukuni.ac.nz
New Zealand

Hugh Kennedy
70042.710@compuserve.com
Germany

Branko Lankester
branko@hacktic.nl
+31 2159 42242
The Netherlands

Miguel Angel Gallardo
gallardo@batman.fi.upm.es
(341) 474 38 09
Spain

Hugh Miller
hmiller@lucpul.it.luc.edu
(312) 508-2727
USA

Colin Plumb
colin@nyx.cs.du.edu
Toronto, Ontario, Canada

Jean-loup Gailly
jloup@chorus.fr France

Bugs in PGP should be reported via E-mail to MIT, the official distribution site of PGP. The E-mail address for bug reports is pgp-bugs@mit.edu.

Bugs in the help file should be reported to Xanthur@aol.com

Computer Related Political Groups

PGP is a very political piece of software. It seems appropriate to mention here some computer- related activist groups. Full details on these groups, and how to join them, is provided in a separate document file in the PGP release package.

The Electronic Frontier Foundation (EFF) was founded in 1990 to assure freedom of expression in digital media, with a particular emphasis on applying the principles embodied in the US Constitution and the Bill of Rights to computer-based communication. They can be reached in Washington DC, at (202) 347-5400. Internet E-mail address: eff@eff.org.

Computer Professionals For Social Responsibility (CPSR) empowers computer professionals and computer users to advocate for the responsible use of information technology and empowers all who use computer technology to participate in public policy debates on the impacts of
computers on society. They can be reached at: 415-322-3778 in Palo Alto, E-mail address: cpsr@csli.stanford.edu.

The League for Programming Freedom (LPF) is a grass-roots organization of professors, students, businessmen, programmers and users dedicated to bringing back the freedom to write programs. They regard patents on computer algorithms as harmful to the US software industry. They can be reached at (617) 433-7071. E-mail address: lpf@uunet.uu.net.

For more details on these groups, see the accompanying document in the PGP release package.

Recommended Reading

1) Bruce Schneier, "Applied Cryptography: Protocols, Algorithms, and Source Code in C", John Wiley & Sons, 1993 (This book is a watershed work on the subject.)
2) Dorothy Denning, "Cryptography and Data Security", Addison-Wesley, Reading, MA 1982
3) Dorothy Denning, "Protecting Public Keys and Signature Keys", IEEE Computer, Feb 1983
4) Martin E. Hellman, "The Mathematics of Public-Key Cryptography,"
5) Scientific American, Aug 1979 5) Steven Levy, "Crypto Rebels", WIRED, May/Jun 1993, page 54. (This is a "must-read" article on PGP and other related topics.)
6) Ronald Rivest, "The MD5 Message Digest Algorithm", MIT Laboratory for Computer Science, 1991
7) Xuejia Lai, "On the Design and Security of Block Ciphers", ETH Series on Information Processing (Ed. J. L. Massey), Vol. 1, Hartung-Gorre Verlag, Konstanz, Switzerland, 1992
8) Philip Zimmermann, "A Proposed Standard Format for RSA Cryptosystems", Advances in Computer Security, Vol III, edited by Rein Turn, Artech House, 1988 9) Paul Wallich, "Electronic Envelopes", 9) Scientific American, Feb 1993, page 30. (This is an article on PGP)
10) William Buckeley, "Cipher Probe", Wall Street Journal, 28 April 1994, front page. (This is an article on PGP and Zimmermann)

Philip Zimmermann may be reached at:

Boulder Software Engineering 3021 Eleventh Street Boulder, Colorado 80304 USA
Internet: prz@acm.org Phone 303-541-0140 (voice) (10:00am - 7:00pm Mountain Time)
Fax line available, if you arrange it via voice line.

Jeff Sheets may be reached at:

Xanthur@aol.com

Editing Your User ID or Pass Phrase

Sometimes you may need to change your pass phrase, perhaps because someone looked over your shoulder while you typed it in. Or you may need to change your user ID, because you got married and changed your name, or maybe you changed your E-mail address. Or maybe you want to add a second or third user ID to your key, because you may be known by more than one name or E-mail address or job title. PGP lets you attach more than one user ID to your key, any one of which may be used to look up your key on the key ring. To edit your own userid or pass phrase for your secret key:

```
pgp -ke userid [keyring]
```

PGP prompts you for a new user ID or a new pass phrase.

The optional [keyring] parameter, if specified, must be a public keyring, not a secret keyring. The userid field must be your own userid, which PGP knows is yours because it appears on both your public keyring and your secret keyring. Both keyrings will be updated, even though you only specified the public keyring.

The -ke command works differently depending on whether you use it on a public or secret key. It can also be used to edit the trust parameters for a public key.

Editing the Trust Parameters for a Public Key

Sometimes you need to alter the trust parameters for a public key on your public key ring. For a discussion on what these trust parameters mean, see the section "How Does PGP Keep Track of Which Keys are Valid?" in the Essential Topics volume of the PGP User's Guide.

To edit the trust parameters for a public key:

```
pgp -ke userid [keyring]
```

The optional [keyring] parameter, if specified, must be a public keyring, not a secret keyring.

Checking if everything is OK on Your Public Key Ring

Normally, PGP automatically checks any new keys or signatures on your public key ring and updates all the trust parameters and validity scores. In theory, it keeps all the key validity status information up to date as material is added to or deleted from your public key ring. But perhaps you may want to explicitly force PGP to perform a comprehensive analysis of your public key ring, checking all the certifying signatures, checking the trust parameters, updating all the validity scores, and checking your own ultimately-trusted key against a backup copy on a write-protected floppy disk. It may be a good idea to do this hygienic maintenance periodically to make sure nothing is wrong with your public key ring. To force PGP to perform a full analysis of your public key ring, use the -kc (key ring check) command:

```
pgp -kc
```

You can also make PGP check all the signatures for just a single selected public key by:

```
pgp -kc userid [keyring]
```

For further information on how the backup copy of your own key is checked, see the description of the BAKRING parameter in the configuration file section of this manual.

Verifying a Public Key Over the Phone

If you get a public key from someone that is not certified by anyone you trust, how can you tell if it's really their key? The best way to verify an uncertified key is to verify it over some independent channel other than the one you received the key through. One convenient way to tell, if you know this person and would recognize them on the phone, is to call them and verify their key over the telephone. Rather than reading their whole tiresome (ASCII-armored) key to them over the phone, you can just read their key's "fingerprint" to them. To see this fingerprint, use the -kvc command:

```
pgp -kvc userid [keyring]
```

This will display the key with the 16-byte digest of the public key components. Read this 16-byte fingerprint to the key's owner on the phone, while she checks it against her own, using the same - kvc command at her end.

You can both verify each other's keys this way, and then you can sign each other's keys with confidence. This is a safe and convenient way to get the key trust network started for your circle of friends.

Note that sending a key fingerprint via E-mail is not the best way to verify the key, because E- mail can be intercepted and modified. It's best to use a different channel than the one that was used to send the key itself. A good combination is to send the key via E-mail, and the key fingerprint via a voice telephone conversation. Some people distribute their key fingerprint on their business cards, which looks really cool.

If you don't know me, please don't call me to verify my key over the phone-- I get too many calls like that. Since every PGP user has a copy of my public key, no one could tamper with all the copies that are out there. The discrepancy would soon be noticed by someone who checked it from more than one source, and word would soon get out on the Internet.

Handling Large Public Keyrings

PGP was originally designed for handling small personal keyrings for keeping all your friends on, like a personal rolodex. A couple hundred keys is a reasonable size for such a keyring. But as PGP has become more popular, people are now trying to add other large keyrings to their own keyring. Sometimes this involves adding thousands of keys to your keyring. PGP, in its present form, cannot perform this operation in a reasonable period of time, while you wait at your keyboard. Not for huge keyrings.

You may want to add a huge "imported" keyring to your own keyring, because you are only interested in a few dozen keys on the bigger keyring you are bringing in. If that's all you want from the other keyring, it would be more efficient if you extract the few keys you need from the big foreign keyring, and then add just these few keys to your own keyring. Use the -kx command to extract them from the foreign keyring, specifying the keyring name on the command line. Then add these extracted keys to your own keyring.

The real solution is to improve PGP to use advanced database techniques to manage large keyrings efficiently. Until this happens, you will just have to use smaller keyrings, or be patient.

Selecting Keys Via Key ID

In all commands that let the user type a user ID or fragment of a user ID to select a key, the hexadecimal key ID may be used instead. Just use the key ID, with a prefix of "0x", in place of the user ID. For example:

```
pgp -kv 0x67F7
```

This would display all keys that had 67F7 as part of their key IDs.

This feature is particularly useful if you have two different keys from the same person, with the same user ID. You can unambiguously pick which key you want by specifying the key ID.

With the BATCHMODE flag enabled on the command line, PGP will not ask any unnecessary questions or prompt for alternate filenames. Here is an example of how to set this flag:

```
pgp +batchmode cipherfile
```

This is useful for running PGP non-interactively from Unix shell scripts or MSDOS batch files. Some key management commands still need user interaction even when BATCHMODE is on, so shell scripts may need to avoid them. BATCHMODE may also be enabled to check the validity of a signature on a file. If there was no signature on the file, the exit code is 1. If it had a signature that was good, the exit code is 0.

This command-line flag makes PGP assume "yes" for the user response to the confirmation request to overwrite an existing file, or when removing a key from the keyring via the -kr command. Here is an example of how to set this flag:

pgp +force cipherfile or: pgp -kr +force Smith

This feature is useful for running PGP non-interactively from a Unix shell script or MSDOS batch file.

To facilitate running PGP in "batch" mode, such as from an MSDOS ".bat" file or from a Unix shell script, PGP returns an error exit status to the shell. An exit status code of zero means normal exit, while a nonzero exit status indicates some kind of error occurred. Different error exit conditions return different exit status codes to the shell.

Environmental Variable for Pass Phrase

Normally, PGP prompts the user to type a pass phrase whenever PGP needs a pass phrase to unlock a secret key. But it is possible to store the pass phrase in an environmental variable from your operating system's command shell. The environmental variable PGPPASS can be used to hold the pass phrase that PGP will attempt to use first. If the pass phrase stored in PGPPASS is incorrect, PGP recovers by prompting the user for the correct pass phrase. For example, on MSDOS, the shell command:

```
SET PGPPASS=zaphod beeblebrox for president
```

would eliminate the prompt for the pass phrase if the pass phrase were indeed "zaphod beeblebrox for president".

This dangerous feature makes your life more convenient if you have to regularly deal with a large number of incoming messages addressed to your secret key, by eliminating the need for you to repeatedly type in your pass phrase every time you run PGP.

I added this feature because of popular demand. However, this is a somewhat dangerous feature, because it keeps your precious pass phrase stored somewhere other than just in your brain. Even worse, if you are particularly reckless, it may even be stored on a disk on the same computer as your secret key. It would be particularly dangerous and stupid if you were to install this command in a batch or script file, such as the MSDOS AUTOEXEC.BAT file. Someone could come along on your lunch hour and steal both your secret key ring and the file containing your pass phrase.

I can't emphasize the importance of this risk enough. If you are contemplating using this feature, be sure to read the sections "Exposure on Multi-user Systems" and "How to Protect Secret Keys from Disclosure" in this volume and in the Essential Topics volume of the PGP User's Guide. If you must use this feature, the safest way to do it would be to just manually type in the shellcommand to set PGPPASS every time you boot your machine to start using PGP, and then erase it or turn off your machine when you are done. And you should definitely never do it in an environment where someone else may have access to your machine. Someone could come along and simply ask your computer to display the contents of PGPPASS.

Config.txt Commands

TMP
LANGUAGE
MYNAME
TEXTMODE
CHARSET
ARMOR
COMPRESS
COMPLETES_NEEDED
MARGINALS_NEEDED
CERT_DEPTH
PUBRING
SECRING
RANDSEED
SHOWPASS
TZFIX
CLEARSIG
VERBOSE
INTERACTIVE
ARMORLINES
KEEPBINARY
BAKRING
PAGER
NOMANUAL

Probably the simplest attack is if you leave your pass phrase for your secret key written down somewhere. If someone gets it and also gets your secret key file, they can read your messages and make signatures in your name.

Don't use obvious passwords that can be easily guessed, such as the names of your kids or spouse. If you make your pass phrase a single word, it can be easily guessed by having a computer try all the words in the dictionary until it finds your password. That's why a pass phrase is so much better than a password. A more sophisticated attacker may have his computer scan a book of famous quotations to find your pass phrase. An easy to remember but hard to guess pass phrase can be easily constructed by some creatively nonsensical sayings or very obscure literary quotes. For further details, see the section "How to Protect Secret Keys from Disclosure" in the Essential Topics volume of the PGP User's Guide.

A major vulnerability exists if public keys are tampered with. This may be the most crucially important vulnerability of a public key cryptosystem, in part because most novices don't immediately recognize it. The importance of this vulnerability, and appropriate hygienic countermeasures, are detailed in the section "How to Protect Public Keys from Tampering" in the Essential Topics volume.

To summarize: When you use someone's public key, make certain it has not been tampered with. A new public key from someone else should be trusted only if you got it directly from its owner, or if it has been signed by someone you trust. Make sure no one else can tamper with your own public key ring. Maintain physical control of both your public key ring and your secret key ring, preferably on your own personal computer rather than on a remote timesharing system. Keep a backup copy of both key rings.

"Not Quite Deleted" Files

Keep in mind that this very problem is inherent in using Microsoft Windows PGP shells due to the possibility of the swapfile containing plaintext after deleting files.

Another potential security problem is caused by how most operating systems delete files. When you encrypt a file and then delete the original plaintext file, the operating system doesn't actually physically erase the data. It merely marks those disk blocks as deleted, allowing the space to be reused later. It's sort of like discarding sensitive paper documents in the paper recycling bin instead of the paper shredder. The disk blocks still contain the original sensitive data you wanted to erase, and will probably eventually be overwritten by new data at some point in the future. If an attacker reads these deleted disk blocks soon after they have been deallocated, he could recover your plaintext.

In fact this could even happen accidentally, if for some reason something went wrong with the disk and some files were accidentally deleted or corrupted. A disk recovery program may be run to recover the damaged files, but this often means some previously deleted files are resurrected along with everything else. Your confidential files that you thought were gone forever could then reappear and be inspected by whomever is attempting to recover your damaged disk. Even while you are creating the original message with a word processor or text editor, the editor may be creating multiple temporary copies of your text on the disk, just because of its internal workings. These temporary copies of your text are deleted by the word processor when it's done, but these sensitive fragments are still on your disk somewhere.

Let me tell you a true horror story. I had a friend, married with young children, who once had a brief and not very serious affair. She wrote a letter to her lover on her word processor, and deleted the letter after she sent it. Later, after the affair was over, the floppy disk got damaged somehow and she had to recover it because it contained other important documents. She asked her husband to salvage the disk, which seemed perfectly safe because she knew she had deleted the incriminating letter. Her husband ran a commercial disk recovery software package to salvage the files. It recovered the files alright, including the deleted letter. He read it, which set off a tragic chain of events.

The only way to prevent the plaintext from reappearing is to somehow cause the deleted plaintext files to be overwritten. Unless you know for sure that all the deleted disk blocks will soon be reused, you must take positive steps to overwrite the plaintext file, and also any fragments of it on the disk left by your word processor. You can overwrite the original plaintext file after encryption by using the PGP -w (wipe) option. You can take care of any fragments of the plaintext left on the disk by using any of the disk utilities available that can overwrite all of the unused blocks on a disk. For example, the Norton Utilities for MSDOS can do this.

Even if you overwrite the plaintext data on the disk, it may still be possible for a resourceful and determined attacker to recover the data. Faint magnetic traces of the original data remain on the disk after it has been overwritten. Special sophisticated disk recovery hardware can sometimes be used to recover the data.

Viruses and Trojan Horses

Another attack could involve a specially-tailored hostile computer virus or worm that might infect PGP or your operating system. This hypothetical virus could be designed to capture your pass phrase or secret key or deciphered messages, and covertly write the captured information to a file or send it through a network to the virus's owner. Or it might alter PGP's behavior so that signatures are not properly checked. This attack is cheaper than cryptanalytic attacks.

Defending against this falls under the category of defending against viral infection generally. There are some moderately capable anti-viral products commercially available, and there are hygienic procedures to follow that can greatly reduce the chances of viral infection. A complete treatment of anti-viral and anti-worm countermeasures is beyond the scope of this document. PGP has no defenses against viruses, and assumes your own personal computer is a trustworthy execution environment. If such a virus or worm actually appeared, hopefully word would soon get around warning everyone.

Another similar attack involves someone creating a clever imitation of PGP that behaves like PGP in most respects, but doesn't work the way it's supposed to. For example, it might be deliberately crippled to not check signatures properly, allowing bogus key certificates to be accepted. This "Trojan horse" version of PGP is not hard for an attacker to create, because PGP source code is widely available, so anyone could modify the source code and produce a lobotomized zombie imitation PGP that looks real but does the bidding of its diabolical master. This Trojan horse version of PGP could then be widely circulated, claiming to be from me. How insidious.

You should make an effort to get your copy of PGP from a reliable source, whatever that means. Or perhaps from more than one independent source, and compare them with a file comparison utility.

There are other ways to check PGP for tampering, using digital signatures. If someone you trust signs the executable version of PGP, vouching for the fact that it has not been infected or tampered with, you can be reasonably sure that you have a good copy. You could use an earlier trusted version of PGP to check the signature on a later suspect version of PGP. But this will not help at all if your operating system is infected, nor will it detect if your original copy of PGP.EXE has been maliciously altered in such a way as to compromise its own ability to check signatures. This test also assumes that you have a good trusted copy of the public key that you use to check the signature on the PGP executable.

Physical Security Breach

A physical security breach may allow someone to physically acquire your plaintext files or printed messages. A determined opponent might accomplish this through burglary, trash-picking, unreasonable search and seizure, or bribery, blackmail or infiltration of your staff. Some of these attacks may be especially feasible against grassroots political organizations that depend on a largely volunteer staff. It has been widely reported in the press that the FBI's COINTELPRO program used burglary, infiltration, and illegal bugging against antiwar and civil rights groups. And look what happened at the Watergate Hotel.

Don't be lulled into a false sense of security just because you have a cryptographic tool. Cryptographic techniques protect data only while it's encrypted-- direct physical security violations can still compromise plaintext data or written or spoken information. This kind of attack is cheaper than cryptanalytic attacks on PGP.

Tempest Attacks

Another kind of attack that has been used by well-equipped opponents involves the remote detection of the electromagnetic signals from your computer. This expensive and somewhat labor-intensive attack is probably still cheaper than direct cryptanalytic attacks. An appropriately instrumented van can park near your office and remotely pick up all of your keystrokes and
messages displayed on your computer video screen. This would compromise all of your passwords, messages, etc. This attack can be thwarted by properly shielding all of your computer equipment and network cabling so that it does not emit these signals. This shielding technology is known as "Tempest", and is used by some Government agencies and defense contractors. There are hardware vendors who supply Tempest shielding commercially, although it may be subject to some kind of Government licensing. Now why do you suppose the Government would restrict access to Tempest shielding?

The best shielding is usually a grounded conductive cage.

Protecting against Bogus Timestamps

A somewhat obscure vulnerability of PGP involves dishonest users creating bogus timestamps on their own public key certificates and signatures. You can skip over this section if you are a casual user and aren't deeply into obscure public key protocols. There's nothing to stop a dishonest user from altering the date and time setting of his own system's clock, and generating his own public key certificates and signatures that appear to have been created at a different time. He can make it appear that he signed something earlier or later than he actually did, or that his public/secret key pair was created earlier or later. This may have some legal or financial benefit to him, for example by creating some kind of loophole that might allow him to repudiate a signature.

A remedy for this could involve some trustworthy Certifying Authority or notary that would create notarized signatures with a trustworthy timestamp. This might not necessarily require a centralized authority. Perhaps any trusted introducer or disinterested party could serve this function, the same way real notary publics do now. A public key certificate could be signed by the notary, and the trusted timestamp in the notary's signature would have some legal significance. The notary could enter the signed certificate into a special certificate log controlled by the notary. Anyone can read this log. The notary could also sign other people's signatures, creating a signature certificate of a signature certificate. This would serve as a witness to the signature the same way real notaries do now with paper. Again, the notary could enter the detached signature certificate (without the actual whole document that was signed) into a log controlled by the notary. The notary's signature would have a trusted timestamp, which might have greater credibility than the timestamp in the original signature. A signature becomes "legal" if it is signed and logged by the notary.

This problem of certifying signatures with notaries and trusted timestamps warrants further discussion. This can of worms will not be fully covered here now. There is a good treatment of this topic in Denning's 1983 article in IEEE Computer (see references). There is much more detail to be worked out in these various certifying schemes. This will develop further as PGP usage increases and other public key products develop their own certifying schemes.

Exposure on Multi-user Systems

PGP was originally designed for a single-user MSDOS machine under your direct physical control. I run PGP at home on my own PC, and unless someone breaks into my house or monitors my electromagnetic emissions, they probably can't see my plaintext files or secret keys.

But now PGP also runs on multi-user systems such as Unix and VAX/VMS. On multi-user systems, there are much greater risks of your plaintext or keys or passwords being exposed. The Unix system administrator or a clever intruder can read your plaintext files, or perhaps even use special software to covertly monitor your keystrokes or read what's on your screen. On a Unix system, any other user can read your environment information remotely by simply using the Unix "ps" command. Similar problems exist for MSDOS machines connected on a local area network. The actual security risk is dependent on your particular situation. Some multi-user systems may be safe because all the users are trusted, or because they have system security measures that are safe enough to withstand the attacks available to the intruders, or because there just aren't any sufficiently interested intruders. Some Unix systems are safe because they are only used by one user-- there are even some notebook computers running Unix. It would be unreasonable to simply exclude PGP from running on all Unix systems.

PGP is not designed to protect your data while it is in plaintext form on a compromised system. Nor can it prevent an intruder from using sophisticated measures to read your secret key while it is being used. You will just have to recognize these risks on multi-user systems, and adjust your expectations and behavior accordingly. Perhaps your situation is such that you should consider running PGP only on an isolated single-user system under your direct physical control. That's what I do, and that's what I recommend.

Even if the attacker cannot read the contents of your encrypted messages, he may be able to infer at least some useful information by observing where the messages come from and where they are going, the size of the messages, and the time of day the messages are sent. This is analogous to the attacker looking at your long distance phone bill to see who you called and when and for how long, even though the actual content of your calls is unknown to the attacker. This is called traffic analysis. PGP alone does not protect against traffic analysis. Solving this problem would require specialized communication protocols designed to reduce exposure to traffic analysis in your communication environment, possibly with some cryptographic assistance.

Cryptanalysis

An expensive and formidable cryptanalytic attack could possibly be mounted by someone with vast supercomputer resources, such as a Government intelligence agency. They might crack your RSA key by using some new secret factoring breakthrough. Perhaps so, but it is noteworthy that the US Government trusts the RSA algorithm enough in some cases to use it to protect its own nuclear weapons, according to Ron Rivest. And civilian academia has been intensively attacking it without success since 1978.

Perhaps the Government has some classified methods of cracking the IDEA(tm) conventional encryption algorithm used in PGP. This is every cryptographer's worst nightmare. There can be no absolute security guarantees in practical cryptographic implementations.

Still, some optimism seems justified. The IDEA algorithm's designers are among the best cryptographers in Europe. It has had extensive security analysis and peer review from some of the best cryptanalysts in the unclassified world. It appears to have some design advantages over the DES in withstanding differential cryptanalysis, which has been used to crack the DES. Besides, even if this algorithm has some subtle unknown weaknesses, PGP compresses the plaintext before encryption, which should greatly reduce those weaknesses. The computational workload to crack it is likely to be much more expensive than the value of the message.

If your situation justifies worrying about very formidable attacks of this caliber, then perhaps you should contact a data security consultant for some customized data security approaches tailored to your special needs. Boulder Software Engineering, whose address and phone are given at the
end of this document, can provide such services.

In summary, without good cryptographic protection of your data communications, it may have been practically effortless and perhaps even routine for an opponent to intercept your messages, especially those sent through a modem or E-mail system. If you use PGP and follow reasonable precautions, the attacker will have to expend far more effort and expense to violate your privacy. If you protect yourself against the simplest attacks, and you feel confident that your privacy is not going to be violated by a determined and highly resourceful attacker, then you'll probably be safe using PGP. PGP gives you Pretty Good Privacy.

Trademarks, Copyrights, and Warranties

The author assumes no liability for damages resulting from the use of this software, even if the damage results from defects in this software, and makes no representations concerning the merchantability of this software or its suitability for any specific purpose. It is provided "as is" without express or implied warranty of any kind. Because certain actions may delete files or render them unrecoverable, the author assumes no responsibility for the loss or modification of any data.

Patent Rights on the Algorithms

The RSA public key cryptosystem was developed at MIT, which holds a patent on it (U.S. patent #4,405,829, issued 20 Sep 1983). A company in California called Public Key Partners (PKP) holds the exclusive commercial license to sell and sub-license the RSA public key cryptosystem. MIT distributes a freeware version of PGP under the terms of the RSAREF license from RSA Data Security, Inc. (RSADSI).

Non-US users of earlier versions of PGP should note that the RSA patent does not apply outside the US, and at least at the time of this writing, the author is not aware of any RSA patent in any other country. Federal agencies may use the RSA algorithm, because the Government paid for the development of RSA with grants from the National Science Foundation and the Navy. But despite the fact of Government users having free access to the RSA algorithm, Government use of PGP has additional restrictions imposed by the agreement I have with ViaCrypt, as explained later. At the time of this writing (September 1994), it appears that PKP may be breaking up soon, in which case the patents they hold may fall into other hands. The RSA patent may end up with RSADSI.

I wrote my PGP software from scratch, with my own independently developed implementation of the RSA algorithm. Before publishing PGP, I got a formal written legal opinion from a patent attorney with extensive experience in software patents. I'm convinced that publishing PGP the way I did does not violate patent law.

Not only did PKP acquire the exclusive patent rights for the RSA cryptosystem, but they also acquired the exclusive rights to three other patents covering other public key schemes invented by others at Stanford University, also developed with federal funding. This essentially gives one company a legal lock in the USA on nearly all practical public key cryptosystems. They even appear to be claiming patent rights on the very concept of public key cryptography, regardless of what clever new original algorithms are independently invented by others. I find such a comprehensive monopoly troubling, because I think public key cryptography is destined to become a crucial technology in the protection of our civil liberties and privacy in our increasingly connected society. At the very least, it places these vital tools at risk by affording to the Government a single pressure point of influence.

Beginning with PGP version 2.5 (distributed by MIT, the holders of the original RSA patent), the freeware version of PGP uses the RSAREF subroutine library to perform its RSA calculations, under the RSAREF license, which allows noncommercial use in the USA. RSAREF is a subroutine package from RSA Data Security Inc, that implements the RSA algorithm. The RSAREF subroutines are used instead of PGP's original subroutines to implement the RSA functions in PGP. See the RSAREF license for terms and conditions of use of RSAREF applications.

PGP 2.5 was released by MIT for a brief test period in May, 1994 before releasing 2.6. Although 2.5 was released under the 16 March, 1994 RSAREF license, which is a perpetual license, it would be better for users in the United States to upgrade to version 2.6 to facilitate the demise of PGP 2.3a and earlier versions. Also, PGP 2.5 has bugs that are corrected in 2.6, and 2.5 will not read the new data format after September 1, 1994. (See the section on Compatibility with Previous Versions of PGP.)

The PGP 2.0 release was a joint effort of an international team of software engineers, implementing enhancements to the original PGP with design guidance from me. It was released by Branko Lankester in The Netherlands and Peter Gutmann in New Zealand, out of reach of US patent law. Although released only in Europe and New Zealand, it spontaneously spread to the USA without help from me or the PGP development team.

The IDEA(tm) conventional block cipher used by PGP is covered by a patent in Europe, held by ETH and a Swiss company called Ascom-Tech AG. The US Patent number is US005214703, and the European patent number is EP 0 482 154 B1. IDEA(tm) is a trademark of Ascom-Tech AG. There is no license fee required for noncommercial use of IDEA. Commercial users of IDEA may obtain licensing details from Dieter Profos, Ascom Tech AG, Teleservices Section, Postfach 151, 4502 Solothurn, Switzerland, Tel +41 65 242885, Fax +41 65 235761.

Ascom-Tech AG has granted permission for the freeware version PGP to use the IDEA cipher in non-commercial uses, everywhere. In the US and Canada, all commercial or Government users must obtain a licensed version from ViaCrypt, who has a license from Ascom-Tech for the IDEA cipher. Ascom-Tech has recently been changing its policies regarding the use of IDEA in PGP for commercial use outside the US, and that policy still seems to be in flux.

The ZIP compression routines in PGP come from freeware source code, with the author's permission. I'm not aware of any patents on the compression algorithms used in the ZIP routines, but you're welcome to check into that question yourself.

Licensing and Distribution

In the USA, PGP 2.6 is available from the Massachusetts Institute of Technology, under the terms of the RSAREF license. I have no objection to anyone freely using or distributing the freeware version of PGP, without payment of fees to me, as long as it is for personal non-commercial use. For commercial use, contact ViaCrypt in Phoenix, Arizona (phone 602-944-0773). You must keep the copyright, patent, and trademark notices on PGP and keep all the documentation with it.

NOTE: Regardless of the complexities and partially overlapping restrictions from all the other terms and conditions imposed by the various patent and copyright licenses (RSA, RSAREF, and IDEA) from various third parties, an additional overriding restriction on PGP usage is imposed by my own agreement with ViaCrypt: The freeware version of PGP is only for personal, noncommercial use -- all other users in the USA and Canada must obtain a fully licensed version of PGP from ViaCrypt.

I had to make an agreement with ViaCrypt in the summer of 1993 to license the exclusive commercial rights to PGP, so that there would be a legally safe way for corporations to use PGP without risk of a patent infringement lawsuit from PKP. For PGP to succeed in the long term as a viable industry standard, the legal stigma associated with the RSA patent rights had to be resolved. ViaCrypt had already obtained a patent license from PKP to make, use, and sell products that practice the RSA patents. ViaCrypt offered a way out of the patent quagmire for PGP to penetrate the corporate environment. They could sell a fully-licensed version of PGP, but only if I licensed it to them under these terms. So we entered into an agreement to do that, opening the door for PGP's future in the commercial sector, which was necessary for PGP's long-term political future.

PGP is not shareware, it's freeware. Published as a community service. Giving PGP away for free will encourage far more people to use it, which hopefully will have a greater social impact. This could lead to widespread awareness and use of the RSA public key cryptosystem.

Feel free to disseminate the complete PGP release package as widely as possible, but be careful not to violate U.S. export controls if you live in the USA. Give it to all your friends. If you have access to any electronic Bulletin Boards Systems, please upload the complete PGP executable object release package to as many BBS's as possible. The freeware version of PGP is available in source code form, and you may disseminate the source release package too, if you've got it. NOTE: Under no circumstances should PGP be distributed without the PGP documentation, including this PGP User's Guide and the RSAREF license agreement.

The PGP version 2.6 executable object release package for MSDOS contains the PGP executable software, documentation, RSAREF license, sample key rings including my own public key, and signatures for the software and this manual, all in one PKZIP compressed file called pgp26.zip. The PGP source release package for MSDOS contains all the C source files in one PKZIP compressed file called pgp26src.zip. The filename for the release package is derived from the version number of the release.

The primary release site for PGP is the Massachusetts Institute of Technology, at their FTP site "net-dist.mit.edu", in their /pub/PGP directory. You may obtain free copies or updates to PGP from this site, or any other Internet FTP site or BBS that PGP has spread to. Don't ask me for a copy directly from me, especially if you live outside the US or Canada.

After all this work I have to admit I wouldn't mind getting some fan mail for PGP, to gauge its popularity. Let me know what you think about it and how many of your friends use it. Bug reports and suggestions for enhancing PGP are welcome, too. Perhaps a future PGP

release will reflect your suggestions.

This project has not been funded and the project has nearly eaten me alive. This means you can't count on a reply to your mail, unless you only need a short written reply and you include a stamped self-addressed envelope. But I often do reply to E-mail. Please keep it in English, as my foreign language skills are weak. If you call and I'm not in, it's best to just try again later. I usually don't return long distance phone calls, unless you leave a message that I can call you collect. If you need any significant amount of my time, I am available on a paid consulting basis, and I do return those calls.

The most inconvenient mail I get is for some well-intentioned person to send me a few dollars asking me for a copy of PGP. I don't send it to them because I'd rather avoid any legal problems with PKP. Or worse, sometimes these requests are from foreign countries, and I would be risking a violation of US cryptographic export control laws. Even if there were no legal hassles involved in sending PGP to them, they usually don't send enough money to make it worth my time. I'm just not set up as a low cost low volume mail order business. I can't just ignore the request and keep the money, because they probably regard the money as a fee for me to fulfill their request. If I return the money, I might have to get in my car and drive down to the post office and buy some postage stamps, because these requests rarely include a stamped self-addressed envelope. And I have to take the time to write a polite reply that I can't do it. If I postpone the reply and set the letter down on my desk, it might be buried within minutes and won't see the light of day again for months. Multiply these minor inconveniences by the number of requests I get, and you can see the problem. Isn't it enough that the software is free? It would be nicer if people could try to get PGP from any of the myriad other sources. If you don't have a modem, ask a friend to get it for you. If you can't find it yourself, I don't mind answering a quick phone call.

If anyone wants to volunteer to improve PGP, please let me know. It could certainly use some more work. Some features were deferred to get it out the door. A number of PGP users have since donated their time to port PGP to Unix on Sun SPARCstations, to Ultrix, to VAX/VMS, to OS/2, to the Amiga, and to the Atari ST. Perhaps you can help port it to some new environments. But please let me know if you plan to port or add enhancements to PGP, to avoid duplication of effort, and to avoid starting with an obsolete version of the source code.

Because so many foreign language translations of PGP have been produced, most of them are not distributed with the regular PGP release package because it would require too much disk space. Separate language translation "kits" are available from a number of independent sources, and are sometimes available separately from the same distribution centers that carry the regular PGP release software. These kits include translated versions of the file LANGUAGE.TXT, PGP.HLP, and the PGP User's Guide. If you want to produce a translation for your own native language, contact me first to get the latest information and standard guidelines, and to find out if it's been translated to your language already. To find out where to get a foreign language kit for your language, you might check on the Internet newsgroups, or get it from Mike Johnson (mpj@csn.org).

If you have access to the Internet, watch for announcements of new releases of PGP on the Internet newsgroups "sci.crypt" and PGP's own newsgroup, "alt.security.pgp". If you want to know where to get PGP, MIT is the primary FTP distribution site (net-dist.mit.edu). Or ask Mike Johnson (mpj@csn.org) for a list of Internet FTP sites and BBS phone numbers.

Future versions of PGP may have to change the data formats for messages, signatures, keys and key rings, in order to provide important new features. This may cause backward compatibility problems with this version of PGP. Future releases may provide conversion

utilities to convert old keys, but you may have to dispose of old messages created with the old PGP.

Export Controls

The U.S. Government has made it illegal in most cases to export good cryptographic technology, and that may include PGP. They regard this kind of software just like they regard munitions. This is determined by volatile State Department, Defense Department and Commerce Department policies, not fixed laws. I will not export this software out of the US or Canada in cases when it is illegal to do so under US controls, and I urge other people not to export it on their own.

If you live outside the US or Canada, I urge you not to violate US export laws by getting any version of PGP in a way that violates those laws. Since thousands of domestic users got the first version after its initial publication, it somehow leaked out of the US and spread itself widely abroad, like dandelion seeds blowing in the wind.

Starting with PGP version 2.0 through version 2.3a, the release point of the software has been outside the US, on publicly-accessible computers in Europe. Each release was electronically sent back into the US and posted on publicly-accessible computers in the US by PGP privacy activists in foreign countries. There are some restrictions in the US regarding the import of munitions, but I'm not aware of any cases where this was ever enforced for importing cryptographic software into the US. I imagine that a legal action of that type would be quite a spectacle of controversy.

ViaCrypt PGP version 2.4 is sold in the United States and Canada and is not for export. The following language was supplied by the US Government to ViaCrypt for inclusion in the ViaCrypt PGP documentation: "PGP is export restricted by the Office of Export Administration, United States Department of Commerce and the Offices of Defense Trade Controls and Munitions Control, United States Department of State. PGP cannot be exported or reexported, directly or indirectly, (a) without all export or reexport licenses and governmental approvals required by any applicable laws, or (b) in violation of any prohibition against the export or reexport of any part of PGP." The Government may take the position that the freeware PGP versions are also subject to those controls.

The freeware PGP versions 2.5 and 2.6 were released through a posting on a controlled FTP site maintained by MIT. This site has restrictions and limitations which have been used on other FTP sites to comply with export control requirements with respect to other encryption software such as Kerberos and software from RSA Data Security, Inc. I urge you not to do anything which would weaken those controls or facilitate any improper export of ViaCrypt PGP or the freeware PGP versions.

Some foreign governments impose serious penalties on anyone inside their country for merely using encrypted communications. In some countries they might even shoot you for that. But if you live in that kind of country, perhaps you need PGP even more.

TMP - Directory Pathname for Temporary Files---Default setting: TMP = ""

The configuration parameter TMP specifies what directory to use for PGP's temporary scratch files. The best place to put them is on a RAM disk, if you have one. That speeds things up quite a bit, and increases security somewhat. If TMP is undefined, the temporary files go in the current directory. If the shell environmental variable TMP is defined, PGP instead uses that to specify where the temporary files should go.

LANGUAGE - Foreign Language Selector

Default setting: LANGUAGE = "en"

PGP displays various prompts, warning messages, and advisories to the user on the screen. For example, messages such as "File not found.", or "Please enter your pass phrase:". These messages are normally in English. But it is possible to get PGP to display its messages to the user in other languages, without having to modify the PGP executable program. A number of people in various countries have translated all of PGP's display messages, warnings, and prompts into their native languages. These hundreds of translated message strings have been placed in a special text file called "language.txt", distributed with the PGP release. The messages are stored in this file in English, Spanish, Dutch, German, French, Italian, Russian, Latvian, and Lithuanian. Other languages may be added later.

The configuration parameter LANGUAGE specifies what language to display these messages in. LANGUAGE may be set to "en" for English, "es" for Spanish, "de" for German, "nl" for Dutch, "fr" for French, "it" for Italian, "ru" for Russian, "lt3" for Lithuanian, "lv" for Latvian, "esp" for Esperanto. For example, if this line appeared in the configuration file:

    LANGUAGE = "fr"

PGP would select French as the language for its display messages. The default setting is English.

When PGP needs to display a message to the user, it looks in the "language.txt" file for the equivalent message string in the selected foreign language and displays that translated message to the user. If PGP can't find the language string file, or if the selected language is not in the file, or if that one phrase is not translated into the selected language in the file, or if that phrase is missing entirely from the file, PGP displays the message in English.

To conserve disk space, most foreign translations are not included in the standard PGP release package, but are available separately.

MYNAME - Default User ID for Making Signatures Default setting: MYNAME = ""

The configuration parameter MYNAME specifies the default user ID to use to select the secret key for making signatures. If MYNAME is not defined, the most recent secret key you installed on your secret key ring will be used. The user may also override this setting by specifying a user ID on the PGP command line with the -u option.

TEXTMODE - Assuming Plaintext is a Text File : Default setting: TEXTMODE = off

The configuration parameter TEXTMODE is equivalent to the -t command line option. If enabled, it causes PGP to assume the plaintext is a text file, not a binary file, and converts it to "canonical text" before encrypting it. Canonical text has a carriage return and a linefeed at the end of each line of text. This mode will be automatically turned off if PGP detects that the plaintext file contains what it thinks is non-text binary data. If you intend to use PGP primarily for E-mail purposes, you should turn TEXTMODE=ON.

For VAX/VMS systems, the current version of PGP defaults TEXTMODE=ON.

"Sending ASCII Text Files Across Different Machine Environments".

CHARSET - Specifies Local Character Set for Text Files

Default setting: CHARSET = NOCONV

Because PGP must process messages in many non-English languages with non-ASCII character sets, you may have a need to tell PGP what local character set your machine uses. This determines what character conversions are performed when converting plaintext files to and from canonical text format. This is only a concern if you are in a non-English non-ASCII environment.

The configuration parameter CHARSET selects the local character set. The choices are NOCONV (no conversion), LATIN1 (ISO 8859-1 Latin Alphabet 1), KOI8 (used by most Russian Unix systems), ALT_CODES (used by Russian MSDOS systems), ASCII, and CP850 (used by most western European languages on standard MSDOS PCs).

LATIN1 is the internal representation used by PGP for canonical text, so if you select LATIN1, no conversion is done. Note also that PGP treats KOI8 as LATIN1, even though it is a completely different character set (Russian), because trying to convert KOI8 to either LATIN1 or CP850 would be futile anyway. This means that setting CHARSET to NOCONV, LATIN1, or KOI8 are all equivalent to PGP.

If you use MSDOS and expect to send or receive traffic in western European languages, set CHARSET = "CP850". This will make PGP convert incoming canonical text messages from LATIN1 to CP850 after decryption. If you use the -t (textmode) option to convert to canonical text, PGP will convert your CP850 text to LATIN1 before encrypting it.
"Sending ASCII Text Files Across Different Machine Environments".

ARMOR - Enable ASCII Armor Output : Default setting: ARMOR = off

The configuration parameter ARMOR is equivalent to the -a command line option. If enabled, it causes PGP to emit ciphertext or keys in ASCII Radix-64 format suitable for transporting through E-mail channels. Output files are named with the ".asc" extension.

If you intend to use PGP primarily for E-mail purposes, you should turn ARMOR=ON.

"Sending Ciphertext Through E-mail Channels: Radix-64 Format"

ARMORLINES - Size of ASCII Armor Multipart Files : Default setting: ARMORLINES = 720

When PGP creates a very large ".asc" radix-64 file for sending ciphertext or keys through the E- mail, it breaks the file up into separate chunks small enough to send through Internet mail utilities. Normally, Internet mailers prohibit files larger than about 50000 bytes, which means that if we restrict the number of lines to about 720, we'll be well within the limit. The file chunks are named with suffixes ".as1", ".as2", ".as3", ...

The configuration parameter ARMORLINES specifies the maximum number of lines to make each chunk in a multipart ".asc" file sequence. If you set it to zero, PGP will not break up the file into chunks.

Fidonet email files usually have an upper limit of about 32K bytes, so 450 lines would beappropriate for Fidonet environments.

"Sending Ciphertext Through E-mail Channels: Radix-64 Format"

KEEPBINARY - Keep Binary Ciphertext Files After Decrypting : Default setting: KEEPBINARY = off

When PGP reads a ".asc" file, it recognizes that the file is in radix-64 format and will convert it back to binary before processing as it normally does, producing as a by-product a ".pgp" ciphertext file in binary form. After further processing to decrypt the ".pgp" file, the final output file will be in normal plaintext form. You may want to delete the binary ".pgp" intermediate file, or you may want PGP to delete it for you automatically. You can still rerun PGP on the original ".asc" file.

The configuration parameter KEEPBINARY enables or disables keeping the intermediate ".pgp" file during decryption.

Sending Ciphertext Through E-mail Channels: Radix-64

COMPRESS - Enable Compression : Default setting: COMPRESS = on

The configuration parameter COMPRESS enables or disables data compression before encryption. It is used mainly for debugging PGP. Normally, PGP attempts to compress the plaintext before it encrypts it. Generally, you should leave this alone and let PGP attempt to compress the plaintext

COMPLETES_NEEDED - Number of Completely Trusted Introducers Needed : Default setting: COMPLETES_NEEDED = 1

The configuration parameter COMPLETES_NEEDED specifies the minimum number of completely trusted introducers required to fully certify a public key on your public key ring. This gives you a way of tuning PGP's skepticism.

How Does PGP Keep Track of Which Keys are Valid?

MARGINALS_NEEDED - Number of Marginally Trusted Introducers Needed : Default setting: MARGINALS_NEEDED = 2

The configuration parameter MARGINALS_NEEDED specifies the minimum number of marginally trusted introducers required to fully certify a public key on your public key ring. This gives you a way of tuning PGP's skepticism.

How does PGP Keep Track of Which Keys are Valid?

CERT_DEPTH - How Deep May Introducers Be Nested : Default setting: CERT_DEPTH = 4

The configuration parameter CERT_DEPTH specifies how many levels deep you may nest introducers to certify other introducers to certify public keys on your public key ring. For example, If CERT_DEPTH is set to 1, there may only be one layer of introducers below your own ultimately-trusted key. If that were the case, you would be required to directly certify the public
keys of all trusted introducers on your key ring. If you set CERT_DEPTH to 0, you could have no introducers at all, and you would have to directly certify each and every key on your public key ring in order to use it. The minimum CERT_DEPTH is 0, the maximum is 8.

How Does PGP Keep Track of Which Keys are Valid?

BAKRING - Filename for Backup Secret Keyring
: Default setting: BAKRING = ""

All of the key certification that PGP does on your public key ring ultimately depends on your own ultimately-trusted public key (or keys). To detect any tampering of your public key ring, PGP must check that your own key has not been tampered with. To do this, PGP must compare your public key against a backup copy of your secret key on some tamper-resistant media, such as a write-protected floppy disk. A secret key contains all the information that your public key has, plus some secret components. This means PGP can check your public key against a backup copy of your secret key. The configuration parameter BAKRING specifies what pathname to use for PGP's trusted backup copy of your secret key ring. On MSDOS, you could set it to "a:\secring.pgp" to point it at a write-protected backup copy of your secret key ring on your floppy drive. This check is performed only when you execute the PGP -kc option to check your whole public key ring.

If BAKRING is not defined, PGP will not check your own key against any backup copy.

How to Protect Public Keys from Tampering
HowDoes PGP Keep Track of Which Keys are Valid?

PUBRING - Filename for Your Public Keyring : Default setting: PUBRING = "$PGPPATH/pubring.pgp"

You may want to keep your public keyring in a directory separate from your config.txt file in the directory specified by your $PGPPATH environmental variable. You may specify the full path and filename for your public keyring by setting the PUBRING parameter. For example, on an MSDOS system, you might want to keep your public keyring on a floppy disk by:

PUBRING = "a:pubring.pgp"

This feature is especially handy for specifying an alternative keyring on the command line.

SECRING - Filename for Your Secret Keyring : Default setting: SECRING = "$PGPPATH/secring.pgp"

You may want to keep your secret keyring in a directory separate from your config.txt file in the directory specified by your $PGPPATH environmental variable. This comes in handy for putting your secret keyring in a directory or device that is more protected than your public keyring. You may specify the full path and filename for your secret keyring by setting the SECRING parameter. For example, on an MSDOS system, you might want to keep your secret keyring on a floppy disk by:

SECRING = "a:secring.pgp"

RANDSEED - Filename for Random Number Seed : Default setting: RANDSEED = "$PGPPATH/randseed.bin"

You may want to keep your random number seed file (for generation of session keys) in a directory separate from your config.txt file in the directory specified by your $PGPPATH environmental variable. This comes in handy for putting your random number seed file in a directory or device that is more protected than your public keyring. You may specify the full path and filename for your random seed file by setting the RANDSEED parameter. For example, on an MSDOS system, you might want to keep it on a floppy disk by:

RANDSEED = "a:randseed.bin"

PAGER - Selects Shell Command to Display Plaintext Output : Default setting: PAGER = ""

PGP lets you view the decrypted plaintext output on your screen (like the Unix-style "more" command), without writing it to a file, if you use the -m (more) option while decrypting. This displays the decrypted plaintext display on your screen one screenful at a time. If you prefer to use a fancier page display utility, rather than PGP's built-in one, you can specify the name of a shell command that PGP will invoke to display your plaintext output file. The configuration parameter PAGER specifies the shell command to invoke to display the file. For example, on MSDOS systems, you might want to use the popular shareware program "list.com" to display your plaintext message. Assuming you have a copy of "list.com", you may set PAGER accordingly:

PAGER = "list"

However, if the sender specified that this file is for your eyes only, and may not be written to disk, PGP always uses its own built-in display function.

Displaying Decrypted Plaintext on Your Screen

SHOWPASS - Echo Pass Phrase to User: Default setting: SHOWPASS = off

Normally, PGP does not let you see your pass phrase as you type it in. This makes it harder for someone to look over your shoulder while you type and learn your pass phrase. But some typing-impaired people have problems typing their pass phrase without seeing what they are typing, and they may be typing in the privacy of their own homes. So they asked if PGP can be configured to let them see what they type when they type in their pass phrase.

The configuration parameter SHOWPASS enables PGP to echo your typing during pass phrase entry.

TZFIX-Timezone Adjustment

Default setting: TZFIX = 0

PGP provides timestamps for keys and signature certificates in Greenwich Mean Time (GMT), or Coordinated Universal Time (UTC), which means the same thing for our purposes. When PGP asks the system for the time of day, the system is supposed to provide it in GMT.

But sometimes, because of improperly configured MSDOS systems, the system time is returned in US Pacific Standard Time time plus 8 hours. Sounds weird, doesn't it? Perhaps because of some sort of US west-coast jingoism, MSDOS presumes local time is US Pacific time, and pre-corrects Pacific time to GMT. This adversely affects the behavior of the internal MSDOS GMT time function that PGP calls. However, if your MSDOS environmental variable TZ is already properly defined for your timezone, this corrects the misconception MSDOS has that the whole world lives on the US west coast.

The configuration parameter TZFIX specifies the number of hours to add to the system time function to get GMT, for GMT timestamps on keys and signatures. If the MSDOS environmental variable TZ is defined properly, you can leave TZFIX=0. Unix systems usually shouldn't need to worry about setting TZFIX at all. But if you are using some other obscure operating system that doesn't know about GMT, you may have to use TZFIX to adjust the system time to GMT.

On MSDOS systems that do not have TZ defined in the environment, you should make TZFIX=0 for California, -1 for Colorado, -2 for Chicago, -3 for New York, -8 for London, -9 for Amsterdam. In the summer, TZFIX should be manually decremented from these values. What a mess.

It would be much cleaner to set your MSDOS environmental variable TZ in your AUTOEXEC.BAT file, and not use the TZFIX correction. Then MSDOS gives you good GMT timestamps, and will handle daylight savings time adjustments for you. Here are some sample lines to insert into AUTOEXEC.BAT, depending on your time zone:

For Los Angeles: SET TZ=PST8PDT For Denver: SET TZ=MST7MDT For Arizona: SET TZ=MST7
(Arizona never uses daylight savings time)
For Chicago: SET TZ=CST6CDT
For New York: SET TZ=EST5EDT
For London: SET TZ=GMT0BST
For Amsterdam: SET TZ=MET-1DST
For Moscow: SET TZ=MSK-3MSD
For Aukland: SET TZ=NZT-13

CLEARSIG-Enable Signed Messages to be Encapsulated as Clear Text

Default setting: CLEARSIG = on

Normally, unencrypted PGP signed messages have a signature certificate prepended in binary form. Also, the signed message is compressed, rendering the message unreadable to casual human eyes, even though the message is not actually encrypted. To send this binary data through a 7-bit E-mail channel, radix-64 ASCII armor is applied (see the ARMOR parameter). Even if PGP didn't compress the message, the ASCII armor would still render the message unreadable to human eyes. The recipient must use PGP to strip the armor off and decompress it before reading the message.

If the original plaintext message is in text (not binary) form, there is a way to send a signed message through an E-mail channel in such a way that the signed message is not compressed and the ASCII armor is applied only to the binary signature certificate, but not to the plaintext message. The CLEARSIG flag provides this useful feature, making it possible to generate a signed message that can be read with human eyes, without the aid of PGP. Of course, you still need PGP to actually check the signature.

The CLEARSIG flag is preset to "on" beginning with PGP version 2.5. To enable the full CLEARSIG behavior, the ARMOR and TEXTMODE flags must also be turned on. Set ARMOR=ON (or use the -a option), and set TEXTMODE=ON (or use the -t option). If your config file has CLEARSIG turned off, you can turn it back on again directly on the command line,
like so:

```
pgp -sta +clearsig=on message.txt
```

This message representation is analogous to the MIC-CLEAR message type used in Internet Privacy Enhanced Mail (PEM). It is important to note that since this method only applies ASCII armor to the binary signature certificate, and not to the message text itself, there is some risk that the unarmored message may suffer some accidental molestation while en route. This can happen if it passes through some E-mail gateway that performs character set conversions, or in some cases extra spaces may be added to or stripped from the ends of lines. If this occurs, the signature will fail to verify, which may give a false indication of intentional tampering. But since PEM lives under a similar vulnerability, it seems worth having this feature despite the risks.

Beginning with PGP version 2.2, trailing blanks are ignored on each line in calculating the signature for text in CLEARSIG mode.

VERBOSE - Quiet, Normal, or Verbose Messages : Default setting: VERBOSE = 1

VERBOSE may be set to 0, 1, or 2, depending on how much detail you want to see from PGP diagnostic messages. The settings are:

0 - Display messages only if there is a problem. Unix fans wanted this "quiet mode" setting.

1 - Normal default setting. Displays a reasonable amount of detail in diagnostic or advisory messages.

2 - Displays maximum information, usually to help diagnose problems in PGP. Not recommended for normal use. Besides, PGP doesn't have any problems, right?

INTERACTIVE - Ask for Confirmation for Key Adds : Default Setting: INTERACTIVE = off

Enabling this mode will mean that if you add a key file containing multiple keys to your key ring, PGP will ask for confirmation for each key before adding it to your key ring.

NOMANUAL - Let PGP Generate Keys Without the Manual : Default Setting: NOMANUAL = off

It is important that the freeware version of PGP not be distributed without the user documentation, which normally comes with it in the standard release package. This manual contains important information for using PGP, as well as important legal notices. But some people have distributed previous versions of PGP without the manual, causing a lot of problems for a lot of people who get it. To discourage the distribution of PGP without the required documentation, PGP has been changed to require the PGP User's Guide to be found somewhere on your computer (like in your PGP directory) before PGP will let you generate a key pair. However, some users like to use PGP on tiny palmtop computers with limited storage capacity, so they like to run PGP without the documentation present on their systems. To satisfy these users, PGP can be made to relax its requirement that the manual be present, by enabling the NOMANUAL flag on the command line during key generation, like so:

pgp -kg +nomanual

The NOMANUAL flag can only be set on the command line, not in the config file. Since you must read this manual to learn how to do enable this override feature, I hope this will still be effective in discouraging the distribution of PGP without the manual.

Some people may object to PGP insisting on finding the manual somewhere in the neighborhood to generate a key. They bristle against this seemingly authoritarian attitude. Some people have even modified PGP to defeat this feature, and redistributed their hotwired version to others. That creates problems for me. Before I added this feature, there were maimed versions of the PGP distribution package floating around that lacked the manual. One of them was uploaded to Compuserve, and was distributed to countless users who called me on the phone to ask me why such a complicated program had no manual. It spread out to BBS systems around the country. And a freeware distributor got hold of the package from Compuserve and enshrined it on CD-ROM, distributing thousands of copies without the manual. What a mess.

Whats New in Each New Version

2.2
2.3 (including 2.3a)
2.4
2.5
2.6 (also known as 2.6MIT)
2.6ui (NOT approved by Philip Zimmermann)
2.62
2.62i (NOT an official version)

Changes to PGP 2.6

This version of PGP uses a version of RSAREF provided to MIT by RSA Data Security for use in PGP. This version is legal within the U.S. See the enclosed RSAREF license for full details. Basically this is a non-commercial release. If you want to use it in a commercial or governmental setting, talk to ViaCrypt (2014 West Peoria Avenue, Phoenix, Arizona 85029, +1 602 944-0773).

PGP 2.6 will read messages, signatures and keys created with versions of PGP post 2.2. (i.e., 2.3, 2.3a, 2.4 and 2.5). However after 9/1/94 Version 2.6 will create messages which contain a version number of "3" in signatues, messages and keys (see pgformat.doc for details). PGP2.6 will be able to read these signatures, messages and keys, but prior versions will not.

Versions prior to 2.6 would not permit a new signature to be added to a key if there was an already existing signature from the same signer. Starting with version 2.6 newer signatures will override older ones *as long as the newer signature verifies*. This change is important because many keys have
signatures on them that were created by PGP version 2.2 or earlier. These signatures can not be verified by PGP 2.5 or higher. Owners of keys with these obsolete signatures should attempt to gather new signatures and add them to their key.

Significant changes were also made for version 2.5. Because version 2.6 is coming out very soon after 2.5 (which was only really a beta test version) readers are encouraged to read the file "newfor25.doc" as well as this file.

Changes to PGP 2.5

***** MOST IMPORTANT *****

This version of PGP uses RSAREF 2.0, so it's legal in the U.S.! The RSAREF license forbids you to (among other things; see the license for full details) "use the program to provide services to others for which you are compensated in any manner", but that still covers a lot of people. If you want to use it in a commercial or governmental setting, talk to ViaCrypt (2014 West Peoria Avenue, Phoenix, Arizona
85029, +1 602 944-0773).

PGP 2.5 should always be distributed with a copy of the RSAREF 2.0 license of March 16, 1994 from RSA Data Security, Inc., so that all users will be aware of their obligations under the RSAREF license.

Since the RSAREF license conflicts with the GNU General Public License that PGP was formerly distributed under, the GPL had to go. PGP is still freely distributable, though. (From a copyright point of view; export controls or some other legal hassle may apply.)

*** IMPORTANT CHANGE:

RSAREF 2.0 can understand only the pkcs_compat=1 formats for signatures and encrypted files. This has been the default since 2.3, so old files should not be too much of a problem, but old key signatures will encounter difficulties. This change will result in a hole being ripped in the "web of trust" as many old signatures are invalidated. Please check your key rings (pgp -kc) and re-issue any signatures that have been invalidated. PGP by default offers to remove such signatures. Even if you leave them in, they are not trusted.

Another RSAREF limitation is that it cannot cope with keys longer than 1024 bits. PGP now prints a reasonably polite error message in such a case.

OTHER CHANGES:

The support files are thinner. The various contrib directory utilities have not been updated since 2.3a, and since the PGP developers know how annoying it is to have people using an ancient version and complaining about a bug in a program that was fixed a year ago, they have been omitted rather than annoy the contributors in this way. Also, the language translation file, language.txt, is incomplete. The strings that were in 2.3a are there, and some that could be updated without much knowledge of the language, but others that are new to 2.5 are untranslated. The format should be obvious and some tools for manipulating the language traslations are included in the contrib directory.

Printed KeyIDs have been incresed to 32 bits, as there were enough keys out there that 24-bit keyIDs were no longer sufficiently unique. The previous 24-bit keyID is the LAST 6 digits of an 8-digit 32-bit keyID. For example, what was printed as A966DD now appears as C7A966DD.

The config-file options
pubring=<filename>,
secring=<filename>, and
randseed=<filename>
have been added. Hopefully, the uses will be obvious. With these, you can keep keyrings anywhere you like. Of course, they can also be specified on the command line with +pubring= (or abbreviated to +pub=). If the line
comment=<string>
appears in the config file, the line "Comment: <string>" appears in ASCII armor output.

Of course, you can also use this from the command line, e.g. to include a filename in the ASCII armor, do "pgp -eat +comment=filename filename recipient".

PGP now enables clearsig by default. If you sign and ascii-armor a text file, and do not encrypt it, it is clearsigned unless you ask for this not to be done.

The now enables textmode. Textmode detects non-text files and automatically turns itself off, so it's quite safe to leave on all the time. If you haven't got these defaults yourself, you might want to enable them.

All prompts and progress messages are now printed to stderr, to make them easier to find and ensure they don't get confused with data on standard output such as pgp -m output.

PGP now wipes temp files (and files wiped with pgp -w) with pseudo-random data in an attempt to force disk compressors to overwrite as much data as possible.

On Unix, if the directory /usr/local/lib/pgp exists, it is searched for help files, language translations, and the PGP documentation. On VMS, the equivalent is PGP$LIBRARY:. (This is PGP_SYSTEM_DIR, defined in fileio.h, if you need to change it for your site.)

Also, it is searched for a default global config.txt. This file may be overridden by a local config.txt, and it may not set pubring, secring, randseed or myname (which should be strictly personal)

The normal help files (pgp -h) are pgp.hlp or <language>.hlp, such as fr.hlp. Now, there is a separate help file for pgp -k, called pgpkey.hlp, or <language>key.hlp. No file is provided by default; PGP will use its one-page internal help by default, but you can create such a file at your site.

On Unix systems, $PGPPATH defaults to $HOME/.pgp.

PGP used to get confused if you had a keyring containing signatures from you, but not your public key. (PGP can't use the signatures in this case. Only signatures from keys in the keyring are counted.)
PGP still can't use the signatures, but prints better warning messages. Also, adding a key on your secret key ring to your public keyring now asks if the key should be considered ultimately-trusted.
Prviously, you had to run pgp -ke to force this check, which was non-obvious.

Due to a few people distributing PGP without the manual (including one run of a few thousand CD-ROMs), and the resultant flood of phone calls from confused users, PGP now looks to make sure a manual is somewhere in the vicinity when running to discourage this sort of thing. (If you're getting this warning and need details on how to get rid of it, try pgp -kg.)

On Unix, PGP now figures out the resolution of the system clock at run time for the purpose of computing the amount of entropy in keystroke timings. This means that on many Unix machines, less typing should be required to generate keys. (SunOS and Linux especially.)

The small prime table used in generating keys has been enlarged, which should speed up key generation somewhat.

There was a bug in PGP 2.3a (and, in fact in 2.4 and dating back to 1.0!) when generating primes 2 bits over a multiple of the unit size (16 bits on PC's, 32 bits on most larger

computers), if the processor doesn't deal with expressions like "1<<32" by producing a result of 1. In practice, that corresponds to a key size of 64*x+4 bits.

Code changes:

At the request of Windows programmers, the PSTR() macro used to translate string has been renamed to LANG().

The random-number code has been *thoroughly* cleaned up. So has the IDEA code and the MD5 code. The MD5 code was developed from scratch and is available for public use.

The Turbo C makefile was dropped in favour of a Borland C .prj file. You can use makefile.msc as a guide if you need one for a command-line Turbo C.

Changes to PGP 2.4:

- Fixed a bug with the -z <passphrase> option. If no passphrase was given, PGP used to crash.

- When using -c, the IV is generated properly now, and the randseed.bin postwash is done. (This bug could have resulted in the same ciphertext being generated for the same plaintext, if the same passphrase is used.)

- Memory allocated with halloc() is now freed with hfree() in ztrees.c and zdeflate.c. (MS-DOS only.)

- The decompression code now detects end of input reliably, fixing a bug that used to have it produce infinite amounts of output on come corrputed input. Decompression has also been sped up.

- PGP -m won't try to write its final output to the current directory. This makes it less efficent if you want to save the text to a file, but more secure if you don't.

- Number of bits allowed when generating keys limited to 1024, in line with the limits in RSAREF and BSAFE. It used to be higher, but folks, if you think you need a key larger than that, do some research
into the complexity of factoring.

- Version number changed to pgp2.4

News for PGP 2.3a

There was a bug in PGP's handling of clear-signed messages when lines were terminated with CR-LF pairs. This has been revamped. The previous limit on the length of lines in clear-signed messages has been eliminated.

The randseed.bin file was not closed when read, which resulted in it not being rewritten with a new value under some operating systems. Fixed.

Not all of the bytes in randseed.bin were being used, resulting in less randomness than desired when picking session keys. While it did not make the compromise of session keys likely, it was undesirable and has been fixed.

PGP should now compile with less difficulty under OS/2. The Turbo C makefile was incorrect. Fixed.
The VMS build files were out of date. Fixed.

PGP was not accepting octal escapes in the language.txt file that did not begin with \0. \ 377 is now acceptable. The language.txt file got mangled in the middle somehow. Fixed.

News for PGP 2.3

This PGP 2.3 release has several bug fixes over PGP 2.2, and a few new (although somewhat esoteric) features. Among them are:

- An important bug: there was a bug with compression under MS-DOS which caused the wrong piece of memory to be freed, with results that ranged from none to undecodable messages to machine crashes.

- When adding keys, PGP now properly closes all the files it opens, so you don't run out of file handles (MS-DOS) or file descriptors (UNIX).

- Sometimes PGP would not properly ask the user to set trust parameters when keys were validated by adding new signatures. This has been fixed.

- When PGP messages are sent through a MIME mail system, a conflict arises over the use of the '=' character. PGP can now decode ASCII armored messages which have been mangled by MIME's quoting mechanism.

- PGP previously kept track of one pass phrase (from the PGPPASS environment variable, the file descriptor named by the PGPPASSFD environment variable, a -z <password> option, or previous user
prompts), and tried it if it needed a subsequent pass phrase. This caused bugs if you attempted something that required two pass phrases, such as pgp -sc (sign and conventionally encrypt). PGP now keeps track of any number of pass phrases, including multiple -z options, and uses them as necessary. Mostly, it just Does The Right Thing, but if you care, the exact algorithm is as follows:

- There is a pool of private-key pass phrases that starts out with the contents of the PGPPASS environment variable (if any), and has every pass phrase that is successfully used to unlock a private key added to it. When a private key needs unlocking, every pass phrase in the
pool is tried first.

- There is a list of PGP pass phrases available for use by whatever needs one. This is initialized with the -z command-line options and the phrase read from the PGPPASSFD file

descriptor. When a pass phrase is needed, it is taken from the front of that list. When a pass phrase is needed to unlock a secret key, every key on the list is tried, and if it "fits" and unlocks the secret key, it is moved to the key pass phrase pool.

- If the above fails to produce a pass phrase, the user is prompted to supply one.

Key generation (we need all the keystrokes we can get for random-number accumulation) and key signing (to make sure the user really means to do what they're doing) are exceptions; the user is always prompted for a pass phrase under those circumstances.

New options:

+pkcs_compat=n
This defaults to 1, which tells PGP to generate encryption key and signature blocks in a format derived from the PKCS standards. This format is understood (but not generated) by PGP 2.2. If set to 0, the old format is generated, which may be needed for portability to PGP versions before 2.2. PGP is still incompatible with the PKCS standards in many ways, but in future, values of 2 or higher may be used to produce formats which are more compatible.

Other notes:

The MS-DOS executable was compiled with Borland C++ version 3.0, optimized for maximum speed, except that jump optimisation was turned off. If it is turned on, the Transform() function in md5.c is compiled incorrectly. The pgp.prj file that was used is included in the source distribution.

Thanks to everyone who worked on PGP and sent in bug reports. Two who didn't make it into the manual are to Lindsay DuBois for a bit of last- minute translation, and Reptilian Research for support in developing PGP.

And thanks to the Cypherpunks who managed to get PGP so much attention in Wired magazine recently.

I hope you enjoy PGP!

-Colin <colin@nyx.cs.du.edu>

News for PGP 2.2

The main change since PGP 2.1 is a speedup in key management, and the ability to encrypt for more than one recipient. Apart from this there are some bugfixes and some new options to make it easier
to use PGP from shell scripts or mailers.

You can encrypt for more than one recipient by specifying additional userids on the command line eg:

```
pgp -e plaintext Alice Bob Carol
```

Some notes about the changes:

- PGP doesn't do a keycheck on a keyfile before it is added anymore, this is to speed up merging a big keyfile with your public keyring which may already have most of the keys in the keyfile you are adding. After PGP has checked a signature it sets a flag in your public keyring to mark this signature as checked. Because PGP 2.1 didn't have these flags, PGP will check *all* signatures on your keyring the first time
you add a key with PGP 2.2. After that PGP will only check new signatures. Also by using an older version than 2.2 on your keyring you will clear these flags again.

New options:

+interactive
: If you add a keyfile, PGP will ask for each new key if it should be added to your keyring.

Options for use in shell scripts:

+verbose=n
: The default is 1. With +verbose=0, PGP will only print an error message if something goes wrong. With +verbose=2, PGP will tell you what it's doing in detail suitable for debugging.

+force
: Overwrite output file without asking, or with -kr: remove key without asking (only if it has just one userid).

+batchmode
: With this option PGP won't ask any questions or prompt for alternate file names. Some of the key commands still need user interaction and can't be done from a shell script. You can also use this option to check if a file has a good signature. If the input file did not have a signature the exit code will be 1, if the file had a signature and if it checked OK the exit code will be 0. Note that if the input file has more than one armored messages, a good signature on one of these messages will make the exit code 0 (if there are no errors).

These "long" options can be abbreviated as long as the abbreviation is unambiguous. "interactive" and "verbose" can also be set in config.txt; you can then turn these flags off on the command line with +option=.

Some of the bug fixes:

- Key lookup on keyID (eg 0x12AB) fixed for -ks/-krs.
- Dearmoring of Macintosh type text files (CR only) now also works.

PGP 2.6ui (NOT approved by Philip Zimmermann)

>>> THIS IS AN UNOFFICIAL RELEASE OF PGP <<<
MIT have released what they call version 2.6 of PGP. Unlike PGP 2.3a, it uses the RSAREF library. This has a number of ramifications:

1. Key sizes are limited to 1024 bits.
2. You must agree to the RSAREF license.
3. MIT PGP 2.6 is not publicly and widely available outside the USA and Canada, because RSAREF falls under ITAR export restrictions.

Worse, MIT's PGP 2.6 has been deliberately crippled, so that after September 1st 1994 it will produce encoded data which PGP 2.3a cannot decode.

This is an unofficial release of PGP based on 2.3a, modified for interoperability with MIT's PGP 2.6. The following changes have been made:

* This version of PGP will read encoded data produced by both MIT PGP 2.6 and PGP 2.3a.

* You can choose to write data either in the "new" format used by MIT PGP 2.6, or in the old PGP 2.3a format. To do this, use the command line switch +version_byte=3 for MIT PGP 2.6 format, or +version_byte=2 for PGP 2.3a format (the default).

You can also specify a line like:

version_byte = 3

in your config.txt file.

This version does *not* have any time-bomb code in it. If you want to switch version byte like MIT PGP does, you'll have to do it manually on September 1st. There's no advantage in doing so, unless you want it to look like you're running MIT PGP.

* You can choose the version text which you want to have appear in ASCII armoured files. The default is 2.6, and if you're in the USA you probably don't want to change it, as a well known net.personality tends to harass people whose PGP armor says anything else.

To change the version text, use a command line argument such as +armor_version=2.6ui or +armor_version=2.3 -- but please do try and use this feature responsibly, and don't go creating random version strings unnecessarily.

You can also specify

armor_version = 2.6ui --- or similar in your config.txt file.

* This version of PGP displays and accepts 8 characters of the key ID. Hence there's less chance of two keys having the same visible ID.

* Makefile entries have been added for sunos5cc and sunos5gcc, for people using SPARC workstations running Solaris 2. I have personally tested the sunos5gcc build on Solaris 2.3, and it compiles cleanly. A line for NeXTstep Intel has also been added (next486).

* The file idea68k.s has been removed, at the request of the author. It was obsolete. Better 68k routines are available; for example, suitable routines for the Amiga are available on Aminet.

* A message has been added to the key generation section, reminding the user that MIT PGP 2.6 will only handle keys of 504-1024 bits. This version has no key length crippling, however.

* Memory allocated with _fcalloc and freed with _ffree in ztrees.c and zdeflate.c, to avoid memory leakage in the MS-DOS version.

* The -w option wipes files with pseudo-random data, to try and ensure the file is wiped even if you're using a disk compressor. Note that this still isn't perfect; DOS can randomly duplicate bits of cleartext files in partially-used clusters, and those fragments won't be removed if the original file is wiped.

* Branko Lankester and Paul C Leyland's patches have been applied, so that newer key certification signatures automatically replace older ones.

This release reports itself as 2.6ui. It's 2.6 because if it were called 2.3b, users would get confused as to whether 2.3b was compatible with MIT PGP 2.6, given the much smaller version number. I haven't
called this version 2.7, because I have no wish to get involved in the kind of "version number war" which goes on between (say) Microsoft, IBM and Novell over DOS.

The letters "ui" stand for Unofficial International release. It's an unofficial release in that it has NOT been approved by Philip Zimmermann.

This version was assembled by mathew <mathew@mantis.co.uk> from the standard PGP 2.3a sources, and from source code patches obtained from the net. All patches were scrutinized carefully before being applied by hand. No binary patches were used. The DOS executables were built by mathew using Microsoft Visual C++ version 1.0 (MS C v8).

No RSAREF source code was used; in fact, I used no source code from MIT PGP at all. I haven't even looked at the MIT sources. (No, really.)

Thanks go to Alan Barrett <barrett@ee.und.ac.za>, Planar <Damien.Doligez@inria.fr> and Pr0duct Cypher for 2.6 information and interoperability patches. Philip C. Kizer <pckizer@gonzo.tamu.edu>
provided the Solaris 2 patches. The key signature patches were by Paul C Leyland <pcl@black.ox.ac.uk> and Branko Lancaster <branko@suppnet1.support.nl>. Thanks also to those who tested 2.6ui against MIT 2.6; you'd better remain anonymous, but you know who you are...

Disclaimer: This software is nothing to do with Mantis Consultants, and is without warranty or guarantee of any kind. Using it in the USA is probably very naughty.

If you used this version of PGP in the USA, and if you remembered to change version_byte to 3 on September 1st 1994, your PGP messages would almost certainly be indistinguishable from ones produced using MIT PGP 2.6. That would be very naughty and devious, though, so don't do it.

My key is on the usual key servers; detached signatures are available for the zip and tar files. Note that if the signature matches, that only really means that the files you have are the same as the ones I have; it doesn't mean that I guarantee this release of PGP works, or that I've examined it for cryptographic back doors.

Naturally, you should only use this release of PGP if you think you can trust me, or if you've compared the source with 2.3a and compiled it yourself. I've created some md5

values in contrib/md5sum. The old pgp23a ones are there, so you can detect which files I've changed :-)

If you have any patches to fix bugs or add features, feel free to mail them to me, and I'll consider adding them to any future unofficial release.

mathew
<mathew@mantis.co.uk>

-----BEGIN PGP SIGNATURE-----
Version: 2.6

iQCVAgUBLeXrEnzXN+VrOblFAQGNJgQAmgZEiop3zrMIxvx9Tg4dHRdG28wtay7l
RO3bMpDl0FiVB+KL/56zmbt1p4kDpiyxCoalyY0br9URl8aoS+JRzs7yqmxE7BC0
Z49x18ednTv8rQiAH0pPXHiJJs7Ds2Ea74kLmWZvDyjTZWJD+bMUlPjVEc0IH/9u
jjgMjJ+H3uU=
=Vdz2
-----END PGP SIGNATURE-----

Changes for PGP 2.6.2

Some people reported a "bug" that you could stick an extra paragraph in the beginning of a clear-signed message and PGP would still report a good signature. PGP allows comment "headers" before ASCII-armor blocks (like the Version: header that's there for debugging purposes), terminated, as with e-mail and usenet messages, by a blank line. These headers are just window dressing; PGP ignores them. So this is actually a "feature"; the bug is that people think it's part of the signed message. There are a number of ways to fake the visual appearance of a blank line using common file-viewing utilities, a blank line is easy to miss even if you know about it, and headers are not presently used in clear-signed messages. So now headers are forbidden at the beginning of a clear-signed message. Also, PGP enforces an RFC-822-like syntax on header lines before ASCII armor. Note that in no case has PGP's output ever been compromised; the problem arises only if people see the "good signature" message but try to read the input directly to see what was signed.

- Fixed MAX_BIT_PRECISION to 2048.

- Fixed a bug in key editing with the public and secret keyrings in different directories. The old code was assuming they were in the same directory, which used to be a safe assumption, but no more.

PGP 2.6 will read messages, signatures and keys created with versions of PGP post 2.2. (i.e., 2.3, 2.3a, 2.4 and 2.5). However after 9/1/94 Version 2.6 will create messages which contain a version number of "3" in signatues, messages and keys (see pgformat.doc for details). PGP2.6 will be able to read these signatures, messages and keys, but prior versions will not.

Versions prior to 2.6 would not permit a new signature to be added to a key if there was an already existing signature from the same signer. Starting with version 2.6 newer signatures will override older ones *as long as the newer signature verifies*. This change is important because many keys have signatures on them that were created by PGP version 2.2 or earlier. These signatures can not be verified by PGP 2.5 or higher. Owners of keys with these obsolete signatures should attempt to gather new signatures and add them to their key.

The config-file options

pubring=<filename>,
secring=<filename>, and
randseed=<filename>

have been added. Hopefully, the uses will be obvious. With these, you can keep keyrings anywhere you like. Of course, they can also be specified on the command line with +pubring= (or abbreviated to +pub=)

Using PGP as a Better Uuencode

A lot of people in the Unix world send binary data files through E-mail channels by using the Unix "uuencode" utility to convert the file into printable ASCII characters that can be sent via email. No encryption is involved, so neither the sender nor the recipient need any special keys. The uuencode format was designed for a similar purpose as PGP's radix-64 ASCII transport armor format described in the "Sending Ciphertext Through E-mail Channels: Radix-64 Format" section, but not as good. A different radix-64 character set is used. Uuencode has its problems, such as 1) several slightly incompatible character sets for different versions of uuencode in the MSDOS and Unix worlds, and 2) the data can be corrupted by some E-mail gateways that strip trailing blanks or do other modifications to the character set used by uuencode.

PGP may be used in a manner that offers the same general features as uuencode, and then some. You can get PGP to just convert a file into PGP's radix-64 ASCII transport armor format, but you don't have to encrypt the file or sign it, so no keys are needed by either party. Simply use the -a option alone. For example:

```
pgp -a filename
```

This would produce a radix-64 armored file called "filename.asc".

If you read the
"Sending Ciphertext Through E-mail Channels: Radix-64 Format"
section, you will see that PGP's approach offers several important advantages over the uuencode approach:

* PGP will break big files up into chunks small enough to E-mail.
* PGP will append a CRC error detection code to each chunk.
* PGP will attempt to compress the data before converting it to radix-64 armor.
* PGP's radix-64 character set is more resilient to E-mail character conversions than the one used by uuencode.
* Textfiles can be converted by the sender to canonical text format, as explained in the "Sending ASCII Text Files Across Different Machine Environments" section.

The recipient can restore the sender's original filename by unwrapping the message with PGP's -p option. You can use "pgp -a" in any situation in which you could have used uuencode, if the recipient also has PGP. PGP is a better uuencode than uuencode.

File Formats Used by PGP 2.6 (22 May 94)

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

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

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

Byte Order
Multiprecision Integers
Key ID
User ID
Timestamp
Cipher Type Byte (CTB)
RSA public-key-encrypted packet
Signature packet
Message digest "packet"
Conventional Data Encryption Key (DEK) "packet"
Conventional Key Encrypted data packet
Compressed data packet
Literal data packet, with filename and mode
Comment packet
Secret key certificate
Public key certificate
User ID packet
Keyring trust packet
Public Key Ring Overall Structure

All integer data used by PGP is externally stored most significant byte (MSB) first, regardless of the byte order used internally by the host CPU architecture. This is for portability of messages and keys between hosts. This covers multiprecision RSA integers, bit count prefix fields, byte count prefix fields, checksums, key IDs, and timestamps. The MSB-first byte order for external packet representation was chosen only because many other crypto standards use it.

RSA arithmetic involves a lot of multiprecision integers, often having hundreds of bits of precision. PGP externally stores a multiprecision integer (MPI) with a 16-bit prefix that gives the number of significant bits in the integer that follows. The integer that follows this bitcount field is stored in the usual byte order, with the MSB padded with zero bits if the bitcount is not a multiple of 8. The bitcount always specifies the exact number of significant bits. For example, the integer value 5 would be stored as these three bytes: 00 03 05 An MPI with a value of zero is simply stored with the 16-bit bitcount prefix field containing a 0, with no value bytes following it.

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

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

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

Cipher Type Byte (CTB)

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

Bit 7: Always 1, which designates this as a CTB Bit 6: Reserved. Bits 5-2: CTB type field, specifies type of packet that follows

0001 - public-key-encrypted packet
0010 - secret-key-encrypted (signature) packet
0101 - Secret key certificate
0110 - Public key certificate
1000 - Compressed data packet
1001 - Conventional-Key-Encrypted data
1011 - Raw literal plaintext data, with filename and mode
1100 - Keyring trust packet
1101 - User ID packet, associated with public or secret key
1110 - Comment packet
Other CTB packet types are unimplemented.

Bits 1-0: Length-of-length field:

00 - 1 byte packet length field follows CTB
01 - 2 byte packet length field follows CTB
10 - 4 byte packet length field follows CTB
11 - no length field follows CTB, unknown packet length.

The 8-, 16-, or 32-bit packet length field after the CTB gives the length in bytes of the rest of the packet, not counting the CTB and the packet length field.

RSA public-key-encrypted packet

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

Signature packet

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

If the plaintext that was signed is included in the same file as the signature packet, it begins right after the RSA secret-key-signed packet that contains the message digest. The plaintext has a "literal" CTB prefix. The original idea had a variable length field following the length of following material byte, before the Key ID. In particular, the possibility of a 2-byte validity period was defined, although no previous version of PGP ever generated those bytes. Owing to the way the MD5 is computed for the signature, if that field is variable length, it is possible to generate two different messages with the same MD5 hash. One would be a file of length N, with a 7-byte following section consisting of a signature type byte, 4 bytes of timestamp, and 2 of validity period, while the other would be a file of length N+2, whose last two bytes would be the siganture type byte and the first byte of timestamp, and the last three bytes of timestamp and the validity period would instead be interpreted as a signature type byte and a timestmap. It should be emphasized that the messages are barely different and special circumstances must arise for this to be possible, so it is extremely unlilely that this would be exploitable, but it is a potential weakness. It has been plugged by allowing only the currently implemented 5-byte option. Validity periods will be added later with a different format. The signature classification field describes what kind of signature certificate this is. There are various hex values:

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

certificate. Independent of user ID associations. Material signed is public key pkt ONLY.

30 - Key/userid revocation. User can sign his own revocation to dissolve an association between a key and a user ID, or certifier may revoke his previous certification of this key/userid pair. Material signed is public key pkt and User ID pkt.

40 - Timestamping a signature certificate made by someone else. Can be used to apply trusted timestamp, and log it in notary's log. Signature of a signature. (Planned, not implemented.)

When a signature is made to certify a key/UserID pair, it is computed across two packets_ the public key packet, and the separate User ID packet. See below. The packet headers (CTB and length fields) for the public key packet and the user ID packet are both omitted from the signature calculation for a key certification. A key compromise certificate may be issued by someone to revoke his own key when his secret key is known to be compromised. If that happens, a user would sign his own key compromise certificate with the very key that is being revoked. A key revoked by its own signature means that this key should never be used or trusted again, in any form, associated with any user ID. A key compromise certificate issued by the keyholder shall take precedence over any other key certifications made by anyone else for that key. A key compromise signed by someone other than the key holder is invalid. Note that a key compromise certificate just includes the key packet in its signature calculation, because it kills the whole key without regard to any userid associations. It isn't tied to any particular userid association. It should be inserted after the key packet, before the first userid packet. When a key compromise certificate is submitted to PGP, PGP will place it on the public keyring. A key compromise certificate is always accompanied in its travels by the public key and userIDs it affects. If the affected key is NOT already on the keyring, the compromise certificate (and its key and user ID) is merely added to the keyring anywhere. If the affected key IS already on the keyring, the compromise certificate is inserted after the affected key packet. This assumes that the actual key packet is identical to the one already on the key ring, so no duplicate key packet is needed. If a key has been revoked, PGP will not allow its use to encipher any messages, and if an incoming signature uses it, PGP will display a stern warning that this key has been revoked. NOTE: Key/userid revocation certificates ARE NOT SUPPORTED in this version of PGP. But if we ever get around to supporting them, here are some ideas on how they should work... A key/userid revocation certificate may be issued by someone to dissolve the association between his own key and a user ID. He would sign it with the very key that is being revoked. A key/userid revocation certificate issued by the keyholder shall take precedence over any other key certifications made by anyone else for that key/userid pair. Also, a third party certifier may revoke his own previous certification of this key/userid pair by issuing a key/userid revocation certificate. Such a revocation should not affect the certifications by other third parties for this same key/userid pair. When a key/userid revocation certificate is submitted to PGP, PGP will place it on the public keyring. A key/userid revocation certificate is always accompanied in its travels by the public key it affects (the key packet and user ID packet precedes the revocation certificate). If the affected key is NOT already on the keyring, the revocation certificate (and its key and user ID) is merely added to the keyring anywhere. If the affected key IS already on the keyring, the revocation certificate is integrated in with the key's other certificates as though it were just another key certification. This assumes that the actual key packet is identical to the one already on the key ring, so no duplicate key packet is needed.

Message digest "packet"

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

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

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

MSB	.	.	.	LSB
0	1	MD(16 bytes)	0	FF(n bytes) 1

Enough bytes of FF padding are added to make the length of this whole string equal to the number of bytes in the modulus. PGP versions 2.3 and later encode the MD into the MPI as follows:

MSB	.	.	.	LSB
0	1	FF(n bytes)	0	ASN(18 bytes)MD(16 bytes)

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

3020300c06082a864886f70d020505000410

Enough bytes of FF padding are added to make the length of this whole string equal to the number of bytes in the modulus. All this mainly affects the rsa_private_encrypt() and rsa_public_decrypt() functions in rsaglue.c. There is no checksum included. The padding serves to verify that the correct RSA key was used.

Conventional Data Encryption Key (DEK) "packet"

The DEK has no CTB packet framing. The DEK is stored packetless and naked, with padding, encrypted inside the MPI in the RSA public-key-encrypted packet. PGP versions 2.3 and later use a new format for encoding the message digest into the MPI in the signature packet. (This format is not presently based on any RFCs due to the use of the IDEA encryption system.) This format is accepted but not written by version 2.2. The older format used by versions 2.2 and earlier is also accepted by versions up to 2.4, but the RSAREF code in 2.5 is unable to cope with it. PGP versions 2.2 and earlier encode the MD into the MPI as follows:

MSB . . . LSB

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

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

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

MSB . . . LSB

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

CSUM refers to a 16-bit checksum appended to the high end of the DEK. RND is a string of NONZERO pseudorandom bytes, enough to make the length of this whole string equal to the number of bytes in the modulus. For both versions, the 16-bit checksum is computed on the rest of the bytes in the DEK key material, and does not include any other material in the calculation. In the above MSB-first representation, the checksum is also stored MSB-first. The checksum is there to help us determine if we used the right RSA secret key for decryption. All this mainly affects the rsa_public_encrypt() and rsa_private_decrypt() functions in rsaglue.c.

Conventional Key Encrypted data packet

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

The decrypted ciphertext may contain a compressed data packet or a literal plaintext packet. After decrypting the conventionally-encrypted data, a special 8-byte random prefix and 2 "key check" bytes are revealed. The random prefix and key check prefix are inserted before encryption and discarded after decryption. This prefix group is visible after decrypting the ciphertext in the packet. The random prefix serves to start off the cipher feedback chaining process with 64 bits of random material. It may be discarded after decryption. The first 8 bytes is the random prefix material, followed by the 2-byte "key-check" prefix. The key-check prefix is composed of two identical copies of the last 2 random bytes in the random prefix, in the same order. During decryption, the 9th and 10th bytes of decrypted plaintext are checked to see if they match the 7th and 8th bytes, respectively. If these key-check bytes meet this criterion, then the conventional key is assumed to be correct. One unusual point about the way encryption is done. Using the IDEA cipher in CFB mode, the first 10 bytes are decrypted normally, but bytes 10 to 17, the first 8 bytes of the data proper, are encrypted using bytes 2 to 9 (the last 8 bytes of the key check prefix) as the IV. This is essentially using CFB-16 for one part of the encryption, while CFB-64 is used elsewhere.

Compressed data packet

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

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

Literal data packet, with filename and mode

Offset	Length	Meaning 0 1 CTB for raw literal data packet
1	4	32-bit (or maybe 16-bit) length of packet
5	1	mode byte, 'b'= binary or 't'= canonical text
6	?	filename, with leading string length byte
?	4	Timestamp of last-modified date, or 0, or right now
?	?	raw literal plaintext data

The timestamp may be have to be derived in a system dependent manner. ANSI C functions should be used to get it if available, otherwise store the current time in it. Or maybe store 0 if it's somehow not applicable. When calculating a signature on a literal packet, the signature calculation only includes the raw literal plaintext data that begins AFTER the header fields in the literal packet_ after the CTB, the length, the mode byte, the filename, and the timestamp. The reason for this is to guarantee that detached signatures are exactly the same as attached signatures prefixed to the message. Detached signatures are calculated on a separate file that has no packet encapsulation.

A comment packet is generally just skipped over by PGP, although it may be displayed to the user when processed. It can be put in a keyring, or anywhere else. Comment packets are currently not generated by PGP.

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

Secret key certificate

All secret fields in the secret key certificate may be password-encrypted, including the checksum. The checksum is calculated from all of the bytes of the unenciphered secret components. The public fields are not encrypted. The encrypted fields are done in CFB mode, and the checksum is used to tell if the password was good. The CFB IV field is just encrypted random data, assuming the "true" IV was zero. NOTE: The secret key packet does not contain a User ID field. The User ID is enclosed in a separate packet that always follows the secret key packet on a keyring or in any other context.

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

Public key certificate

NOTE: The public key packet does not contain a User ID field. The User ID is enclosed in a separate packet that always follows somewhere after the public key packet on a keyring or in any other context. The validity period is currently always set to 0.

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

User ID packet

The User ID packet follows a public key on a public key ring. It also follows a secret key on a secret key ring. When a key is certified by a signature, the signature covers both the public key packet and the User ID packet. The signature certificate thereby logically "binds" together the user ID with the key. The user ID packet is always associated with the most recently occurring public key on the key ring, regardless of whether there are other packet types appearing between the public key packet and the associated user ID packet. There may be more than one User ID packet after a public key packet. They all would be associated with the preceding public key packet.

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

Keyring trust packet

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

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

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

Bits 0-2 - OWNERTRUST bits - Trust bits for this key owner. Values are:
000 - undefined, or uninitialized trust.
001 - unknown, we don't know the owner of this key.
010 - We usually do not trust this key owner to sign other keys.
011 - reserved
100 - reserved
101 - We usually do trust this key owner to sign other keys.
110 - We always trust this key owner to sign other keys.
111 - This key is also present in the secret keyring.

Bits 3-4 - Reserved.

Bit 5 - DISABLED bit - Means that this key is disabled, and
should not be used.

Bit 6 - Reserved

Bit 7 - BUCKSTOP bit - Means this key also appears in secret key ring.
Signifies the ultimately-trusted "keyring owner".
"The buck stops here". This bit computed from looking
at secret key ring. If this bit is set, then all the
KEYLEGIT fields are set to maximum for all the user IDs for
this key, and OWNERTRUST is also set to ultimate trust.

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

Bit 0-1 - KEYLEGIT bits - Validity bits for this key.
Set if we believe the preceding key is legitimately owned by
who it appears to belong to, specified by the preceding user
ID. Computed from various signature trust packets that
follow. Also, always fully set if BUCKSTOP is set.
To define the KEYLEGIT byte does not require that
OWNERTRUST be nonzero, but OWNERTRUST nonzero does require
that KEYLEGIT be fully set to maximum trust.
00 - unknown, undefined, or uninitialized trust.
01 - We do not trust this key's ownership.
10 - We have marginal confidence of this key's ownership.
Totally useless for certifying other keys, but may be useful
for checking message signatures with an advisory warning
to the user.
11 - We completely trust this key's ownership.
This requires either:
- 1 ultimately trusted signature (a signature from
yourself, SIGTRUST=111)

- COMPLETES_NEEDED completely trusted signatures
  (SIGTRUST=110)
- MARGINALS_NEEDED marginally trusted signatures
  (SIGTRUST=101)

COMPLETES_NEEDED and MARGINALS_NEEDED are configurable constants.

Bit 7 - WARNONLY bit - If the user wants to use a not fully validated key for encryption, he is asked if he really wants to use this key. If the user answers 'yes', the WARNONLY bit gets set, and the next time he uses this key, only a warning will be printed. This bit gets cleared during the maintenance pass.

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

Bits 0-2 - SIGTRUST bits - Trust bits for this signature. Value is copied directly from OWNERTRUST bits of signer:
000 - undefined, or uninitialized trust.
001 - unknown
010 - We do not trust this signature.
011 - reserved
100 - reserved
101 - We reasonably trust this signature.
110 - We completely trust this signature.
111 - ultimately trusted signature (from the owner of the ring)

Bits 3-6 - Reserved.

Bit 6 - CHECKED bit - This means that the key checking pass (pgp -kc, also invoked automatically whenever keys are added to the keyring) has tested this signature and found it good. If this bit is not set, the maintenance pass considers this signature untrustworthy.

Bit 7 - CONTIG bit - Means this signature leads up a contiguous trusted certification path all the way back to the ultimately-trusted keyring owner, where the buck stops. This bit is derived from other trust packets. It is currently not used for anything in PGP.

The OWNERTRUST bits are set by the user. PGP does not modify them. PGP computes the BUCKSTOP bit by checking to see if the key is on the secret key ring. If it is, it was created by this user, and thus controlled by him. All other trust is derived from the BUCKSTOP keys in a special maintenance pass over the keyring. Any good signature made by a given key has its SIGTRUST equal to the key's OWNERTRUST. Based on COMPLETES_NEEDED and MARGINALS_NEEDED, if enough trusted signatures are on a key/userID pair, the key/userid association is considered legitimate. To be precise, an ultimately trusted key has weight 1, a completely trusted key has weight 1/COMPLETES_NEEDED (or 0 if COMPLETES_NEEDED is 0), and a marginally trsuted key has weight 1/MARGINALS_NEEDED. Other trust values have weight 0. If the total weight of the signatures on a key/userid pair is 1 or more, the userid is considered legitimate. When a key has a legitimate userid, the user is asked to set the OWNERTRUST for the corresponding key. Ths idea is that the userid identifies someone the user knows, at least by reputation, so once it has been established who holds the secret key, that person's trustworthiness as an introducer can be established and assigned to the key. Once that is done, the key's signatures then have weight establishing other key/userid associations. There is a limit to the depth to which this can go. Keys on the secret keyring are at depth 0. Keys signed by those keys are at depth 1. Keys which are fully certified using only signatures from keys at depth 1 or less are at depth 2. Keys which are fully certified using only signatures from keys at depth 2 or less are at depth 3, and so on. If you know all of your trusted introducers personally, and have signed their

keys, then you will never have a key at a depth of greater than 2. The maximum depth is limited my MAX_CERT_DPETH. It never gets very large in a well-connected "web of trust". This redundant and decentralized method of determining public key legitimacy is one of the principal strengths of PGP's key management architecture, as compared with PEM, when used in social structures that are not miltiary-style rigid hierarchies. The trust of a key owner (OWNERTRUST) does not just reflect our estimation of their personal integrity, it also reflects how competent we think they are at understanding key management and using good judgement in signing keys. The OWNERTRUST bits are not computed from anything _ it requires asking the user for his opinion.

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

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

When a key is added to the key ring the trust bytes are initialized to zero (undefined). [_manual setting of SIGTRUST/OWNERTRUST not implemented] Normally, we derive the value of the SIGTRUST field by copying it directly from the signer key's OWNERTRUST field. Under special circumstances, if the user explicitly requests it with a special PGP command, we may let the user override the copied value for SIGTRUST by displaying an advisory to him and asking him for ratification, like so:

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

Or:

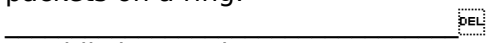
This key is signed by "Oliver North", whom you usually do not trust to sign keys. Do you trust "Oliver North" to certify the key for "Daniel Ellsberg"? (1=Yes, 2=No, 3=Somewhat, 4=I don't know) ? _ <default is no>

An "I don't know" response to this question would have the same effect as a response of "no". If we had no information about the trustworthiness of the signer (the OWNERTRUST field was uninitialized), we would leave the advisory note off. Certifying a public key is a serious matter, essentially promising to the world that you vouch for this key's ownership. But sometimes I just want to make a "working assumption" of trust for someone's public key, for my own purposes on my own keyring, without taking the serious step of actually certifying it for the rest of the world. In that case, we can use a special PGP keyring management command to manually set the
KEYLEGIT field, without relying on it being computed during a maintenance pass. Later, if a maintenance pass discovers a KEYLEGIT bit set that would not have been otherwise computed as set by the maintenance pass logic, it alerts me and asks me to confirm that I really want it set. [_end of not implemented section] During routine use of the public keyring, we don't actually check the associated signatures certifying a public key. Rather, we always rely on trust bytes to tell us whether to trust the key in question. We depend on a separate checking pass (pgp -kc) to actually check the key signature certificates against the associated keys, and to set the trust bytes accordingly. This pass checks signatures, and if a signature fails to verify, obnoxiously alerts the user and drops it from the key ring. Then it tuns a maintenance pass to calculate the ring-wide effects of this. A failed signature should be exceedingly rare, and it may not even result in a KEYLEGIT field being downgraded. Having several signatures certifying each key should prevent damage from spreading too far from a failed certificate. But if dominoes do keep falling from this, it may indicate the discovery of an important elaborate attack. The maintenance pass is run every time the keyring changes, and operates in a top-of-pyramid-down manner as follows. If at any time during any of these steps the KEYLEGIT field goes from not fully set

to fully set, and the OWNERTRUST bits are still undefined, the user is asked a question to define the OWNERTRUST bits. First, for all keys with BUCKSTOP set, check if they are really present in the secret keyring, if not, the BUCKSTOP bit is cleared. SIGTRUST and KEYLEGIT is initialized to zero for non-buckstop keys. The real maintenance pass is done in a recursive scan: Start with BUCKSTOP keys, find all userid/key pairs signed by a key and update the trust value of these signatures by copying the OWNERTRUST of the signer to the SIGTRUST of the signature. If this makes a key fully validated, start looking for signatures made by this key, and update the trust value for them. Repeat until everything has settled down.

Public Key Ring Overall Structure

A public key ring is comprised of a series of public key packets, keyring trust packets, user ID packets, and signature certificates. Here is an example of an ordered collection of packets on a ring:



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

Key/userid revocation certificates (PGP 2.62)

NOTE: Key/userid revocation certificates ARE NOT SUPPORTED in this version of PGP. But if we ever get around to supporting them, here are some ideas on how they should work... A key/userid revocation certificate may be issued by someone to dissolve the association between his own key and a user ID. He would sign it with the very key that is being revoked. A key/userid revocation certificate issued by the keyholder shall take precedence over any other key certifications made by anyone else for that key/userid pair. Also, a third party certifier may revoke his own previous certification of this key/userid pair by issuing a key/userid revocation certificate. Such a revocation should not affect the certifications by other third parties for this same key/userid pair. When a key/userid revocation certificate is submitted to PGP, PGP will place it on the public keyring. A key/userid revocation certificate is always accompanied in its travels by the public key it affects (the key packet and user ID packet precedes the revocation certificate). If the affected key is NOT already on the keyring, the revocation certificate (and its key and user ID) is merely added to the keyring anywhere. If the affected key IS already on the keyring, the revocation certificate is integrated in with the key's other certificates as though it were just another key certification. This assumes that the actual key packet is identical to the one already on the key ring, so no duplicate key packet is needed.

How to Install PGP

The first question is, what platform are you on?

The base PGP 2.6 distribution runs on several varieties of Unix, MS-DOS and VAX VMS. Ports can be expected shortly to the Atari, Amiga, and possibly other systems. Naturally, installation instructions differ
depending on your hardware. Separate instructions are provided here for MSDOS and Unix.

No matter what the machine you are on, though, do this...

STEP 1:

READ THE DOCUMENTATION. At least read Volume I of the PGP User's Guide. Cryptography software is easy to misuse, and if you don't use it properly much of the security you could gain by using it will be lost! You might also be unfamiliar with the concepts behind public key cryptography; the manual explains these ideas. Even if you are already familiar with public key cryptography, it is important that you understand the various security issues associated with using PGP. PGP may be an unpickable lock, but you have to install it in the door properly or it won't provide security.

See the section below for your system's particular installation instructions.

If you do not have any of these systems, you will either have to port the sources to your machine or find someone who has already done so.

##
For MSDOS:

PGP is distributed in a compressed archive format, which keeps all the relevant files grouped together, and also saves disk space and transmission time.

The current version, 2.6, is archived with the ZIP utility, and the PGP executable binary release system is in a file named PGP26.ZIP. This contains the executable program, the user documentation, the
RSAREF license, and a few keys and signatures. There is also a second file available containing the C and assembly source code, called PGP26SRC.ZIP. If you are a programmer, this may be of interest
to you. This should be available from the same source from which you got PGP26.ZIP. If not, and you want it, see the Licensing and Distribution section of the PGP User's Guide.

You will need PKUNZIP version 1.1 or later to uncompress and split the PGP26.ZIP archive file into individual files. PKUNZIP is shareware and is widely available on MSDOS machines.

Create a directory for the PGP files. For this description, let's use the directory C:\PGP as an example, but you should substitute your own disk and directory name if you use something different. Type these commands to make the new directory:

```
c:
md \pgp
cd \pgp
```

Uncompress the distribution file PGP26.ZIP to the directory. For this example, we will

assume the file is on floppy drive A - if not, substitute your own file location.

```
pkunzip -d a:pgp26
```

If you omit the -d flag, all the files in the doc subdirectory will be deposited in the pgp directory. This merely causes clutter.

This will create the file PGP26I.ZIP and PGP26I.ASC. Unzip PGP26I.ZIP with the command:

```
pkunzip -d pgp26i
```

Keep the PGP26I.ZIP file around. Once you have PGP working you can use PGP26I.ASC to verify the digital signature on PGP26I.ZIP. It should come from Jeffrey I. Schiller (whose key is included in keys.asc).

Setting the Environment

Next, you can set an MSDOS "environment variable" to let PGP know where to find its special files, in case you use it from other than the default PGP directory. Use your favorite text editor to add the following lines to your AUTOEXEC.BAT file (usually on your C: drive):

```
SET PGPPATH=C:\PGP
SET PATH=C:\PGP;%PATH%
```

Substitute your own directory name if different from "C:\PGP".

The CONFIG.TXT file contains various preferences. You can change the language PGP operates in, and the character set it uses. The IBM PC's default character set, "Code Page 850" will be used if the
line "charset = cp850" appears in the config.txt file. You probably want to add that line.

Another environmental variable you should set in MSDOS is "TZ", which tells MSDOS what time zone you are in, which helps PGP create GMT timestamps for its keys and signatures. If you properly define TZ in AUTOEXEC.BAT, then MSDOS gives you good GMT timestamps, and will handle daylight savings time adjustments for you. Here are some sample lines to insert into AUTOEXEC.BAT, depending on your time zone:

```
For Los Angeles:   SET TZ=PST8PDT
For Denver:        SET TZ=MST7MDT
For Arizona:       SET TZ=MST7
   (Arizona never uses daylight savings time)
For Chicago:       SET TZ=CST6CDT
For New York:      SET TZ=EST5EDT
For London:        SET TZ=GMT0BST
For Amsterdam:     SET TZ=MET-1DST
For Moscow:        SET TZ=MSK-3MSD
For Aukland:       SET TZ=NZT-13
```

Now reboot your system to run AUTOEXEC.BAT, which will set up PGPPATH and TZ for you.

Generating Your First Key

One of the first things you will want to do to really use PGP (other than to test itself) is to generate your own key. This is described in more detail in the "RSA Key Generation" section of the PGP User's
Guide. Remember that your key becomes something like your written signature or your bank card code number or even a house key - keep it secret and keep it secure! Use a long, unguessable pass phrase and remember it. Right after you generate a key, put it on your key rings and copy your secret keyring (SECRING.PGP) to a blank floppy and write protect the floppy.

If you are a first-time user of PGP, it is a good idea to generate a short test key, with a short passphrase, to play around with PGP for a little bit and see how it works, or even more than one so you can pretend to be sending messages between two different people. Since you won't be guarding any secrets, this can be short and have a simple pass phrase. But when you generate your permanent key, that you intend to give to others so they can send secure messages to you, be much more careful.

After you generate your own key pair, you can add a few more public keys to your key ring. A collection of sample public keys is provided with the release in the file KEYS.ASC. To add them to your public key ring, see the PGP User's Guide, in the section on adding keys to your key ring.

##
For UNIX:

You likely will have to compile PGP for your system; to do this, first make sure the unpacked files are in the correct unix textfile format (the files in pgp23src.zip are in MSDOS CRLF format, so for Unix you
must unpack with "unzip -a"; the tar file pgp23.tar.Z uses normal Unix line feed conventions). Then copy the file "makefile.unx" in the distribution to "Makefile".

Then, you will need the March 16, 1994 release of the RSAREF 2.0 package. It is included with the PGP 2.5 distribution from MIT. It should be unpacked in a directory named "rsaref2" that is a sibling of the directory that PGP is unpacked in. (If you use a different location, you will have to modify the Makefile and rsaglue2.c.)

Make a directory rsaref2/unix, copy the makefile over from rsaref2/install/unix, and build the rsaref.a library. The RSAREF package has more detailed instructions.

If you don't have an ANSI C compiler you will need the unproto package written by Wietse Venema. unproto was posted on comp.sources.misc and can be obtained from the various sites that archive this newsgroup (volume 23: v23i012 and v23i013) or ftp.win.tue.nl file: /pub/programming/unproto4.shar.Z Read the file README in the unproto distribution for instructions on how to use unproto. The unix makefile for pgp (makefile.unx) contains a few targets for compliling with unproto, these assume you have unpacked unproto in a subdirectory "unproto" in the pgp "src" directory.

Then...

type:
"make sungcc" for Sun with GNU gcc

"make suncc" for Sun with cc and unproto
"make sysv_386" for SVR4 386 with asm primitives
"make x286" for XENIX/286 with asm primitives and unproto
"make ultrix" for DEC 4.2BSD Ultrix with gcc
"make rs6000" for RS6000 AIX
"make irix_asm" for IRIX with asm primitives
"make" to list the available platforms

There are more targets in makefile.unx. If your system doesn't have a target in makefile.unx you will have to edit the makefile, make sure you compile for the correct byte order for your system: define
HIGHFIRST if your system is big-endian (eg. Motorola 68030). There are also some platform-specific parameters in the include file "platform.h". Some platforms may have to modify this file.

If all goes well, you will end up with an executable file called "pgp".

Before you install pgp, run these tests: (do not create your real public key yet, this is just for testing pgp)

- create a public/secret key pair (enter "test" as userid/password):
    pgp -kg

- add the sample keys from the file "keys.asc" to the public keyring:
    pgp -ka keys.asc
  pgp will ask if you want to sign the keys you are adding, answer yes for at least one key.

- do a keyring check:
    pgp -kc

- encrypt pgpdoc1.txt:
    pgp -e pgpdoc1.txt test -o testfile.pgp

- decrypt this file:
    pgp testfile.pgp

this should produce the file "testfile" compare this file with pgpdoc1.txt

If everything went well, install pgp in a bin directory.

Place the documentation, pgpdoc1.txt and pgpdoc2.txt somewhere where you can reasonably read it. The software looks for it when running (especially generating keys), so someplace reasonably obvious would be good. "pgp -kg" will give you full details if it can't find the manuals.

Place the man page (pgp.1) in an appropriate spot. If you don't know anything about how man pages work, you can make the man page look human readable yourself by typing "nroff -man pgp.1 >pgp.man" and reading "pgp.man".

Create a subdirectory somewhere in your home directory hierarchy to hold your public and private key rings and anything else pgp might need (like the language.txt file). The default name PGP assumes is ~/.pgp. If you want to use a different name, you must set the environment variable "PGPPATH" to point to this place before you use the system.

> IMPORTANT: This directory cannot be shared! It will contain your < > personal private

keys! <

If you are installing PGP for yourself, copy the files "language.txt", "config.txt", and the ".hlp" files from the distribution into this subdirectory.

If you are installing PGP system-wide, the directory to use is /usr/local/lib/pgp for the config, language and help files. This can be changed in fileio.h when compiling. It's the value of PGP_SYSTEM_DIR.

Tell PGP the character set and language you wish to use in the config.txt file. If you have a terminal that only displays 7-bit ASCII, use "charset=ascii" to display an approximation (accents are omitted) of extended characters.

>> IMPORTANT: Please read the sections in the man page and manual << >> about vulnerabilities before using this software on a multi- << >> user machine!
<<

Now, if you haven't done so yet, GO READ THE MANUAL.

##
For VMS:

Read the file readme.vms in the doc subdirectory

##

PGP 2.62i

This is to announce the release of PGP 2.6.2i, the latest international version of PGP. PGP 2.6.2i can be downloaded via WWW from:

http://www.ifi.uio.no/~staalesc/PGP/home.html

It will soon be available by FTP, too. I quote from the README file for PGP 2.6.2i:

ABOUT THIS VERSION

PGP 2.6.2i is not an official PGP version. It is based on the source code for MIT PGP 2.6.2 (the latest official version of PGP) and has been modified for international use. PGP 2.6.2i is probably illegal to use within the USA, but is fine in almost every other country in the world. This release is basically a bug-fix for PGP 2.6.i, which was based on PGP 2.6

DIFFERENCES BETWEEN PGP 2.6.2i AND 2.6.2

PGP 2.6.2i differs from MIT PGP 2.6.2 in the following ways:

(1) It identifies itself as version 2.6.2i

This is to clearly distinguish it from PGP 2.6.2. This is important because users within the USA should not use PGP 2.6.2i, and also because script files, shells and other PGP add-ons may need to know exactly how your copy of PGP will behave under different circumstances.

(2) It uses PRZ's MPILIB instead of RSAREF

PGP 2.3a and earlier versions use a special library for all the RSA encryption/decryption routines, called MPILIB, and written by Philip R. Zimmermann (PRZ), the original author of PGP. However, starting with version 2.5, all official releases of PGP have been using the RSAREF library from RSADSI Inc, a US company that holds the patent on the RSA algorithm in the USA. This change was made in order to make PGP legal to use within the USA.

Please observe that PGP 2.6.2i does NOT use RSAREF, but rather PRZ's original MPILIB library, which is functionally identical to RSAREF and slightly faster on most platforms. Because 2.6.2i uses MPILIB rather than RSAREF, this PGP version is also able to verify key signatures made with PGP 2.2 or earlier versions. This is not true for MIT PGP, because the RSAREF library only understands the new PKCS signature format introduced in PGP 2.3.

The use of the MPILIB library is the main reason why PGP 2.6.2i is probably illegal to use within the USA. If you are in the USA, you should compile the source code using the -DMIT option and link it with the RSAREF library rather than MPILIB. Note that RSAREF is NOT included in this distribution, so if you are a US user, it is probably easier to get a copy of the original MIT 2.6.2 release.

(3) It lets you disable the "legal kludge"

PGP 2.6.2 contains a "feature" that will cause it to generate keys and messages that are not readable by PGP 2.3a and earlier versions. This is the "legal kludge", and was introduced to encourage users in the USA to upgrade from PGP 2.3a.

PGP 2.6.2i provides you with a way to disable the "legal kludge". This means that

messages and keys generated with PGP 2.6.2i can be used and understood by all existing 2.x versions of PGP. To disable the legal kludge, uncomment the following line in your config.txt file so that it reads:

legal_kludge = off

This option may also be set on the command line: "pgp +le=off <command>".

(4) It allows you to generate keys up to and including 2048 bits

Because of a bug in PGP 2.6.2, this version would not let you generate keys bigger than 2047 bits on some platforms. This problem has been corrected in PGP 2.6.2i.

(5) It contains a number of bug-fixes

PGP 2.6.2i also fixes a number of other bugs found in PGP 2.6.2, most notably the signature bug for keys over 2034 bits, as reported by ViaCrypt.

(6) It can be compiled on many new platforms

PGP 2.6.2i has been modified in order to let it compile "out of the box" for such platforms as Amiga, Atari, FreeBSD, UnixWare and various flavours of VMS. It can also be compiled under MS-DOS using Borland C (MIT PGP 2.6.2 only supports Microsoft C).

(7) It includes updated documentation and language files

The language files for MIT PGP 2.6.2 had not been updated for a long time. This has been fixed in this version. PGP 2.6.2i comes with standard language files for French, Spanish and Norwegian. All the other text and documentation files for PGP 2.6.2i have also been brought up to date.

DIFFERENCES BETWEEN PGP 2.6.2i and 2.6ui

A PGP version that has been very popular among non-US users of PGP is 2.6ui. If you have been using PGP 2.6ui up to now, you should note that PGP 2.6.2i differs from this version in the following ways:

(1) It is a "real" 2.6 version

PGP 2.6.2i is based on the source code for PGP 2.6.2, whereas PGP 2.6ui is based on the source code for 2.3a. This means that 2.6.2i contains a lot of bug-fixes that are not present in 2.6ui, and it also adds a number of new features that are lacking in 2.6ui. These include the new PUBRING, SECRING, RANDSEED and COMMENT options in config.txt.

(2) It doesn't have the version_byte option

PGP 2.6ui has an option to allow you to choose which message format to use when generating keys and messages. This is the version_byte option, and can be set both in the config.txt file and on the command line:

version_byte = 2 (use backwards-compatible format, default)
version_byte = 3 (use new 2.6 format)

In PGP 2.6.2i, the same is accomplished using the legal_kludge flag:

legal_kludge = off (use backwards-compatible format)
legal_kludge = on (use new 2.6 format, default)

(3) It doesn't have the armor_version option

PGP 2.6ui has an option to let you "forge" the version number in the ASCII armored files produced by PGP. In PGP 2.6.2i, the armor_version option is NOT supported, as this is a feature that is heavily misused.

If you must change the version number of your keys and messages, you can do so in the language.txt file instead.

LEGAL STUFF

PGP 2.6.2i is not approved by MIT or PRZ or NSA or the Pope or anyone else. However, it should be possible to use it legally by anyone in the free world (i.e. all countries except USA, Iraq and a few others). There are three reasons why people may claim (incorrectly) that PGP 2.6.2i is illegal:

(1) It is based on source code that was illegally exported from the USA

The ITAR regulations classifies cryptography in the same category as munitions, and so it is very likely that exporting PGP from the USA is considered illegal by US authorities. In the case of PGP 2.6.2i, large portions of the code were written inside the USA, and later exported to the rest of the world. However, this is not a problem, because it is the _export_ that is illegal, not the _use_ of the program. Once the software is (illegally) exported, anyone may use it legally. (I didn't export it, and I strongly recommend that you won't do it either.) As long as you make sure that you get your copy of PGP 2.6.2i from somewhere outside the USA, then you should be on the safe side.

(2) It infringes the RSA patent

This is not a problem either, because PGP 2.6.2i is not intended for use in the USA (which just happens to be the only country in the world where the RSA patent is valid, and still the validity of this patent is somewhat dubious). If you are inside the USA, you should obtain a copy of PGP 2.6.2 instead, or compile the source using the -DMIT option and link it with the RSAREF library, which will in fact give you a version that is identical to MIT PGP 2.6.2.

(3) It violates the MIT license

The second point in the MIT license for PGP 2.6.2 explicitly forbids anyone to remove the so-called "legal kludge". Still, this is exactly what PGP 2.6.2i does. However, it should be clear that this limitation only refers to the RSAREF versions of PGP. PGP 2.6.2i, on the other hand, does not use RSAREF, and so this point becomes irrelevant. If you still feel uncomfortable about this, take a look at the file przon26i.asc which is included in the distribution archive. This file contains a statement by Phil Zimmermann on PGP 2.6.i and the various "unofficial international" versions of PGP. (PGP 2.6.2i is simply a bug- fix for PGP 2.6.i, so everything that covers 2.6.i applies to 2.6.2i as well.)

COMMENTS AND BUG REPORTS

PGP 2.6.2i was put together by Stale Schumacher <staalesc@ifi.uio.no> with the help of many individuals around the world (see the file diffs.doc for details).

All questions regarding PGP 2.6.2i should be addressed directly to him, or to <pgp-bugs@ifi.uio.no>. Please note that PRZ, MIT and the University of Oslo have nothing to do with this release. Comments, bug reports and suggestions for future releases are welcome.