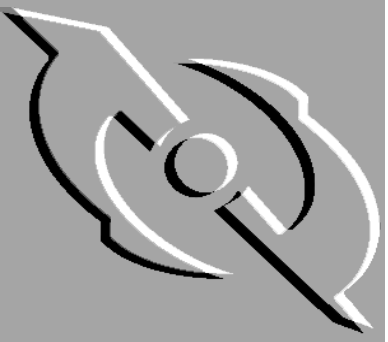

PGP Software Developer's Kit

User's Guide

Version 1.7

PGP* Software Developer's Kit, Version 1.7.1

6-99. Printed in the United States of America.

Network Associates, Inc. (408) 988-3832 main
3965 Freedom Circle
Santa Clara, CA 95054
http://www.nai.com

info@nai.com

***** is sometimes used instead of the **®** for registered trademarks to protect marks registered

LIMITED WARRANTY

Limited Warranty. Network Associates warrants that for sixty (60) days from the date of original purchase the media (for example diskettes) on which the Software is contained will be free from defects in materials and workmanship.

Customer Remedies. Network Associates' and its suppliers' entire liability and your exclusive remedy shall be, at Network Associates' option, either (i) return of the purchase price paid for the license, if any, or (ii) replacement of the defective media in which the Software is contained with a copy on nondefective media. You must return the defective media to Network Associates at your expense with a copy of your receipt. This limited warranty is void if the defect has resulted from accident, abuse, or misapplication. Any replacement media will be warranted for the remainder of the original warranty period. Outside the United States, this remedy is not available to the extent Network Associates is subject to restrictions under United States export control laws and regulations.

Warranty Disclaimer. To the maximum extent permitted by applicable law, and except for the limited warranty set forth herein, THE SOFTWARE IS PROVIDED ON AN “AS IS” BASIS WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. WITHOUT LIMITING THE FOREGOING PROVISIONS, YOU ASSUME RESPONSIBILITY FOR SELECTING THE SOFTWARE TO ACHIEVE YOUR INTENDED RESULTS, AND FOR THE INSTALLATION OF, USE OF, AND RESULTS OBTAINED FROM THE SOFTWARE. WITHOUT LIMITING THE FOREGOING PROVISIONS, NETWORK ASSOCIATES MAKES NO WARRANTY THAT THE SOFTWARE WILL BE ERROR-FREE OR FREE FROM INTERRUPTIONS OR OTHER FAILURES OR THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, NETWORK ASSOCIATES DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT WITH RESPECT TO THE SOFTWARE AND THE ACCOMPANYING DOCUMENTATION. SOME STATES AND JURISDICTIONS DO NOT ALLOW LIMITATIONS ON IMPLIED WARRANTIES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. The foregoing provisions shall be enforceable to the maximum extent permitted by applicable law.

Table of Contents

Preface

Who should read this book?

This User's Guide explains how to use the PGPsdk (software developer's kit), a collection of PGP digital privacy tools for software developers. Read it if you're a software developer working on the Unix, Windows, or Mac OS platform and you want to add PGP compatibility or features to your programs.

You'll need to be an experienced C language programmer to understand and use this material.

Related material

NOTE: All PGPsdk programmers will also need a copy of the PGPsdk Reference Guide, which is a complete catalog of the functions in the PGPsdk libraries. The Reference Guide is included in the PGPsdk package.

If you are new to cryptography and would like an overview of the terminology and concepts you will encounter while using PGP, please see the Network Associates publication *An Introduction to Cryptography*. You may also want to explore the "Recommended Readings." listed at the end of this Preface.

About this User's Guide

The idea of this PGPsdk User's Guide is to show developers:

- the general mechanics of programming with the PGPsdk headers and libraries,
- how to use those headers and the libraries to add PGP functionality to a program, by presenting code fragments for a few of the most frequently-needed PGP operations as examples, and
- for developers new to PGP or crypto in general, an introduction to the fundamental cryptosystem ideas that shaped the API.

To those ends, this User's Guide is organized as follows:

Chapter 1: Introduction to the PGPsdk

Presents an overview of the PGPsdk, describes important basic concepts of the PGP cryptosystem, and introduces the core crypto operations on which the rest of the system depends.

Chapter 2: PGPsdk Organization

Describes how the functionality of the PGP cryptosystem is broken out by subject area into the several runtime libraries and interface headers (.h files).

Chapter 3: Programming with the PGPsdk

Explains coding conventions used in the PGP interfaces, and how to work with the PGP libraries in general.

Chapter 4: Implementing Common PGP Operations

This chapter contains example code snippets that demonstrate how to do most of the basic operations you'll need to provide your users with PGP encryption and authentication services.

Chapter 5: PGPsdk Frequently Asked Questions

From our developer tech support group, Chapter 5 is a list of answers to the questions that developers most often ask regarding programming with the PGPsdk. If you experience any difficulty while working with the PGPsdk, look here first.

Developer support

Network Associates would like to thank you for your interest in adding PGP functionality to your products. If you should need any additional information beyond what we've provided in the PGPsdk, please contact the PGPsdk group in our Developer Technical Services department (see next section).

How to contact Network Associates

Customer service

To order products or obtain product information, contact the Network Associates Customer Care department at (408) 988-3832 or write to the following address:

Network Associates, Inc.
McCandless Towers
3965 Freedom Circle
Santa Clara, CA 95054-1203
U.S.A.

Technical support

Network Associates is famous for its dedication to customer satisfaction. We have continued this tradition by making our site on the World Wide Web a valuable resource for answers to technical support issues. We encourage you to make this your first stop for answers to frequently asked questions, for updates to Network Associates software, and for access to Network Associates news and encryption information.

World Wide Web	http://www.nai.com

Technical Support for your PGP product is also available through these channels:

Phone	(408) 988-3832
Email	PGPSupport@pgp.com

To provide the answers you need quickly and efficiently, the Network Associates technical support staff needs some information about your computer and your software. Please have this information ready before you call:

If the automated services do not have the answers you need, contact Network Associates at one of the following numbers Monday through Friday between 6:00 A.M. and 6:00 P.M. Pacific time.

Phone	(408) 988-3832

To provide the answers you need quickly and efficiently, the Network Associates technical support staff needs some information about your computer and your software. Please have this information ready before you call:

- Product name and version number
- Computer brand and model
- Any additional hardware or peripherals connected to your computer
- Operating system type and version numbers
- Network type and version, if applicable
- Content of any status or error message displayed on screen, or appearing in a log file (not all products produce log files)
- Email application and version (if the problem involves using PGP with an email product, for example, the Eudora plug-in)
- Specific steps to reproduce the problem

Year 2000 Compliance

Information regarding NAI products that are Year 2000 compliant and its Year 2000 standards and testing models may be obtained from NAI's website at http://www.nai.com/y2k.

For further information, email y2k@nai.com.

Network Associates training

For information about scheduling on-site training for any Network Associates product, call (800) 338-8754.

Comments and feedback

Network Associates appreciates your comments and feedback, but incurs no obligation to you for information you submit. Please address your comments about PGP product documentation to: Network Associates, Inc., 3965 Freedom Circle Santa Clara, CA 95054-1203 U.S.A.. You can also e-mail comments to tns_documentation@nai.com.

Recommended Readings

Non-Technical and beginning technical books

- Whitfield Diffie and Susan Eva Landau, “Privacy on the Line,” *MIT Press*; ISBN: 0262041677
  This book is a discussion of the history and policy surrounding cryptography and communications security. It is an excellent read, even for beginners and non-technical people, but with information that even a lot of experts don't know.

- David Kahn, “The Codebreakers” *Scribner*; ISBN: 0684831309
  This book is a history of codes and code breakers from the time of the Egyptians to the end of WWII. Kahn first wrote it in the sixties, and there is a revised edition published in 1996. This book won't teach you anything about how cryptography is done, but it has been the inspiration of the whole modern generation of cryptographers.

- Charlie Kaufman, Radia Perlman, and Mike Spencer, “Network Security: Private Communication in a Public World,” *Prentice Hall;* ISBN: 0-13-061466-1
  This is a good description of network security systems and protocols, including descriptions of what works, what doesn't work, and why. Published in 1995, so it doesn't have many of the latest advances, but is still a good book. It also contains one of the most clear descriptions of how DES works of any book written.

Intermediate books

- Bruce Schneier, “Applied Cryptography: Protocols, Algorithms, and Source Code in C,” *John Wiley & Sons*; ISBN: 0-471-12845-7
  This is a good beginning technical book on how a lot of cryptography works. If you want to become an expert, this is the place to start.

- Alfred J. Menezes, Paul C. van Oorschot, and Scott Vanstone, “Handbook of Applied Cryptography,” *CRC Press;* ISBN: 0-8493-8523-7
  This is the technical book you should get after Schneier. There is a lot of heavy-duty math in this book, but it is nonetheless usable for those who do not understand the math.

- Richard E. Smith, “Internet Cryptography,” *Addison-Wesley Pub Co*; ISBN: 020192480
  This book describes how many Internet security protocols. Most importantly, it describes how systems that are designed well nonetheless end up with flaws through careless operation. This book is light on math, and heavy on practical information.

- William R. Cheswick and Steven M. Bellovin, “Firewalls and Internet Security: Repelling the Wily Hacker” *Addison-Wesley Pub Co*; ISBN: 0201633574
  This book is written by two senior researcher at AT&T Bell Labs, about their experiences maintaining and redesigning AT&T's Internet connection. Very readable.

Advanced books

- Neal Koblitz, “A Course in Number Theory and Cryptography” *Springer-Verlag*; ISBN: 0-387-94293-9
  An excellent graduate-level mathematics textbook on number theory and cryptography.

- Eli Biham and Adi Shamir, “Differential Cryptanalysis of the Data Encryption Standard,” *Springer-Verlag*; ISBN: 0-387-97930-1
  This book describes the technique of differential cryptanalysis as applied to DES. It is an excellent book for learning about this technique.

Introduction to the PGPsdk 1

This chapter uses three sections to introduce the PGPsdk:

- **Overview of the PGPsdk**
- **Understanding the PGP Cryptosystem**
- **Core PGP Operations**

Overview of the PGPsdk

The PGPsdk provides everything a developer needs to incorporate any part of the PGP cryptosystem's functionality into their own software. The SDK includes the same C-language header files and runtime libraries that Network Associates' own developers use to create the PGP software sold and distributed by Network Associates.

Security programming considerations

Like any API guide, this document describes what to do; but note that because PGP is concerned with digital privacy, in many cases we also go further to point out what *not* to do, in order to help you avoid security holes that could make the rest of your efforts moot.

Library source is open for peer review

If you would like to examine the source code for the libraries, including the ciphering routines, Network Associates publishes that too. All of the functions in the PGP libraries are also documented in the accompanying *PGPsdk Reference Guide*.

It's unusual for a company to publish the source code and document the library interfaces of a commercial software product—but then, PGP is unusual software that addresses a uniquely serious problem: digital privacy.

The PGP cryptosystem is open-source because its consumers would have no reason to believe in it—indeed, shouldn't believe in any cryptosystem—unless its ciphering algorithms and protocols could be freely reviewed and approved by the community of cryptographic experts.

PGP continually passes this test.

SDK is available to encourage PGP ubiquity

Although Network Associates creates and publishes PGP software products of its own, we also recognize that PGP becomes more valuable to its users as more people use it. That's why we've gone further than just publishing the library source, to create and publish this PGPsdk in order to encourage other developers to furnish PGP privacy services in their own software as well.

Programming in the cryptosystem, not implementing ciphers

Developers new to the PGP API sometimes expect to find themselves dealing with the mathematical intricacies of robustly scrambling and unscrambling messages, but that really isn't what PGPsdk programming is about.

While some people do get interested in the internal structure of the PGP ciphering algorithms, you won't need to understand them to use the PGP libraries. That's because all the details of ciphering and deciphering operations are transparently handled by the API's high-level functions. For example, once you've done the necessary setup steps, all of the 'scrambling' is performed by the single function call `PGPEncode()`.

What's more important for you to understand are the general concepts of the PGP cryptosystem. It's this whole system that the PGP API supports, not just the ciphering steps. As you read, keep in mind that ciphering is just one operation in a much larger security scheme.

Understanding the PGP cryptosystem

Serious cryptography is not simple, and the PGP cryptosystem is no exception. Before you attempt to implement PGP functionality in code, you should have a good general understanding of the overall structure and functionality of the PGP approach, and especially of the particular parts of it that you plan to add to your program. It's beyond the scope of this User's Guide to give you that larger understanding, but we can present a quick recap of the most central concepts.

For pointers to some further readings on cryptosystems in general and PGP in particular, please see the Preface.

PGP in a nutshell

PGP is a system that can provide individuals and organizations with “Pretty Good Privacy” for their computer systems and data. PGP is based on cryptographic ciphering operations, and uses cryptography not only to encrypt data for security, but also as a basis for identity authentication and detection of data tampering services. PGP as a design has a highly reliable mathematical basis, and PGP as a software library has a highly reliable history of evolution and proven performance.

We’ll start our tour of PGP by covering some of the basic concepts of encryption, and work up to larger concepts like certification and trust relationships.

Why encrypt?

The fundamental aim of encryption is to keep the contents of an item of information secret until the intended recipient is ready to receive it. Since the desire to do this most often arises in the context of communication between two parties via an insecure carrier, the secret data is usually referred to as the *message*; however, it’s sometimes useful for privacy reasons to encrypt information even if it never leaves your own computer. PGP is basically an encryption-based system.

Ciphering and deciphering as keyed, symmetrical transform

Encrypting or *ciphering* digital information has to be an information-preserving, reversible operation; in other words, it requires both one transform function for ciphering, and the exact inverse transform function for deciphering. In the most simple, non-computer-based cryptosystems, these transforms are so simple they can be communicated verbally: “Replace every letter A with the letter D, B with E, etc.” or “Reverse every pair of letters.”

Modern digital ciphering uses far more intricate transforms, that are additionally dependent upon a second piece of data called a *key*, such that ciphering with even a slightly different key produces a radically different encrypted message. With the right key, the encrypted message can be deciphered; without that key, the message is unrecoverable because the mathematics of recovery without the key are just too hairy. PGP uses this kind of mathematically strong, keyed encryption.

Single-key cryptosystems are too vulnerable

Conventional cryptography uses the same key both to encrypt and to decrypt the data. This does a fine job of job of protecting the message, but there's a basic problem with this scheme: Now you have to protect the key, because anyone who can get the key can recover the message. And since the key has to somehow travel to the recipient in order to be used, there's a potential for a lot of exposure to loss. And in the context of transmitting data over a convenient public carrier– i.e. the internet– the single-key approach is just too vulnerable to rely on.

Key pairs make it possible to exchange keys publicly

PGP cryptography, by contrast, uses *public key cryptography*, in which each user has two separate (but closely related) keys: a decryption key that you must keep secure if you wish to protect your encrypted communications, and an encryption key that you must make available to anyone you want to be able to send you encrypted information. In PGP terminology, the decryption key is known as the *private key* and the encryption key is known as the *public key*; together, they make up your *key pair*. By publishing your public key, you make it possible for anyone in the world to send you material encrypted so that only you can read it; and this innovation for the first time makes crypto convenient enough to be useful in a global internet context.

PGP client software helps you keep your private decryption key secret by, again, encrypting it on your own computer with a passphrase (a long, multi-word password) that you have to provide to access the key. In PGP, it's the passphrase that you personally have to remember, not the key itself.

Why public keys are safe

Due to the characteristics of the mathematics used in the PGP ciphering functions, material encrypted using your public encryption key can only be decrypted with your unique, corresponding private decryption key. Despite the fact that your public encryption key is derived from your private decryption key, the public encryption key is itself strongly encrypted, and it's therefore very difficult—practically impossible—to extract the private key from the public key. This is why your public encryption key can be published freely without giving away your unique decryption ability. Most PGP users leave their public keys in automated public databases called *certificate servers* or *key servers* from which anyone in the world can retrieve them.

Why public keys are useful

This arrangement makes it easy for anyone with PGP software to encrypt material in such a way that only you can decrypt it. Because it's practically impossible to access the information without the private decryption key once information is in the encrypted form, it's safe to use an insecure carrier such as the internet to send the encrypted information.

In other words, publishing your public key gives you the ability to receive information securely from anyone interested in sending you information securely. Or, looking at it from the other side of the transaction, encrypting material using someone else's public key produces data that only they can decrypt.

Importance of being certain of the key owner's identity

As a sender, of course, to be really sure of the security of your PGP-encrypted communication, you would have to be really sure that the public key that you think belongs to your colleague Bob is in fact Bob true public key, and not a public key generated by an impostor seeking to intercept Bob secure communications. Lest that sound too paranoid, let's revisit the part where the sender obtains the recipient's key. In practice, public keys are usually held in automated certificate server programs, rather than handed off directly between trusted people, so the question then becomes: How can you be sure it was Bob who deposited that public key that your certificate server tells you has Bob name on it, and not someone else?

This is where the related cryptographic ideas of *digital certificates* and *digital signatures* enter the picture, along with a host of bigger issues like organization-wide encryption policy—all of which boil down, in the end, to a set of formalized structures for dealing with questions of plain old human *trust*. PGP includes a model for representing trust relationships between parties, and this model informs parts of the API design.

Digital signatures and certificates represent authentication

A digital certificate is a block of data with special properties, appended to a message, that's used to indicate that some trusted third party vouches for the truth of another party's assertion made inside that message. For example, an assertion that a particular public key truly belongs to one particular person.

A typical certificate contains a PGP public key and the proper name and user ID of the key's purported owner, plus the digital signature of at least one other person (or any other kind of authority organization such as a business, department, club, or government agency) who vouches that the public key in the certificate really belongs to the named person. (A PGP certificate can, in fact, contain multiple proper names and user IDs for the person—for example, a person's married name and maiden name, or a given name and stage name.)

A PGP digital signature is a small block of data that a PGP user can append to any message or block of other data (whether encrypted or unencrypted) that encodes both the exact content of the data and their own public key. Once you've used PGP software to validate the digital signature on an incoming message, you can determine the identity of whomever signed it, and you can be sure that the data hasn't been tampered with since it was signed.

You can use functions in the PGP libraries to test whether the signature on the certificate is valid, and if it is, then you know that the public key and the user information in the certificate are intact, and you will have also extracted the public key of whomever signed the certificate.

The PGP API supports multiple standards for signatures, certificates, and message digests.

Trust decisions: Who's doing the authenticating?

Of course, at this point the question becomes: Who is this person that signed the certificate– and is that person anyone you should trust? Because if you're misled into encrypting with the wrong public key, your communication will both be locked to the intended recipient, and open to a different, unknown party who's specifically trying to hijack your intended recipient's secure data. (The process of generating a public key and getting it into a certificate and signed by an authority is deliberate enough to make it practically impossible to produce a misleadingly identified certificate accidentally.)

The answer is: **You have to decide** whose vouching you're willing to trust before any of this will work for you. And just to be clear: the risk is limited to the data you send with this one specific public key only; getting fooled into using the wrong public key doesn't expose any of your other encrypted data or communications. Also, this 'trust' question isn't about trusting the other fellow to be a good person, it's limited to making sure that the public key really belongs to the indicated person. PGP's inventor Phil Zimmerman puts it like this: "Trusting a key is not the same as trusting the key's owner."

PGP provides users a way to record their trust decisions in the form of a local *keyring* file that contains a separate digital certificate for each collected public key. When a user decides to trust a key as authentic, the PGP client software adds the user's own digital signature onto the certificate, and the user is able to encrypt messages for that person's eyes only.

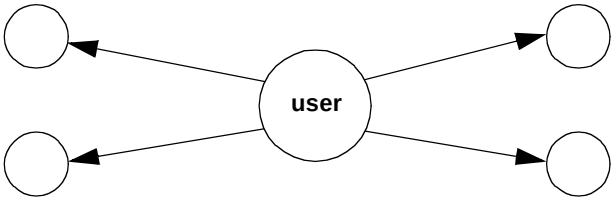

Figure 1-1. Direct trust

Delegating trust decisions

Because people are generally much too busy to actively research every received certificate for trustworthiness, and because organizations sometimes address this problem by delegating this certificate-checking function to particular people or departments—either trusted individuals or, in PGP terminology, institutional *Certification Authorities* (*CAs*). PGP lets you assign each public key on your keyring a *level of trust* as an *introducer* – which is to say, as a signer or voucher of other parties' certificates. This level of trust can be either *complete*, *marginal*, or *untrusted*; PGP will consider as valid the signature of anyone with one 'complete' or at least two 'marginal' signatures on a certificate. This scheme lets you, for example, designate your company CA as a completely *trusted introducer*, so that the CA can do all the authentication work on its set of certificates and you can simply rely on their judgement.

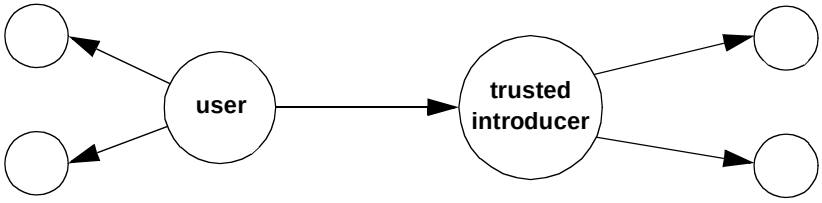

Figure 1-2. Trust via an introducer

Your 'web of trust'

In PGP, the further concept of a *trusted meta-introducer* (a party whom you trust to introduce other parties whom you are willing to also trust) opens the door to the possibility of arbitrarily long chains of signature trust.

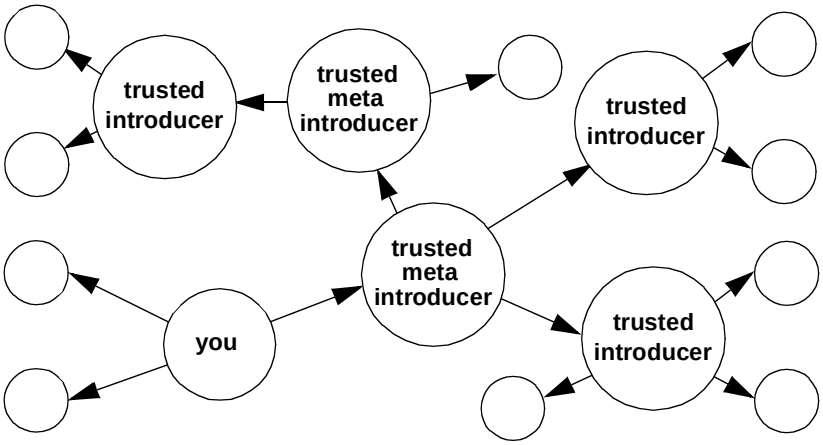

Figure 1-3. Web of trust

We call these sets of chains your *web of trust*, and in the context of the internet, your web of trust is potentially global.

Trust decisions are ultimately the responsibility of each user

All of this trust-checking occurs every time you use any public key to encrypt anything. If you try to encrypt something with an incompletely trusted public key, the PGP client software will alert you – but you always have the option to ignore the warning and use the questionable key anyway.

Recap

In summary: PGP is a public key cryptosystem that supports multiple ciphering, authentication, and certification standards, adds its own decentralized 'web of trust' model, and unifies all these services under a single API which is used both in PGP-branded software from Network Associates and in the PGPsdk.

Core PGP operations

As you can see, the PGP cryptosystem can involve a large number of different entities, operations, and relationships. However, the PGP libraries implement and represent them all through a fairly small number of core operations.

All the variety is produced by combining the following set of core operations in different ways:

- Generating and saving the user's own key pair
- Retrieving the user's key pair from local storage
- Obtaining an intended recipient's public key
- Generating a digital signature (signing a message), with the user's key
- Verifying a received digital signature, with the sender's key
- Encryption for an intended recipient, with their public key
- Decryption of a received message, with the user's private key

The remainder of this User's Guide concentrates on how to implement some of these core operations in your own programs by using the PGP library API.

PGPsdk Organization 2

This chapter uses two sections to explain how the library and headers forming the PGP API are organized:

- **Library and header file organization—**This section shows how the PGP API's library and header files are organized into various subject areas on each supported platform.
- **Subject area overview—**In this section we provide a brief introduction to the services provided in each of those subject areas.

Further information on the organization of the PGP API, including detailed information for each function, can be found in the *PGPsdk Reference Guide*.

Library and header file organization

Depending on the platform, the PGP API ships as two or three libraries. Each library has one or more interface suites, with one C header file per suite. Both the libraries and the headers are broken down by subject area:

Subject area	Header file	Windows library	Unix library	Mac OS library
Option lists	`pgpOptionList.h`	(various)	(various)	(various)
Local key management	`pgpKeys.h`	PGP_SDK.dll	PGPsdk.a	PGPsdkLib
Groups	`pgpGroups.h`			

Ciphering & authentication	`pgpCBC.h`
	`pgpCFC.h`
	`pgpEncode.h`
	`pgpHash.h`
	`pgpHMAC.h`
	`pgpPublicKey.h`
	`pgpSymmetricCipher.h`			

Feature query	`pgpFeatures.h`			

Utilities	`pgpMemoryMgr.h`
	`pgpPubTypes.h`
	`pgpSDKPrefs.h`
	`pgpUtilities.h`
		(various)	(various)	(various)

Random number generation	`pgpRandomPool.h`	PGP_SDK.dll	PGPsdk.a	PGPsdkLib
User interface	`pgpUserInterface.h`	PGPsdkUI.dll	(not available)	PGPsdkUILib
Key server access	`pgpKeyServer.h`	PGPsdkNL.dll	PGPsdkNetworkLib.a	PGPsdkNetworkLib
TLS	`pgpTLS.h`			
Network sockets	`pgpSockets.h`			
Big number management	`pgpBigNum.h`	PGP_SDK.dll	PGPsdk.a	PGPsdkLib
Error codes	`pgpErrors.h` `pgpPFLErrors.h`			

Subject area overview

The PGP API provides a large number of functions and a large number of interface suites. To help break it down into more manageable pieces, you may find it useful to think of the API as being organized in terms of a set of primary PGP operations, plus several categories of secondary, supporting functions:

Primary PGP operations

- **Local key management**
- **Ciphering and authentication**
- **Key server access**

Support Functions

- **PGP API feature query functions**
- **Option list functions**
- **Group functions**
- **Utility toolbox functions**
- **Random number functions**
- **User interface functions**
- **TLS (transport layer security) functions**
- **Network socket functions**
- **Big number management functions**
- **Error functions**

Primary PGP operations

Primary PGP operations are things like encrypting, decrypting, signing, and verifying signatures—the core functionality that drew you to PGP in the first place—and directly related tasks such as getting the keys you need to do them.

Local key management

```
#include "pgpKeys.h"
```

Before you can do any cryptographic operation, you need to get a key. Keys are considered to reside in local ‘key ring’ structures, either on disk or in memory; remote keys are accessed via the key server access functions (see below). A key may have any number of associated sub-keys, additional recipient request keys (ARR), and user IDs; each user ID in turn may have any number of associated signatures.

Local key management operations include:

- Create new key

- Key import and export, via file or buffer
- Get and set key properties (respecting the ‘web of trust’ model which propagates some properties along the web)
- Add key to key ring
- Search for key in keyring
- Remove key from keyring
- Check key validity
- Sign key

For further details on keys and key management, please see Chapter 2 of the *PGPsdk Reference Guide*.

Ciphering and authentication

```
#include "pgpCBC.h"
#include "pgpCFC.h"
#include "pgpEncode.h"
#include "pgpHash.h"
#include "pgpHMAC.h"
#include "pgpPublicKey.h"
#include "pgpSymmetricCipher.h"
```

The PGP API furnishes two kinds of ciphering and authentication services: both high-level and low-level, cryptosystem-specific. Depending on what you want to do, you probably won’t need all the header files listed above.

Because some of these operations are processor-intensive and may take significant amounts of time to execute, a callback mechanism is provided to support both periodical progress-linked tasks (such as animating a progress bar) and more general handling of situations that may arise during cryptographic operations (for example, the need to prompt the user for a passphrase). A set of event codes enumerates these callback conditions, and tells your callback function what to do.

Ciphering and authentication operations include:

- The high-level `PGPEncode()` and `PGPDecode()` functions
- Low-level cipher functions for hashes, HMAC, symmetric cipher, cipher block chaining, cipher feedback blocks, public key, and private key functions

For further details on ciphering and authentication, please see Chapter 5 of the *PGPsdk Reference Guide*.

Key server access

```
#include "pgpKeyServer.h"
```

The PGP API includes facilities for accessing HTTP and LDAP key servers. Because of the communications-based nature of these functions, a thread and callback mechanism is provided, using an enumerated set of event codes to tell your callback procedure what to do.

Key server access operations include:

- Thread storage management functions
- Communication and callback management functions
- Remote key server operations: upload, delete, disable, and search for a key.

For further details on key server access, please see Chapter 10 of the *PGPsdk Reference Guide.*

Support functions

This is setup, cleanup, connectivity, user interface, and other functions that exist only to support the primary functions.

PGP API feature query functions

```
#include "pgpFeatures.h"
```

For a number of reasons including variations in governmental regulation, new feature development, and its long life, there have been lots of different PGP library releases with different feature sets, even at the same version number. For example, some library builds have omitted even the encryption functions. For this reason, a developer accessing the PGP libraries from her code should always use the feature query functions to verify that the needed functions are present in the copy of the PGP library being used.

Feature query operations include:

- Get feature flags, count public key algorithms, count symmetric cipher algorithms, get public key algorithm info, get symmetric cipher algorithm info, and get PGPsdk version.

For further details on API feature query functions, please see Chapter 6 of the *PGPsdk Reference Guide.*

Option list functions

```
#include "pgpOptionList.h"
```

As the "Programming with the PGPsdk" chapter of this manual explains in more detail, several important functions in the API use an 'option list' parameter mechanism. This approach affords the developer a great deal of flexibility in forming the function calls, and avoids the burden of having to fill in large parameter block structures. It does however require the developer to use this set of option list functions to create and populate one opaque object for every desired option. You can think of each function that creates and option object as returning a token for that option.

Further functions support the creation and maintenance of lists of these option tokens, permitting collections of options to be used multiple times once built. For example, when encrypting every file on a given volume, all encryption function options will remain the same except for the input file and the output file; the rest of the options could be kept in a persistent option list.

Option list operations include:

- Functions to create and manage lists of option tokens: new, build, copy, append, and free.
- More than 80 functions that take option parameters (if any) and return option tokens. For example: specifying an output file, choosing a ciphering algorithm, setting a title for a GUI window, etc.

For further details on option list functions, please see Chapter 3 of the *PGPsdk Reference Guide*, and the "Programming with the PGPsdk" chapter of this manual.

Group functions

```
#include "pgpGroups.h"
```

For easier management, key IDs may be kept in named lists, called groups. Key IDs are not the same as keys and cannot be used in cryptographic operations directly, so the PGP API provides functions to resolve a group into the set of keys its key IDs represent. Group structures may be hierarchical; that is, one group may contain another group. Groups are kept in group sets.

Group operations include:

- Group set management functions such as new, get, copy, merge, sort, export, import, count, and free.
- Group management functions such as new, delete, add item, set, sort, and count.
- Group item iterator functions such as new, next, and free.

- Group utility functions such as obtaining the referenced set of keys and obtaining a flattened group from a hierarchical group.

For further details on group functions, please see Chapter 4 of the *PGPsdk Reference Guide*.

Utility toolbox functions

```
#include "pgpMemoryMgr.h"
#include "pgpPubTypes.h"
#include "pgpSDKPrefs.h"
#include "pgpUtilities.h"
```

The utility toolbox supports the use of a number of basic PGP-unique cross-platform resources and other mechanisms.

Utility operations include:

- PGPsdk management and preference functions, including management of subsets such as the memory manager and networking APIs
- Functions to create and manage opaque objects and structures such as Contexts, file specifications, and dates and times

For further details on utility toolbox functions, please see Chapter 7 of the *PGPsdk Reference Guide*.

Random number functions

```
#include "pgpRandomPool.h"
```

PGP's ciphering services require access to high-quality random numbers, and that implies an infrastructure for creating and managing them. So PGP supports a 'pool' of random numbers, and facilities for the acquisition of random data of various kinds.

Random number operations include:

- Functions to manage the pool of random numbers
- Functions supporting the collection of entropy for use in forming new random numbers

For further details on random number functions, please see Chapter 8 of the *PGPsdk Reference Guide*.

User interface functions

```
#include "pgpUserInterface.h"
```

The PGPsdk allows developers to access the same user interface toolkit that the PGP-brand software from Network Associates uses. The toolkit includes customizable dialogs for several common operations such as entering passphrases for various purposes, selecting keys to encrypt to, working with a remote key server, and so forth. A set of options supports custom dialog items.

User interface operations include:

- User interface management functions
- PGP dialog functions
- PGPOptions for custom dialog items

For further details on user interface functions, please see Chapter 9 of the *PGPsdk Reference Guide*.

TLS (transport layer security) functions

```
#include "pgpTLS.h"
```

In a TLS communication connection, dataflow is broken into blocks and each transmitted block is signed by the sending machine. By verifying the signature on each block and rejecting blocks whose signatures don't verify, the receiving machine can be sure that the received data is intact and truly originated on the sending machine (as opposed to an impostor). These PGPsdk functions provide access to the low-level TLS functions that PGP uses when communicating with TLS-capable remote key servers.

TLS operations include:

- Create, manage, and free TLS contexts
- Create, configure, manage, and free TLS sessions
- Attach a socket to a TLS session

For further details on TLS functions, please see Chapter 11 of the *PGPsdk Reference Guide*.

Network socket functions

```
#include "pgpSockets.h"
```

The PGPsdk includes a simple platform-independent abstraction layer for internet stream and datagram sockets. Note that associating a TLS session with this socket layer results in an unusually easy-to-use, secure communication package.

Socket operations include:

- Create, manage, and delete sockets
- Thread storage management
- Socket listen, bind, and connect
- Data send and send to
- Data receive and receive from
- Net-to-native byte ordering translations

For further details on network socket functions, please see Chapter 12 of the *PGPsdk Reference Guide.*

Big number management functions

```
#include "pgpBigNum.h"
```

Generating keys and ciphering data with them involves math operations using integers much larger than standard C compilers support. So the PGPsdk includes several data types and functions for working with these 'big numbers.'

Big number operations include:

- Create, copy, and free `BigNum` structures
- Math functions that take `BigNum`s as operands
- Unary operator functions, such as shifts, for `BigNum`s
- Comparison operator functions for `BigNum`s
- Type translation and assignment functions
- Functions to access subranges of a `BigNum`

For further details on big number management functions, please see Chapter 13 of the *PGPsdk Reference Guide.*

Error functions

```
#include "pgpErrors.h"
#include "pgpPFLErrors.h"
```

The PGPsdk uses a single error code space, and provides a function to translate an error code number into a text string explaining the error.

Error operations include:

- Translate error code to text string.

For further details on PGP SDK errors, please see Appendix A of the *PGPsdk Reference Guide.*

Programming with the PGPsdk 3

This Chapter covers general topics related to programming with the PGP API:

- **Adding the PGP library and interfaces to your project**
- **Programming style**
- **Working with PGP's opaque data types**
- **The PGPContext structure**
- **Working with option lists**
- **Key management concepts**
- **Callback and event concepts**
- **Platform-Specific Issues**

These topics lay the groundwork for the more specific coverage of PGP operations and functions that appears in Chapter 4, "Implementing Common PGP Operations."

Working with PGP's opaque data types

For reasons of encapsulation and reliability across PGP library releases, many important data structures used in the PGP API are presented in the interfaces in terms of opaque data references. The functions that create these structures return references to them, and you then manipulate and use them only by passing the references to other PGP API functions– never by accessing the members of the data structures directly.

For each of these data types, existence and memory management are handled by a 'New' and a 'Free' function, and a `RefIsValid` macro is supplied so you can avoid attempts to operate on bad references.

Important opaque data types include:

`PGPContext`	A PGPsdk operating context
`PGPOptionListRef`	An entry in a list of options for a function
`PGPKeySetRef`	A collection of one or more keys

For further details on these opaque data types, please see Chapter 1 of the *PGPsdk Reference Guide*.

The PGPContext structure

Probably the most prominent of the PGP API's opaque data types is `PGPContext`. Nearly all significant PGP operations require a valid `PGPContextRef`, either directly or indirectly as a member of a further opaque data type reference.

The `PGPContext` structure is opaque for a reason, so we won't detail its contents here. What's important to know is that you have to create one early in your program, after calling `PGPsdkInit()` but before doing anything significant with the PGPsdk; and you have to free the context late in your program, after calling `PGPsdkCleanup()` and just before exiting.

```
PGPContextRef context = NULL;

/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
goto Exit;
```

For further details on the `PGPContextRef` and `PGPContext` structures, please see Chapter 1 of the *PGPsdk Reference Guide.*

Working with option lists in PGPsdk function calls

For many of the PGPsdk functions, the number of parameters a given developer might want to use is variable, and the whole set of available parameters is too large to use as conventionally-typed C function call parameters. In fact, it's even too large to keep in a standardized parameter block structure– too much space would be wasted on average. Our solution is to use an Option List technique in the parameter list of these function calls, and this requires developers to work with a C programming idiom they may not have encountered before.

Let's look at the prototype for one of the functions that uses an option list:

```
PGPError PGPEncode(PGPContextRef pgpContext,
           PGPOptionListRef firstOption, ...);
```

Note that the function prototype allows for any number of arguments (“. . .”). To use an Option List, you pass the function an arbitrary number of PGPOptionListRef values, one per option that you wish to specify. You produce each PGPOptionListRef by calling a function appropriate to the parameter you want to provide. Here’s an example of a real-world call to PGPEncode(), replacing the ‘. . .’ with real options:

```
PGPEncode(context,
        PGPOEncryptToKeySet( context, foundUserKeys ),
        PGPOInputFile( context, inFileRef ),
        PGPOOutputFile( context, outFileRef ),
        PGPOLastOption( context ) );
```

When called, the function simply pops the PGPOptionListRef’s off the stack, one at a time, and interprets each one in turn. This mechanism allows all of the formal parameters to have the same type, and that’s what lets you as a user of the function to choose your own set of options, and to supply them in whatever order you prefer.

Note that in this form, an option list is a list of options in the source code, and a sequence of data items on the stack, but as far as your program is concerned, it’s not a free-standing data structure.

How does the called function know where to stop popping PGPOptionRef’s off the stack? It looks for an end-of-list token. That’s why you must terminate your option list with the special token that the function PGPOLastOption() returns.

Creating PGPOptionListRefs

Note that because PGPOptionListRef is one of PGP’s opaque types, each PGPOptionListRef value in the option list should be obtained only by a call to the option list function corresponding to your desired parameter. These functions all have names of the form PGPONameGoesHere(), where NameGoesHere is the name of the option.

For example, as the previous code fragment illustrates, you’ll use the function PGPOInputFile() to specify the input file for many operations: you pass PGPOInputFile() a PGPContextRef and the ID of the file containing your input data, and then use the PGPOptionListRef it returns in the option list for your desired operation.

Persistent option lists

A similar mechanism supports the creation of separate option list data structures in memory, as distinct from option lists used in function parameters. We also call these data structure-based option lists *persistent* option lists in some of our documentation. This mechanism allows you to build an option list just once and then use it multiple times, which is sometimes helpful and sometimes necessary, as some PGPsdk functions take option list pointers as parameters.

To create such an option list in memory, call the function `PGPNewOptionList()`, and then populate the list with `PGPOptionListRef` items at your convenience, using either `PGPBuildOptionList()` or `PGPAppendOptionList()`. Persistent option lists must be released with `PGPFreeOptionList()` once you're done with them.

For more details on option list functions, please see Chapter 3 of the *PGPsdk Reference Guide.*

Key management concepts

The PGP library can transparently maintain a structured, distributed database of available public and private keys, and provides functions to query the database, to obtain lists of keys matching your search criteria. This key database may span any number of files, each of which may be either a local disk file, or a remote file managed by a remote key server programs, or even a local transient, memory-only file.

All accesses to key databases return lists of keys. You can get a list of your user's own public and private keys with the function `PGPOpenDefaultKeyRings()`. Such a list of keys is called a *key set*, and can be kept in an opaque `PGPKeySetRef` object. You can obtain keys matching any given search criteria by creating a search filter to look for those criteria and then requesting a search be performed on an indicated key database, or on a remote key server.

Once you've obtained a key set, you can sort them according to a number of different criteria if you wish, and an *iterator* mechanism for walking a key set is available. Reasons to iterate over a key list might include wanting to modify all the keys in your list, or, once you've sorted your list into your preference order, wanting to try to decrypt a message with each of the keys in turn, in your own order of preference.

Looser, unfiltered collections of keys (such as you may want to keep in disk files) called *groups* are also supported by a separate family of flexible functions (see Chapter 4 of the *PGPsdk Reference Guide*). You can sort, index into, count, name, iterate across, and describe a group. A group can contain other groups, and groups can be kept either in a file or in memory; and you can produce a key set from a group, for use with the many functions that operate on key sets rather than groups.

For further details on the structure of keys and on key management operations, please see Chapters 1 and 2 of the *PGPsdk Reference Guide.*

Callback and event concepts

Some major PGPsdk functions, notably `PGPDecode()`, allow you to provide a callback handler function pointer parameter. If you provide such a callback handler, then the PGPsdk will call your function whenever any of a predefined set of conditions arises in the course of processing the requested operation (i.e. during the `PGPDecode()`). Your function receives an event code parameter that identifies the condition that triggered the callback. Symbols for these event codes start with `kPGPEvent_`, and appear in `pgpEncode.h`.

For example, one of the things that can occur while decrypting a message with `PGPDecode()` is that PGP may find that a passphrase is needed to authorize access to the required private key. This will cause the PGP code to call your callback handler with an event code of `kPGPEvent_PassphraseEvent`. Knowing this, you can write your event handler to put up an appropriate dialog to obtain a passphrase from the user when it receives that event code. For other events that flag a missing piece of information needed for an operation, you can use the function `PGPAddJobOptions()` to supply it.

Typically a callback event handler will be written to accommodate and respond to several different event codes.

The event codes used in each subject area of the PGPsdk API are detailed near the start of each subject area's chapter in the *PGPsdk Reference Guide.*

Implementing Common PGP Operations 4

Nothing explains how to use a library quite like seeing some working code. In this chapter we'll present sample code fragments that illustrate how to implement a number of basic PGP high-level operations:

- **Encrypting a file**
- **Decrypting a file**
- **Signing a file**
- **Verifying a signature on a file**

In the process of demonstrating how to achieve these high-level outcomes, we'll also articulate the supporting low-level PGP steps that you'll need to be able to take in your own code:

- **Initializing PGP and setting up a PGPContext**
- **Setting up an option list**
- **Obtaining a public key from the local keyrings**
- **Obtaining a private key from the local keyring**
- **Ciphering with a public key**
- **Deciphering with a private key**

Our first example, and the most basic one, teaches how to encrypt a text file.

Encrypting a file

Encrypting a message begins with the recipient's public key and the message to be encrypted, and ends with a PGP-encrypted message that can be safely sent by means of an insecure carrier. (Recall that you have to obtain the recipient's public key in order to PGP-encrypt a message for the recipient's eyes only.)

This section demonstrates how to encrypt a file on disk, producing a second, encrypted disk file.

Overview

File encryption is accomplished by calling `PGPEncode()` with a set of options specifying the input file, the output file, and the key(s) to which you wish to encrypt. The code presented in this example demonstrates how to:

- Initialize the PGPsdk library
- Develop the data and create the resources needed to form the options
- Call `PGPEncode()` to perform the encryption
- Free the resources you created
- and finally, shut down the PGPsdk library.

We'll step through the code, explaining one chunk at a time. For context, a full listing of the example code appears at the end of this section.

Setup

The basic setup code for a simple encryption operation looks something like this:

```
/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;
```

Initializing the PGPsdk library

A basic rule or programming with the PGPsdk is that you must always call `PGPsdkInit()` once to initialize the PGPsdk library before you attempt do anything else with PGP. Note that you must also match this call with a call to `PGPsdkCleanup()` before exiting your program.

Also note that we present these function calls within this encryption example for clarity only; in a real-world program you'd make these calls at application startup and shutdown time, not on a per-PGP-operation basis.

Creating a PGPContext

Most interesting PGPsdk functions require a valid `PGPContextRef` parameter, so you also have to call `PGPNewContext()` before you can get very far. Note that you must match this call with a call to `PGPFreeContext()` before exiting.

Again, for most real-world applications only a single context is required for the whole application, so `PGPNewContext()` and `PGPFreeContext()` would ordinarily appear at application startup and shutdown time, rather than being called on a per-PGP-operation basis.

Obtaining the recipient's public key

The next section of code retrieves the public key for the PGP user called `'test@pgp.com'` from your user's local default keyrings, and turns it into a key set that you'll use in one of the options you'll pass to `PGPEncode()`:

```
char const *userID = "test@pgp.com";

/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
           &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;

/* Create a filter to look for the default "test@pgp.com" key */
err = PGPNewUserIDStringFilter( context, userID,
           kPGPMatchSubString, &filter );
if( IsPGPError( err ) )
    goto Exit;

/* Look for the key */
err = PGPFilterKeySet( defaultKeyRing, filter, &foundUserKeys );
if( IsPGPError( err ) )
    goto Exit;
```

Performing the key search

To get the key for `'test@pgp.com'`, you have to search your user's default keyring.

Before you can perform the key search, you first need to enable access to your user's default keyrings with `PGPOpenDefaultKeyRings()`. Then you need to create a search filter that matches for `'test@pgp.com'`, using `PGPNewUserIDStringFilter()`. At that point you can perform the actual search by calling `PGPFilterKeySet()`, which produces a key set reference that holds all the keys that satisfy your search criteria. We're calling that key set `foundUserKeys`.

Note that the result is typed as a set of keys, rather than as just a single key, because your filter may find more than one key. You can use the function `PGPCountKeys()` on the search result key set to see how many keys the filter found for you, for example to help decide whether you need to present them to the user as options to choose from. For clarity in this simple example, because we expect there to be just the one `'test@pgp.com'` key, we've left all that out for now.

Creating the input and output file references

Specifying the input and output files is similar, but simpler:

```
char const *inFileName    = "C:\testPlainText";
char const *outFileName   = "C:\testEncrypted";

PGPFileSpecRef inFileRef      = kInvalidPGPFileSpecRef;
PGPFileSpecRef outFileRef     = kInvalidPGPFileSpecRef;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, inFileName, &inFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, outFileName, &outFileRef );
if( IsPGPError( err ) )
    goto Exit;
```

When you make your call to `PGPEncode()`, you'll need to specify the input and output files in terms of `PGPFileSpecRef` values, so you need to translate your filename strings into that form first. Fortunately, the PGPsdk provides a function to turn a file path string into a `PGPFileSpecRef`. It's called `PGPNewFileSpecFromFullPath()`, and you need to use it to make one `PGPFileSpecRef` for the input file, and another for the output file.

Mac OS developers should note that this process works differently on that platform, and that for clarity we've omitted the Mac OS version in this example. You would use the function `PGPNewFileSpecFromFSSpec()` instead, which in keeping with the Mac OS file system model takes an `FSSpec*` instead of a `char*` path as the second parameter. Setting up the `FSSpec` is your responsibility, though.

Forming options and ciphering the message

At this point, all the hard work's over. Now you just need to transform your three encryption parameters into the form of PGP options, and provide them in your call to PGPEncode():

```
/* This is the main event; everything above and
   below exists only to support this call */

err = PGPEncode( context,
           PGPOEncryptToKeySet( context, foundUserKeys ),
           PGPOInputFile( context, inFileRef ),
           PGPOOutputFile( context, outFileRef ),
           PGPOLastOption( context ) );
```

Notice that the parameters to PGPEncode(), after the context, form a list of PGP option functions, terminated by the PGPOLastOption(context). This is a common idiom in PGPsdk programming that you should become comfortable with; see Chapter 3, "Programming with the PGPsdk," for an introduction to option lists.

Each of the calls PGPOEncryptToKeySet(), PGPOInputFile(), and PGPOOutputFile() creates an opaque option data structure and returns an opaque PGPOptionRef that refers to it:

- PGPOEncryptToKeySet(context, foundUserKeys) tells PGPEncode() to encrypt the input file to the key(s) that your search returned;
- PGPOInputFile(context, inFileRef) tells PGPEncode() to read the file you specified as the input source;
- PGPOOututFile(context, outFileRef) tells PGPEncode() to write the encrypted version of the input file to the output file you specified.

These three calls form a simple option list directly in the call to PGPEncode(). For future reference, note that a couple other option list strategies are possible, including creating a separate PGPOptionList structure and adding PGPOptionListRefs to it as desired, and finally passing the whole option list in as a PGPOptionList parameter to PGPEncode().

Calling PGPEncode()

To perform the actual encryption operation, you just call PGPEncode() as shown. If all the option parameters were set correctly, this will result in the creation of an encrypted output file with the name you specified.

Cleanup

Before you can safely exit, you'll need to dispose of whatever PGPsdk resources you caused to be created, and shut down the PGPsdk library:

```
/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( inFileRef ) )
    PGPFreeFileSpec( inFileRef );

if( PGPFileSpecRefIsValid( outFileRef ) )
    PGPFreeFileSpec( outFileRef );

if( PGPFilterRefIsValid( filter ) )
    PGPFreeFilter( filter );

if( PGPKeySetRefIsValid( foundUserKeys ) )
    PGPFreeKeySet( foundUserKeys );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );

/* ---- PGP Shutdown ----------------------------------------- */

/* Release the PGP context we've been using */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();
```

The PGPsdk library is responsible for managing whatever storage is required to support its opaque data types, but it needs your help in order to know when it's time to free that storage. In this example we've used several opaque types, and so before quitting we have to free up two file specs, a filter, and two key sets.

PGP shutdown

We also have to free up the `PGPContext` we've been using, and lastly, you should always balance your `PGPsdkInit()` call with a call to `PGPsdkCleanup()` before exiting your program.

As with the corresponding setup operations (`PGPsdkInit()` and `PGPNewContext()`), note that we present these two calls within this encryption example for clarity only; in a real-world program you'd ordinarily make these calls at application startup and shutdown time, not on a per-PGP-operation basis.

Full listing: Encrypting a file

If we pull all of the above snippets together, reorganize them rationally, and add some better comments, we get this full listing:

```
/* ==== Encrypt a File code fragment ========================== */
/*
 *  Demonstrates simple file encryption with the PGPsdk.
 *
 *  Encrypts file "C:\testPlainText" with key "test@pgp.com",
 *   producing encrypted file "C:\testEncrypted".
 *
 * ------------------------------------------------------------
 *   More precisely:
 *
 *  The userID key has to be available in the default keyring;
 *   for simplicity, this example uses a key with the user ID
 *   'test@pgp.com'.
 *
 *  Overwrites any existing output file with the same name.
 * ------------------------------------------------------------
 */

/* Specify the key, input filename, and output filename to use */
char const *userID        = "test@pgp.com";
char const *inFileName    = "C:\testPlainText";
char const *outFileName   = "C:\testEncrypted";

/* Declare and initialize variables */
PGPError       err            = kPGPError_NoErr;
PGPContextRef  context        = kInvalidPGPContextRef;
PGPKeySetRef   defaultKeyRing = kInvalidPGPKeySetRef;
PGPFilterRef   filter         = kInvalidPGPFilterRef;
PGPKeySetRef   foundUserKeys  = kInvalidPGPKeySetRef;
PGPFileSpecRef inFileRef      = kInvalidPGPFileSpecRef;
PGPFileSpecRef outFileRef     = kInvalidPGPFileSpecRef;


/* ---- PGP Setup ------------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would call PGPsdkInit()
 * and PGPNewContext() at application startup time, not on
 * an operation-by-operation basis.
 */

/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
```

```
    goto Exit;

/* ---- Prepare Encryption Operation Parameters ------------- */

/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
          &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;

/* Create a filter to look for the default "test@pgp.com" key */
err = PGPNewUserIDStringFilter( context, userID,
          kPGPMatchSubString, &filter );
if( IsPGPError( err ) )
    goto Exit;

/* Look for the key */
err = PGPFilterKeySet( defaultKeyRing, filter, &foundUserKeys );
if( IsPGPError( err ) )
    goto Exit;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, inFileName, &inFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, outFileName, &outFileRef );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Perform the Encryption operation ------------- */

/* This is the main event; everything above and
 * below exists only to support this call.
 *
 * Produces the encrypted output file in the default directory.
 */

err = PGPEncode( context,
          PGPOEncryptToKeySet( context, foundUserKeys ),
          PGPOInputFile( context, inFileRef ),
          PGPOOutputFile( context, outFileRef ),
          PGPOLastOption( context ) );

Exit:

/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( inFileRef ) )
    PGPFreeFileSpec( inFileRef );

if( PGPFileSpecRefIsValid( outFileRef ) )
    PGPFreeFileSpec( outFileRef );

if( PGPFilterRefIsValid( filter ) )
```

```
    PGPFreeFilter( filter );

if( PGPKeySetRefIsValid( foundUserKeys ) )
    PGPFreeKeySet( foundUserKeys );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );

/* ---- PGP Shutdown ---------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would make these calls to
 * PGPFreeContext() and PGPsdkInit() at application shutdown time,
 * and not on an operation-by-operation basis.
 */

/* Release the PGP context we've been using */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();

/* ========== End of Encrypt a File code fragment ========== */
```

Decrypting a file

Decrypting a message meant for your program's user begins with the received encrypted message, and with your user's private key waiting on your user's computer; decryption ends with the recovered decrypted message.

Recall that you have to be able to access your user's private key in order to decrypt a PGP-encrypted message. This is another way of saying that a person who wants to encrypt a message for your user's eyes only must encrypt to the public key that uniquely corresponds to your user's true private key.

This section demonstrates how to decrypt an encrypted disk file, producing a second, unencrypted disk file. It happens to use as input the same file that the previous example, "Encrypting a file" produced. You'll notice that the code here is even simpler than the encryption example was.

Overview

File decryption is accomplished by calling `PGPDecode()` with a set of options specifying the input file, the output file, and the key set containing your user's private key(s). Building on the "Encrypting a file" section, the code presented in this example demonstrates how to:

- Initialize the PGPsdk library
- Develop the data and create the resources needed to form the options
- Call `PGPDecode()` to perform the decryption
- Free the resources you created
- and finally, shut down the PGPsdk library.

Again, we'll step through the code, explaining one chunk at a time.

For context, a full listing of the example code appears at the end of this section.

Setup

Setup code for a simple decryption operation is the same as for encryption:

```
/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;
```

Initializing the PGPsdk library

Note that we present these function calls within this example for clarity only; in a real-world program you'd make these calls at application startup and shutdown time, not on a per-PGP-operation basis. For further explanation, please refer to the **Encrypt a file** example.

Creating the input and output file references

This too is very similar to the encryption example:

```
/* Specify the input filename, output filename  */
char const *inFileName    = "C:\testEncrypted";
char const *outFileName   = "C:\testDecrypted";

/* Declare and initialize variables */
PGPFileSpecRef inFileRef      = kInvalidPGPFileSpecRef;
PGPFileSpecRef outFileRef     = kInvalidPGPFileSpecRef;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, inFileName, &inFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, outFileName, &outFileRef );
if( IsPGPError( err ) )
    goto Exit;
```

You'll have noticed that the only difference from the "Encrypting a file" example is the exact filenames used. This consistency is one of the benefits of the option list strategy– once you know how to specify the input and output files for one PGPsdk operation, you know how to do it for almost every other operation in the whole PGPsdk.

Mac OS developers should refer to the "Encrypting a file" example for an explanation of how file references are handled differently on that platform.

Providing access to your user's private key

For decryption, we need to provide PGPDecode() with access to our user's private key:

```
/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
          &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;
```

Passphrases control access to private decryption keys

Decrypting a message requires access to the recipient's correct private key. Because of the PGPsdk's flexible and modular approach to key management, it's your responsibility when writing PGPsdk decryption code to provide `PGPDecode()` with access to your user's private key(s). These private keys are typically stored in the user's default keyrings. Although the default keyrings may contain more than key, the decryption process extracts the public key that was used to encrypt the message, and that enables `PGPEncode()` to pick the uniquely corresponding private key.

Because private keys are the single most important secret in the PGP public key security scheme, access to them is carefully guarded throughout the PGPsdk. Your decryption code cannot access a private key without feeding `PGPDecode()` the key's passphrase.

For clarity in this example we expose the passphrase in plain text in the source code, but please note that from a security standpoint this is just a terrible practice – anyone with a debugger could read the passphrase simply by inspecting your executable file, and there goes your security integrity, right out the window. In general, you'll want to keep any storage of passphrases as transient and disguised as possible. In most real-world applications of PGP, you'd instead furnish a user interface dialog to ask your user for the passphrase. (See also the note regarding event callback functions below, under "Full listing: Signing a file")

Forming options and deciphering the message

Again, all the hard work's over at this point and you just need to transform your encryption parameters into PGP options and feed them to `PGPDecode()`:

```
err = PGPDecode( context,
            PGPOInputFile( context, inFileRef ),
            PGPOOutputFile( context, outFileRef ),
            PGPOKeySetRef( context, defaultKeyRing ),
            PGPOPassphrase( context, keyPassphrase ),
            PGPOLastOption( context ) );
```

Just as for `PGPEncode()`, the parameters to `PGPDecode()` after the `context` form a list of PGP option functions, terminated by the `PGPOLastOption( context )`. Again, this is a form that, having learned it once, you'll use again and again in PGPsdk programming.

To run down the option parameters:

- `PGPOInputFile( context, inFileRef )` tells `PGPDecode()` to read the file you specified as the input source;

- `PGPOOututFile( context, outFileRef )` tells `PGPDecode()` to write the encrypted version of the input file to the output file you specified;
- `PGPOKeySetRef( context, defaultKeyRing )` feeds the user's default keys to `PGPDecode()` so it can search for the right decryption key there;
- `PGPOPassphrase( context, keyPassphrase )` furnishes `PGPDecode()` with the passphrase needed to access our test key; ordinarily you'd have set `keyPassphrase` from a user dialog.

Calling PGPDecode()

To perform the actual decryption operation, you just call `PGPDecode()` as shown. If all the option parameters were set correctly, this will result in the creation of a decrypted output file with the name you specified.

Cleanup

Cleanup too is very similar to the encryption example:

```
/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( inFileRef ) )
    PGPFreeFileSpec( inFileRef );

if( PGPFileSpecRefIsValid( outFileRef ) )
    PGPFreeFileSpec( outFileRef );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );

/* ---- PGP Shutdown ----------------------------------------- */

/* Release the PGP context we've been using */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();
```

Again, we present the 'PGP Shutdown' calls within this example for clarity only; in a real-world program you'd ordinarily make these calls at application shutdown time, not on a per-PGP-operation basis.

What we left out

Event handler function

As you know, the PGPsdk provides a callback event handling mechanism for some operations. Decrypting a file with `PGPDecode()` is probably the most common use for an event handler, because in the course of decrypting the message any number of unpredictable conditions can arise that need to be handled adaptively.

For example, most users have more than one private key, but because the encryption public key isn't recovered until `PGPDecode()` has started working, your code won't know what passphrase to use at the time of your call to `PGPDecode()`. This is why one of the callback event codes is `kPGPEvent_PassphraseEvent`. If your callback function receives that code, you need to take steps to acquire the needed passphrase, supply it to the PGPsdk in the form of a standard PGP option with the call `PGPAddJobOptions()`, and return.

To supply a simple callback event handler called `myEventHandler()` for `PGPDecode()`, you'd add something like the following option to your `PGPDecode()` call:

```
    /* Use myEventHandler for event callbacks */
    PGPOEventHandler( context, myEventHandler, &myState ),
```

A very simple example of a callback handler that deals only with the `kPGPEvent_PassphraseEvent` might look like this:

```
static PGPError myEventHandler(
    PGPContextRefcontext,
    PGPEvent*event,
    PGPUserValueuserValue)
{
    PGPErrorerr = kPGPError_NoErr;

    if( event->type == kPGPEvent_PassphraseEvent )
    {
        PGPEventPassphraseData *d = &event->data.passphraseData;
        char passphrase[256];

        passphrase[0] = 0;

        if( d->fConventional )
        {
            /* Prompt for a conventional passphrase here.
                See pgpUserInterface.h */
        }
        else
        {
```

```
            /* Prompt for a decryption passphrase here.
                See pgpUserInterface.h */
        }

        err = PGPAddJobOptions( event->job,
                    PGPOPassphraseBuffer( context, passphrase,
                        strlen( passphrase ) ),
                    PGPOLastOption( context ) );
    }

    return err;
}
```

Typically an event handler would be much more extensive than this one; over twenty event codes are currently defined. Note that the previous example ”Encrypting a file” could probably have used an event handler function too.

For further information on event callback functions, please see “Callback and event concepts” on page 31.

Full listing: Decrypting a file

Again reorganizing the previous snippets and improving the comments, we get this full listing:

```
/* ==== Decrypt a File code fragment ========================== */
/*
 *  Demonstrates simple file decryption with the PGPsdk.
 *
 *  Decrypts file "C:\testEncrypted" encrypted with key
 *  "test@pgp.com", producing encrypted file "C:\testDecrypted".
 *
 */

/* Specify the input filename, output filename,
    and key passphrase to use */
char const *inFileName    = "C:\testEncrypted";
char const *outFileName   = "C:\testDecrypted";
char const *keyPassphrase = "testPassphrase";

/* Declare and initialize variables */
PGPError       err            = kPGPError_NoErr;
PGPContextRef  context        = kInvalidPGPContextRef;
PGPKeySetRef   defaultKeyRing = kInvalidPGPKeySetRef;
PGPFileSpecRef inFileRef      = kInvalidPGPFileSpecRef;
PGPFileSpecRef outFileRef     = kInvalidPGPFileSpecRef;

/* ---- PGP Setup ------------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would call PGPsdkInit()
 * and PGPNewContext() at application startup time, not on
 * an operation-by-operation basis.
 */

/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Prepare Decryption Operation Parameters ------------- */

/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
           &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;
```

```
/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, inFileName, &inFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, outFileName, &outFileRef );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Perform the Decryption operation ------------- */

/* This is the main event; everything above and
 * below exists only to support this call.
 *
 * Produces the decrypted output file in the default directory.
 */

err = PGPDecode( context,
           PGPOInputFile( context, inFileRef ),
           PGPOOutputFile( context, outFileRef ),
           PGPOKeySetRef( context, defaultKeyRing ),
           PGPOPassphrase( context, keyPassphrase ),
           PGPOLastOption( context ) );

Exit:

/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( inFileRef ) )
    PGPFreeFileSpec( inFileRef );

if( PGPFileSpecRefIsValid( outFileRef ) )
    PGPFreeFileSpec( outFileRef );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );

/* ---- PGP Shutdown ---------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would make these calls to
 * PGPFreeContext() and PGPsdkInit() at application shutdown time,
 * and not on an operation-by-operation basis.
 */

/* Release the PGP context we've been using */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();

/* ========== End of Decrypt a File code fragment ========== */
```

Signing a file

In PGP, signing a message begins with your own public key and a message to be signed, and ends with a digital signature that you typically append to the message.

As with decryption, you have to be able to access your user's private key in order to sign a message.

This section demonstrates how to sign a disk file, producing a second disk file containing the signature. As you'll notice, signing is more complicated than the previous encryption and decryption examples.

Overview

File signing is accomplished by calling `PGPEncode()` with a set of options specifying the input file, the output file, and the key to sign with and its passphrase; a separate option asks for the signature to be created in a 'detached' file of its own. The code presented in this example demonstrates how to:

- Initialize the PGPsdk library
- Develop the data and create the resources needed to form the options
- Call `PGPEncode()` to perform the signing operation
- Free the resources you created
- and finally, shut down the PGPsdk library.

As before, we'll go over the code one fragment at a time. For more context, a full listing of the example code appears at the end of this section.

Setup

Our setup code is once again the same:

```
/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;
```

Creating the input and output file references

Once again, only the names have changed, and Mac OS would be handled differently:

```
/* Specify the input and output filenames  */
char const *inFileName    = "C:\testIn";
char const *outFileName   = "C:\testSignature";

/* Declare and initialize variables */
PGPKeyIterRef  keyListIterator= kInvalidPGPKeyIterRef;
PGPKeyRef      signingKey     = kInvalidPGPKeyRef;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, inFileName, &inFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, outFileName, &outFileRef );
if( IsPGPError( err ) )
    goto Exit;
```

Accessing your user's private key

This involves a filter and a key search to narrow down the default key ring to one particular key to sign with, very similar to what we did in the "Encrypting a file" example:

```
/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
          &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;

/* Create a filter to look for the default "test@pgp.com" key */
err = PGPNewUserIDStringFilter( context, userID,
          kPGPMatchSubString, &filter );
if( IsPGPError( err ) )
    goto Exit;

/* Look for the key */
err = PGPFilterKeySet( defaultKeyRing, filter, &foundUserKeys );
if( IsPGPError( err ) )
    goto Exit;

/* use the first matching key */
err = PGPOrderKeySet( foundUserKeys, kPGPAnyOrdering,
          &foundKeysList );

err = PGPNewKeyIter( foundKeysList, &keyListIterator );
if( IsPGPError( err ) )
    goto Exit;
err = PGPKeyIterNext( keyListIterator, &signingKey );
```

```
if( IsPGPError( err ) )
    goto Exit;
```

Forming options and signing the message

Again, you need to transform your encryption parameters into PGP options and feed them to `PGPEncode()`:

```
char const *keyPassphrase = "testPassphrase";

err = PGPEncode( context,
            PGPOArmorOutput( context, TRUE ),
            PGPODetachedSig( context,
                /* PGPODetachedSig uses its own option list */
                PGPOLastOption( context ) ),
            PGPOInputFile( context, inFileRef ),
            PGPOOutputFile( context, outFileRef ),
            PGPOSignWithKey( context, signingKey,
                /* PGPOSignWithKey uses its own option list */
                PGPOPassphrase( context, keyPassphrase ),
                PGPOLastOption( context ) ),
            PGPOLastOption( context ) );
```

To run down the option list:

- `PGPOArmorOutput( context, TRUE )` tells `PGPEncode()` to format the signature in the following familiar format, which is designed for email transmission and makes clear what the signature is:

  ```
  -----BEGIN PGP SIGNATURE-----
  Version: PGPsdk version 1.7.1
  (C) 1997-1999 Network Associates, Inc.
  and its affiliated companies.

  iQA/AwUAN03o8g/fXvglvtELEQLk2wCeL+GSNcDQZ31xTu2iGDfh
  JcyEQxQAoPii1lNW1JYMgdfBNATia+zH/8Xe
  =hVUM
  -----END PGP SIGNATURE-----
  ```

- `PGPODetachedSig( context, PGPOLastOption( context) )` tells `PGPEncode()` to create the signature in a separate or ‘detached’ file;
- The call to `PGPOSignWithKey()` tells `PGPEncode()` what key to use when creating the signature, and furnishes the passphrase needed to access it;
- `PGPOInputFile( context, inFileRef )` tells `PGPEncode()` to sign the file you specified as the input source;
- `PGPOOututFile( context, outFileRef )` tells `PGPEncode()` to write the signature data to the output file you specified.

Note that two of these option list functions use option lists themselves. In the `PGPODetachedSig()` call this sub-option list only needs the `PGPOLastOption()`, but in the call to `PGPOSignWithKey()` a real option list appears to furnish the key and passphrase. You'll rarely see option lists nested much deeper than this.

Calling PGPEncode()

To perform the actual signature generation operation, you just call `PGPEncode()` as shown. If all the option parameters were set correctly, this will result in the creation of an output signature file with the name you specified.

Cleanup

Cleanup is again pretty standard, but with a few more items to free this time:

```
/* ---- Release resources used in this operation ------------- */
if( PGPFileSpecRefIsValid( inFileRef ) )
    PGPFreeFileSpec( inFileRef );

if( PGPFileSpecRefIsValid( outFileRef ) )
    PGPFreeFileSpec( outFileRef );

if( PGPKeyIterRefIsValid( keyListIterator ) )
    PGPFreeKeyIter( keyListIterator );

if( PGPKeyListRefIsValid( foundKeysList ) )
    PGPFreeKeyList( foundKeysList );

if( PGPFilterRefIsValid( filter ) )
    PGPFreeFilter( filter );

if( PGPKeySetRefIsValid( foundUserKeys ) )
    PGPFreeKeySet( foundUserKeys );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );

/* ---- PGP Shutdown ----------------------------------------- */
/* Release the PGP context we've been using */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();
```

Full listing: Signing a file

A full listing for this example would look like this:

```
/* ==== Sign a File code fragment ========================== */
/*
 *  Demonstrates simple file signing with the PGPsdk.
 *
 *  Signs file "C:\testIn" with key "test@pgp.com",
 *  producing the separate (aka 'detached') signature file
 *  "C:\testSignature".
 *
 */

/* Specify the input and output filenames, and the ID
 * and passphrase of the key you want to sign with */
char const *inFileName    = "C:\testIn";
char const *outFileName   = "C:\testSignature";
char const *userID        = "test@pgp.com";
char const *keyPassphrase = "testPassphrase";

/* Declare and initialize variables */
PGPError       err            = kPGPError_NoErr;
PGPContextRef  context        = kInvalidPGPContextRef;
PGPKeySetRef   defaultKeyRing = kInvalidPGPKeySetRef;
PGPFileSpecRef inFileRef      = kInvalidPGPFileSpecRef;
PGPFileSpecRef outFileRef     = kInvalidPGPFileSpecRef;
PGPKeySetRef   foundUserKeys  = kInvalidPGPKeySetRef;
PGPFilterRef   filter         = kInvalidPGPFilterRef;
PGPKeyListRef  foundKeysList  = kInvalidPGPKeyListRef;
PGPKeyIterRef  keyListIterator= kInvalidPGPKeyIterRef;
PGPKeyRef      signingKey     = kInvalidPGPKeyRef;

/* ---- PGP Setup ------------------------------------------ */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would call PGPsdkInit()
 * and PGPNewContext() at application startup time, not on
 * an operation-by-operation basis.
 */

/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Prepare Decryption Operation Parameters ------------ */

/* Open the default keyring */
```

```
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
          &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;

/* Create a filter to look for the default "test@pgp.com" key */
err = PGPNewUserIDStringFilter( context, userID,
          kPGPMatchSubString, &filter );
if( IsPGPError( err ) )
    goto Exit;

/* Look for the key */
err = PGPFilterKeySet( defaultKeyRing, filter, &foundUserKeys );
if( IsPGPError( err ) )
    goto Exit;

/* use the first matching key */
err = PGPOrderKeySet( foundUserKeys, kPGPAnyOrdering,
          &foundKeysList );

err = PGPNewKeyIter( foundKeysList, &keyListIterator );
if( IsPGPError( err ) )
    goto Exit;

err = PGPKeyIterNext( keyListIterator, &signingKey );
if( IsPGPError( err ) )
    goto Exit;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, inFileName, &inFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, outFileName, &outFileRef );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Perform the Signing operation ------------- */

/* This is the main event; everything above and
 * below exists only to support this call.
 *
 * Produces the signature output file in the default directory.
 */

err = PGPEncode( context,
          PGPOArmorOutput( context, TRUE ),
          PGPODetachedSig( context,
              /* PGPODetachedSig uses its own option list */
              PGPOLastOption( context ) ),
          PGPOInputFile( context, inFileRef ),
          PGPOOutputFile( context, outFileRef ),
          PGPOSignWithKey( context, signingKey,
              /* PGPOSignWithKey uses its own option list */
              PGPOPassphrase( context, keyPassphrase ),
```

```
                PGPOLastOption( context ),
            PGPOLastOption( context ) );

Exit:

/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( inFileRef ) )
    PGPFreeFileSpec( inFileRef );

if( PGPFileSpecRefIsValid( outFileRef ) )
    PGPFreeFileSpec( outFileRef );

if( PGPKeyIterRefIsValid( keyListIterator ) )
    PGPFreeKeyIter( keyListIterator );

if( PGPKeyListRefIsValid( foundKeysList ) )
    PGPFreeKeyList( foundKeysList );

if( PGPFilterRefIsValid( filter ) )
    PGPFreeFilter( filter );

if( PGPKeySetRefIsValid( foundUserKeys ) )
    PGPFreeKeySet( foundUserKeys );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );

/* ---- PGP Shutdown ----------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would make these calls to
 * PGPFreeContext() and PGPsdkInit() at application shutdown time,
 * and not on an operation-by-operation basis.
 */

/* Release the PGP context we've been using */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();

/* ========== End of Sign a File code fragment ========== */
```

Verifying a signature

In PGP, verifying a signature for a message begins with the message and the signature, and ends with an indication of whether the signature could be successfully verified or not.

To verify a signature, you have to be able to access a key database with the public key of the person (or other entity) that appears to have signed the message.

This section demonstrates how to verify a signature found in a disk file (a 'detached' signature) and display a message telling your user either that the signature successfully verified, or that it didn't. Verifying a signature is a little more complicated than the signing example because you have to provide an event handler function that does a certain amount of work.

Overview

Verifying a detached signature is accomplished by calling `PGPDecode()` with a set of options specifying the input file, the detached signature file, a set of keys where the signing key can be found, and an event handler function. As ever, the code presented in this example demonstrates how to:

- Initialize the PGPsdk library
- Develop the data and create the resources needed to form the options
- Call `PGPDecode()` to perform the verification
- Free the resources you created
- and finally, Shut down the PGPsdk library.

As before, we'll go over the code chunk by chunk. For more context, a full listing of the example code appears at the end of this section.

Setup

Our setup code is once again the same:

```
/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;
```

Creating the file references

In detached signature verification there are two input files (original data and the signature made from it) but no output files, so the file references look a little different this time:

```
/* Specify the filenames */
char const *origFileName  = "C:\testIn";
char const *sigFileName   = "C:\testSignature";

/* Declare and initialize variables */
PGPFileSpecRef        origFileRef    = kInvalidPGPFileSpecRef;
PGPFileSpecRef        sigFileRef     = kInvalidPGPFileSpecRef;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, origFileName, &origFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, sigFileName, &sigFileRef );
if( IsPGPError( err ) )
    goto Exit;
```

Again, Mac OS file references would be handled differently because of the FSSpecs.

Accessing a key database

Again we need access to a key database where the public key used to create the signature can be found. For our simple example using the ‘test@pgp.com’ key, we’ll use the default key ring for this:

```
/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
           &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;
```

Forming options and verifying the signature

As always, you need to transform your encryption parameters into PGP options and feed them to `PGPDecode()`:

```
err = PGPDecode( context,
            PGPOKeySetRef( context, defaultKeyRing ),
            PGPOEventHandler( context, myEventHandler, &sigData ),
            PGPOInputFile( context, sigFileRef ),
            PGPODetachedSig( context,
                /* PGPODetachedSig uses its own option list */
                PGPOInputFile( context, origFileRef),
                PGPOLastOption( context ) ),
            PGPOLastOption( context ) );

if( IsntPGPError( err ) )
{
    /* Test validity of signature here */

    if( sigData.checked )
    {
        /* There was a signature in the file */

        if( sigData.verified )
        {
            /* The signature verified */
        }
        else
        {
            /* Signature was bad */
        }
    }
}
```

Verification options

To run down the option list:

- `PGPOKeySetRef( context, defaultKeyRing )` gives `PGPDecode()` access to a key database where additional information about the signing key can be found;
- `PGPOEventHandler( context, myEventHandler, &myState )` tells `PGPDecode()` to use the event callback handler function you've written when callback events occur;
- `PGPOInputFile( context, origFileRef )` tells `PGPDecode()` to look at the indicated file when verifying the signature;
- The `PGPODetachedSig()` call tells `PGPEncode()` to read the signature from the indicated 'detached' file;

Again you'll notice that one of these option list functions uses an option list of its own.

Calling PGPDecode()

To initiate the signature verification operation, you call `PGPDecode()` as shown. If all the option parameters were set correctly, this will result in a callback to your event callback handler function of type `kPGPEvent_SignatureEvent`.

Note: It's in your callback handler that you'll learn whether the signature verified successfully, not at the `PGPDecode()` call itself.

In your event callback handler function

To support the `PGPDecode()` call shown above for detached signature verification, your event callback handler function will need to include code to deal with the `kPGPEvent_SignatureEvent`. That usually means determining the verification result and reporting it to your user; here's a very simple way of doing that:

```
/* ====================================
 * myEventHandler()
 * ====================================
 *
 * This function will be called during the verification
 * operation, during your PGPDecode() call.
 *
 * Results of the verification will be available here when
 * an event of type kPGPEvent_SignatureEvent is received.
 */

static PGPError  myEventHandler(
    PGPContextRef context,
    PGPEvent*event,
    PGPUserValueuserValue)
{
    (void) context;

    /* branch on the received event code */
    switch( event->type )
    {
        /* ...handle other event codes here... */

        case kPGPEvent_SignatureEvent:
        {
            PGPEventSignatureData*userData;

            userData = (PGPEventSignatureData *) userValue;
            *userData= event->data.signatureData;
```

```
            /* The signing key ref is valid only during the callback */
            userData->signingKey = kInvalidPGPKeyRef;
            break;
        }

        /* ...handle other event codes here... */
    }

    return kPGPError_NoErr;
}
```

The `event` parameter that your function receives is a pointer. You can follow it and access its member `data.signature` to get a pointer to a `PGPEventSignatureData` structure containing the results of the signature verification attempt. For a successful verification, the `signingKey` member will be non-`NULL` and the `verified` member will be `TRUE`.

Cleanup

Back in the main code section, cleanup is simple:

```
/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( sigFileRef ) )
    PGPFreeFileSpec( sigFileRef );

if( PGPFileSpecRefIsValid( origFileRef ) )
    PGPFreeFileSpec( origFileRef );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );


/* ---- PGP Shutdown ----------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would make these calls to
 * PGPFreeContext() and PGPsdkInit() at application shutdown time,
 * and not on an operation-by-operation basis.
 */

/* Release the PGP context and shut down PGP */
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();
```

Full listing: Verifying a detached signature for a file

A full listing for this example would look like this:

```
/* ====================================
 * myEventHandler()
 * ====================================
 *
 * This function will be called during the verification
 * operation, during your PGPDecode() call.
 *
 * Results of the verification will be available here when
 * an event of type kPGPEvent_SignatureEvent is received.
 */

static PGPError myEventHandler(
    PGPContextRef context,
    PGPEvent*event,
    PGPUserValueuserValue)
{
    (void) context;

    /* branch on the received event code */
    switch( event->type )
    {
        /* ...handle other event codes here... */

        case kPGPEvent_SignatureEvent:
        {
            PGPEventSignatureData*userData;

            userData = (PGPEventSignatureData *) userValue;
            *userData= event->data.signatureData;

            /* The signing key ref is valid only during the callback */
            userData->signingKey = kInvalidPGPKeyRef;
            break;
        }

        /* ...handle other event codes here... */
    }

    return kPGPError_NoErr;
}

/* ==== Verify a detached Signature code fragment =============== */
/*
 *  Demonstrates simple file signature verification with the PGPsdk.
 *
 *  Verifies a separate (aka 'detached') signature file
 *  "C:\testSignature" against file "C:\testIn", producing a
 *  yea-or-nay verification judgement.
 *
 */
```

```
/* Specify the filenames */
char const *origFileName  = "C:\testIn";
char const *sigFileName   = "C:\testSignature";

/* Declare and initialize variables */
PGPError               err            = kPGPError_NoErr;
PGPContextRef          context        = kInvalidPGPContextRef;
PGPKeySetRef           defaultKeyRing = kInvalidPGPKeySetRef;
PGPFileSpecRef         origFileRef    = kInvalidPGPFileSpecRef;
PGPFileSpecRef         sigFileRef     = kInvalidPGPFileSpecRef;
PGPEventSignatureData  sigData;

/* ---- PGP Setup ------------------------------------------ */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would call PGPsdkInit()
 * and PGPNewContext() at application startup time, not on
 * an operation-by-operation basis.
 */

/* Initialize the PGPsdk libs */
err = PGPsdkInit();
if( IsPGPError( err ) )
    goto Exit;

/* Create the PGPContext */
err = PGPNewContext( kPGPsdkAPIVersion, &context );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Prepare Decryption Operation Parameters ------------- */

/* Open the default keyring */
err = PGPOpenDefaultKeyRings( context, kPGPKeyRingOpenFlags_Mutable,
          &defaultKeyRing );
if( IsPGPError( err ) )
    goto Exit;

/* Create the file descriptions */
err = PGPNewFileSpecFromFullPath( context, origFileName, &origFileRef );
if( IsPGPError( err ) )
    goto Exit;

err = PGPNewFileSpecFromFullPath( context, sigFileName, &sigFileRef );
if( IsPGPError( err ) )
    goto Exit;

/* ---- Perform the Verify operation ------------- */

/* This is the main event; everything above and
 * below exists only to support this call.
 *
 * NOTE: Verification results are determined in the event handler
 * function, myEventHandler(), not here.
 */
```

```
sigData.verified = FALSE;

err = PGPDecode( context,
            PGPOKeySetRef( context, defaultKeyRing ),
            PGPOEventHandler( context, myEventHandler, &sigData ),
            PGPOInputFile( context, sigFileRef ),
            PGPODetachedSig( context,
                /* PGPODetachedSig uses its own option list */
                PGPOInputFile( context, origFileRef),
                PGPOLastOption( context ) ),
            PGPOLastOption( context ) );

if( IsntPGPError( err ) )
{
    /* Test validity of signature here */

    if( sigData.checked )
    {
        /* There was a signature in the file */

        if( sigData.verified )
        {
            /* The signature verified */
        }
        else
        {
            /* Signature was bad */
        }
    }
}

Exit:

/* ---- Release resources used in this operation ------------- */

if( PGPFileSpecRefIsValid( sigFileRef ) )
    PGPFreeFileSpec( sigFileRef );

if( PGPFileSpecRefIsValid( origFileRef ) )
    PGPFreeFileSpec( origFileRef );

if( PGPKeySetRefIsValid( defaultKeyRing ) )
    PGPFreeKeySet( defaultKeyRing );


/* ---- PGP Shutdown ----------------------------------------- */

/* NOTE: These calls appear here for clarity only.
 * In a real-world application, you would make these calls to
 * PGPFreeContext() and PGPsdkInit() at application shutdown time,
 * and not on an operation-by-operation basis.
 */

/* Release the PGP context and shut down PGP */
```

```
if( PGPContextRefIsValid( context ) )
    PGPFreeContext( context );

/* PGP library shutdown */
PGPsdkCleanup();
```

PGPsdk Frequently Asked Questions 5

This chapter answers some of the questions programmers most often ask PGP developer support regarding programming with the PGPsdk.

Frequently Asked Questions

What operating systems does PGPsdk support?

Windows 95 OSR2, 98, or NT with SP 4 or later

Mac OS System 7.6.1 or later

Linux x86 with 2.0.x kernel or later

Sun Solaris 2.5.1, 2.6, or 2.7

What key management functions are available from the PGPsdk?

The PGPsdk provides access to key management functions that allow applications to create, sign, add, remove, search for, and check the validity of keys on disk-based or in-memory key rings. It also includes functions for checking and setting key property values according to the PGP 'web of trust' model, as well as functions that import and export keys to files and buffers.

The PGPsdk also provides access to functions that support communication with HTTP and LDAP key servers, and that allow developers to search for, add, disable, and delete keys on those servers. In addition the PGPsdk includes functions for connecting to secure severs using TLS (Transport Layer Security, a protocol based on SSL).

Do I have to use a key from a key certificate when encrypting?

No. Your alternative is to use our conventional encryption option to encrypt data to a passphrase instead of a PGP key. Practically speaking, PGP keys exist only in the form of key certificates.

Can I use PGPsdk to generate keys, or do I need a certificate server for that?

Yes, you can generate a key pair with PGP API calls available in the PGPsdk. Key generation is generally a client-based operation, not a server-based operation, and NAI's key server product does not generate keys. In the PGP scheme, a key server is just a central repository of keys for management and control. When a PGP user wants to publish a public key that she has generated, she needs to submit it to a key server.

Can I encrypt to more than one key?

Yes, you can encrypt data to any number of keys with PGP API calls available in the PGPsdk.

Is it possible to encrypt on one platform, say Windows 95, and decrypt on another, say Solaris?

Yes, the PGP cryptosystem is completely platform-independent. Data encrypted on any platform can be decrypted on any other platform, so long as the recipient's correct public key is used for the encryption.

Does the PGPsdk include a random number generator?

PGPsdk includes functions to generate and manage a pool of random numbers seeded from keystrokes and mouse movements, and provides both cryptographically strong pseudo-random numbers and true random numbers based on external events. The PGPsdk internal pseudo-random number generator (RNG) is based on ANSI X9.17.

Does the PGPsdk support Microsoft Visual Basic?

Sorry, the PGPsdk does not currently support Visual Basic. You would need to write an appropriate wrapper layer to use the PGP libraries from VB.

Does the PGPsdk support Java?

Sorry, the PGPsdk does not currently support Java.