

# Programming with GPGME and GPGAppKit on OS X

Gordon Worley

February 28, 2004

*Programming with GPGME and GPGAppKit on OS X* is Copyright 2003-2004 Mac GPG Project. You may distribute verbatim copies, in whole or in part, so long as this copyright information is preserved and no profit is earned, although you may charge fees to cover the costs of distribution. The author reserves the right to distribute copies of this book at a profit and to allow others to do so upon request.

# Preface

On a recent trip to the bookstore I found a shelf of cryptography books with titles like *Cryptography Decrypted* and *Cryptography Demystified*. It reminded me that cryptography is popular, especially relative to other information theory topics (you don't find pulpy titles on books about Shannon Information or block coding). Cryptography has a mystique that attracts people and makes them feel like cloak-and-dagger spies or important people with secrets to protect. It's a form of escapism for the computer enthusiast.

The reality, though, is not so romantic. Cryptographers spend lonely hours cryptanalyzing, auditing code, and calculating, not saving the world from terrorists. Faced with it, most people would much rather just use cryptography like the spy than research it like the cryptographer. That's where programmers come in: making the spy tools.

My goal for this book is to teach you how to add cryptography to your programs because I believe that many applications would benefit from cryptographic features. Although the scope of this book is limited to Cocoa applications on Mac OS X, GPGME and other libraries are available for most systems. Even if my technical advice does not help you, I hope that it inspires you to investigate cryptographic integration on your development platform.

—Gordon Worley  
25 May, 2003



# Contents

<b>1</b>	<b>Getting Started</b>	<b>7</b>
1.1	Prerequisites . . . . .	7
1.2	GPGME . . . . .	7
1.3	GPGME, GPGAppKit, and Mac OS X . . . . .	8
1.4	How to use this book . . . . .	8
1.5	A work in progress . . . . .	9
<b>2</b>	<b>Simple GPGME Example</b>	<b>11</b>
2.1	Creating SimpleGPGMEExample . . . . .	11
2.2	User Interface . . . . .	11
2.3	Architecture . . . . .	12
2.4	Libraries . . . . .	13
2.5	Code . . . . .	13
2.6	Code Analysis . . . . .	16
2.7	Exercises . . . . .	20
<b>3</b>	<b>Simple GPGAppKit Example</b>	<b>21</b>
3.1	Creating SimpleGPGAppKitExample . . . . .	21

3.2	User Interface . . . . .	21
3.3	Architecture . . . . .	22
3.4	Libraries . . . . .	22
3.5	Code . . . . .	23
3.6	Code Analysis . . . . .	26
3.7	Exercises . . . . .	29

# Chapter 1

## Getting Started

Before we can start writing cryptography into our applications, we must cover some basics. There is a lot of material written about Mac OS X programming and cryptography, so although I've chosen the book format, you should treat it more like a paper.

### 1.1 Prerequisites

This book is intended for Mac OS X developers who want to add public key cryptography to their applications. I assume that you are familiar with public key cryptography, PGP (Pretty Good Privacy), and GnuPG (GNU Privacy Guard). If you are not, I suggest reading the *GNU Privacy Handbook*, available Online at <http://www.gnupg.org/gph/en/manual.html>. You should also be familiar with development on Mac OS X using Cocoa, Objective-C, and Apple's developer tools..

### 1.2 GPGME

GPGME is short for GnuPG Made Easy, a library that “provides a High-Level Crypto API for encryption, decryption, signing, signature verification and key management.”[1] It is the technology that sits between our Cocoa frameworks and GnuPG. Although we will not deal directly with the GPGME C library (other than to install it), you should be familiar with its existence. You can learn more about it by visiting GPGME's Website: [http://www.gnupg.org/\(en\)/related\\_software/gpgme/index.html](http://www.gnupg.org/(en)/related_software/gpgme/index.html).

## 1.3 GPGME, GPGAppKit, and Mac OS X

The GPGME C library is not object oriented, nor does it fit the Objective-C idiom. To resolve these dilemmas, Stéphane Corthésy et al. created GPGME.framework, an Objective-C wrapper for the GPGME C library that works with Cocoa and GNUstep. GPGME.framework provides all of the functionality of the C library (plus a few added features), but in a way that is conducive to Objective-C programming.

Because we have two libraries called GPGME, naming can get confusing. As used in this book, ‘GPGME’ refers to the idea of GPGME: an API for cryptographic functions that works with GnuPG/OpenPGP and possibly other programs/protocols. ‘libgpgme’ is the GPGME C library distributed by the GnuPG Project. ‘GPGME.framework’ refers to the Cocoa framework from the Mac GPG Project.

While GPGME.framework provides the logical components for public key cryptography, it lacks common user interface elements for getting data from the user. Often applications using GPGME need to ask for the same data repeatedly: passphrase, recipients, signers. In response, the Mac GPG Project developed GPGAppKit, a Cocoa framework of user interface elements to obtain these data. GPGAppKit returns data as objects from GPGME.framework, so that the two frameworks work together seamlessly.

To follow the examples in this book, you need to download and install GPGME and GPGAppKit. Both are available from <http://macgpg.sourceforge.net/>.

This version of the book is based on GPGME.framework 0.3.14 and GPGAppKit.framework A1.

## 1.4 How to use this book

The chapters of this book contain source code and exercises to help you learn. I suggest that you work through all of the examples and exercises at least once to familiarize you with GPGME.framework’s and GPGAppKit.framework’s objects and methods. If, however, you are using them as a reference, you will find the sample projects on the Web at [http://macgpg.sourceforge.net/GPGME\\_Book/](http://macgpg.sourceforge.net/GPGME_Book/).

You should work through this book sequentially. Each chapter builds on concepts explained in the previous chapters and assumes that you are already familiar with the information introduced earlier.



## **1.5 A work in progress**

This book is still being written. It will eventually contain more chapters explaining how to work with GPGME and GPGAppKit and general information on secure programming and user interface design. Chapter order may change between editions, with new chapters displacing older ones. If you find typos or have suggestions for future editions, please e-mail them to Gordon Worley, [redbird@mac.com](mailto:redbird@mac.com).



# Chapter 2

## Simple GPGME Example

The purpose of this chapter is to familiarize you with GPGME and its workings. After you read this chapter you should have a general feeling for working with GPGME.framework and where to look to learn more. Out of necessity, the user interface for our example application in this chapter, SimpleGPGMEExample, is rudimentary. In Chapter 3, you will use GPGAppKit to create more friendly user interfaces.

### 2.1 Creating SimpleGPGMEExample

Start ProjectBuilder and create a Cocoa application project called SimpleGPGMEExample. You can save it anywhere in your filesystem, but I recommend creating a special folder for the examples in this book (‘~/Developer/GPGME Book’, for example) to help you remember where they are. Nothing sucks quite as much as forgetting where you saved your code.

### 2.2 User Interface

When the SimpleGPGMEExample project opens, open MainMenu.nib. In InterfaceBuilder, construct an interface that looks like the one in Figure 2.1. The interface is made of NSTextFields (some in the label style, others in the text entry box style) and one NSButton.

Note that the text field next to the “Passphrase:” label is an NSSecureTextField, not a NSTextField. Although you can use an NSTextField for testing, an NSSecureTextField will echo bullets. It is highly recommended that you use an NSSecureTextField, even during development, so that onlookers will not be able to spy your passphrase.

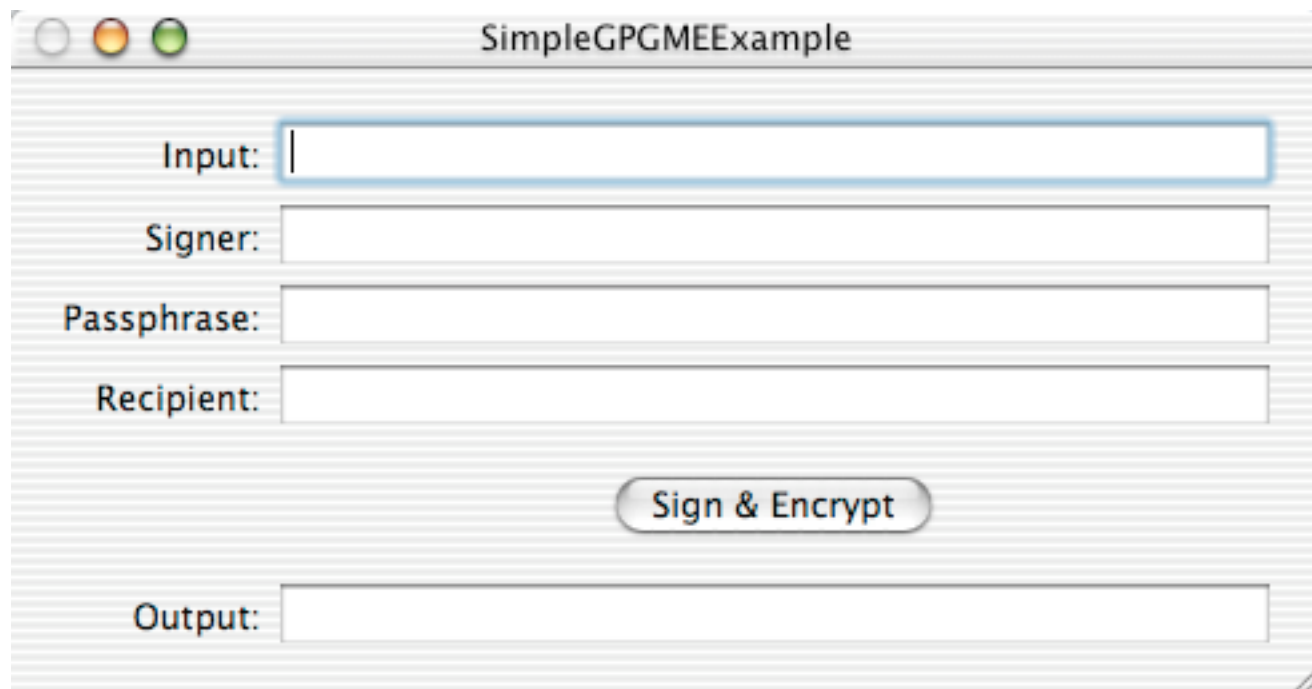


Figure 2.1: SimpleGPGMEExample User Interface

Don't worry about anything else. This is just a simple interface so you can play around with GPGME.framework. In Chapter 3 we will discuss building a more user friendly interface.

## 2.3 Architecture

In InterfaceBuilder, create a new subclass of NSObject called GPGMEController. Add to it 4 NSTextField outlets—data, signer, recipient, and output—and an NSSecureTextField outlet named passphrase. Also add an action called signAndEncrypt:. Instantiate GPGMEController. Connect GPGMEController's outlets to the similarly labeled text fields (output connects to the NSTextField next to the 'Output:' label, etc.). Draw a connection from the button to GPGMEController and make signAndEncrypt: the target of your button. Create the files for GPGMEController and add them to SimpleGPGMEExample.

You are done with InterfaceBuilder for the rest of this example. You can safely close it if you've done everything up to this point correctly.

## 2.4 Libraries

Back in ProjectBuilder, you need to link to GPGME.framework. From the Project menu, choose Add Frameworks... and find GPGME.framework on your hard drive (probably in the build directory of wherever you put the GPGME project when you got it from the Web or CVS). You should have built it using the EmbDeployment style, but if not rebuild it in that style. Once linked, add a build phase to copy GPGME.framework into the Frameworks directory of SimpleGPGMEExample.app.

### Embedding GPGME

I strongly encourage you to embed GPGME.framework in your applications. It simplifies the install process and it ensures that your application always has a working version of GPGME.framework present. If you depend on a copy in a Frameworks directories outside your application, bugs may result from incompatible but installed versions of GPGME or lack of installed copy of GPGME. Including a copy of GPGME in your application does not dramatically increase your applications size and it only adds a few seconds to the time to download your application. By including GPGME in your application you save everyone a lot of headaches at a minimal cost.

See GPGME.framework's documentation for more on this argument.

## 2.5 Code

First you need to modify GPGMEController.h to import the GPGME framework. Add the second `#import` statement (the rest of the file should already look exactly like this):

```
/* GPGMEController */

#import <Cocoa/Cocoa.h>
#import <GPGME/GPGME.h>

@interface GPGMEController : NSObject
{
    IBOutlet NSTextField *input;
    IBOutlet NSTextField *output;
    IBOutlet NSSecureTextField *passphrase;
    IBOutlet NSTextField *recipient;
```

```

        IBOutlet NSTextField *signer;
    }
    - (IBAction)signAndEncrypt:(id)sender;
@end

```

Now you need to fill in GPGMEController.m as below. We will discuss what this code does in the next section.

```

#import "GPGMEController.h"

@implementation GPGMEController

- (IBAction)signAndEncrypt:(id)sender
{
    GPGContext *context = nil;
    GPGRecipients *gpgRecipient = nil;
    GPGKey *gpgSigner = nil;
    GPGData *gpgData = nil;
    GPGData *gpgOutput = nil;

    NS_DURING
        //initializers
        context = [[GPGContext alloc] init];
        gpgData = [[GPGData alloc] initWithString: [input stringValue]];
        gpgRecipient = [[GPGRecipients alloc] init];

        //get the recipients

        [gpgRecipient addName: [recipient stringValue]];

        //get the signers

        gpgSigner =
            [[context keyEnumeratorForSearchPattern: [signer stringValue]
              secretKeysOnly: YES] nextObject];

        [context stopKeyEnumeration];

        //set up the context

        [context addSignerKey: gpgSigner];
        [context setPassphraseDelegate: self];
        [context setUsesArmor: YES];
    NS_END
}

```

```

        //do the GPG magic

        gpgOutput = [context encryptedSignedData: gpgData
                        forRecipients: gpgRecipient
                        allRecipientsAreValid: nil
                    ];

        //display output

        [output setValue: [gpgOutput string]];

    NS_HANDLER
        NSRunAlertPanel(@"Oops, you broke it.", @"%@",
                        nil, nil, nil, localException);
    NS_ENDHANDLER

    [context release];
    [gpgData release];
    [gpgRecipient release];
}

- (NSString *) context: (GPGContext *)context
  passphraseForKey: (GPGKey *)key
    again: (BOOL)again
{
    return [passphrase stringValue];
}

@end

```

After completing GPGMEController.m, build the project and run SimpleGPGMEExample.app. The operation is simple: type a plaintext message in the input field, a search for a secret key in the signer field (key id or fingerprint are best), the passphrase for the key in the next field, and a search for a public key in the recipient field. Press the “Sign & Encrypt” button to produce the cipher text in the output field.

Experiment with intentional errors. See what happens if you enter an invalid passphrase, don’t specify a signer, don’t specify a valid recipient, or specify other potentially error causing values.

## 2.6 Code Analysis

```
#import "GPGMEController.h"

@implementation GPGMEController

- (IBAction)signAndEncrypt:(id)sender
{
    GPGContext *context = nil;
    GPGRecipients *gpgRecipient = nil;
    GPGKey *gpgSigner = nil;
    GPGData *gpgData = nil;
    GPGData *gpgOutput = nil;
```

Here we initialize the variables to nil so that we only need to write the cleanup code once—after the error handling code—to avoid memory leaks that would result if the memory management were attempted inside the error handling code. By setting the variables to nil, when an error is raised inside `NS_DURING`, some of the variables may not be initialized, but if they are set to nil then you can send them a release message which will be accepted and ignored, rather than halting execution the way uninitialized variables sent a message would.

Although some of these variables will enter the autorelease pool, it's good practice to initialize them all to nil. It's one less line of code to change if an autoreleased variable is reprogrammed as a manually released variable.

```
NS_DURING
    //initializers
    context = [[GPGContext alloc] init];
    gpgData = [[GPGData alloc] initWithString: [input stringValue]];
```

In any function using GPGME, the first step is to create a `GPGContext` instance. A `GPGContext` is like a `gpg` process; you can send a `GPGContext` commands much like you would send `gpg` on the command line. The difference is that a `GPGContext` interacts with programmatic data structures rather than human typing.

A program can use as many contexts as you want, but in this case only one is necessary. Assuming that certain conditions are preserved, you can share data between contexts, even if they are in different threads. For example, you might search for keys in one thread, search for recipients in another, and use cryptographic functions with the data collected from the first two threads in yet another thread. Keep in mind, however, that certain user options may prevent real multithreaded use of `gpg` (there may be a mutex on the keyrings, for example).



GPGData is analogous to NSData, but specifically designed for use with GPGME. GPGData has a different internal structure than NSData because libpgme, which GPGME.framework wraps, accepts data using specific data structures that could not be easily managed by extending NSData. But GPGData and NSData work together; you can easily convert from one to the other, or into several other Cocoa objects.

```
gpgRecipient = [[GPGRecipients alloc] init];

//get the recipients

[gpgRecipient addName: [recipient stringValue]];
```

GPGRecipients is an enumeration of public key specifiers, specifically for the purpose of specifying the recipients of a message. You add keys by search string, which can be anything that will find the key. If a search finds multiple keys, only the first one found will be used. In fact, the search will stop after the first match is found, though you should not rely on GnuPG to always return the same key first. Thus it is best to search for recipients using key fingerprints since they have the fewest collisions.

In our code, we add the key specified by the search string in the recipients text field. Although you can theoretically specify multiple recipients, SimpleGPGMEExample allows only one. It is left as an exercise to develop a method for specifying multiple recipients (see §2.7).

```
//get the signers

gpgSigner =
    [[context keyEnumeratorForSearchPattern: [signer stringValue]
        secretKeysOnly: YES] nextObject];

[context stopKeyEnumeration];
```

Just as we need recipients, we need signers. gpgSigner is a GPGKey, which represents a key and contains information about it, but not the actual key itself. We search the context for keys, just as we would in Terminal with gpg and the `-list-keys` command, using the GPGContext instance method `keyEnumeratorForSearchPattern:secretKeysOnly:`. It returns an NSEnumerator of GPGKeys, but we are only interested in the first object for our example, although you can sign data with multiple keys. You should be careful when you specify search strings because, just as with GPGRecipients's `addName:`, collisions can occur, although here you'll get all keys that match the search string rather than the one that happened to be found first.

The second parameter of `keyEnumeratorForSearchPattern:secretKeysOnly:` determines whether public or secret keys are returned. Signers on a GPGContext must be secret keys, so when

obtaining signers the second parameter should always be YES. Setting it to NO is useful for finding recipient keys.

In this example, it is important to send `stopKeyEnumeration` to context because a `GPGContext` will continue to search for keys even after `keyEnumeratorForSearchPattern:secretKeysOnly:` returns. If you want to use a `GPGContext` after a key search, you need to send it `stopKeyEnumeration` or else you will get a busy error. In later examples, when you use `GPGAppKit` to find keys, you will not need to worry about stopping key enumeration: it will be stopped for you.

```
//set up the context

[context addSignerKey: gpgSigner];
[context setPassphraseDelegate: self];
[context setUsesArmor: YES];
```

These lines set options for the `GPGContext`, just like you can set options when using `gpg` in Terminal. `addSignerKey:` acts like `-local-user` or `-u` on the command line, adding a key to the list of signers. The second line specifies the passphrase delegate. We'll talk more about this later. Third we set the `GPGContext` to use ASCII armor so that the output will be 'human readable'—we'll be able to post it to the output text field.

There are several more options available. See `GPGME/GPGContext.h` for them all.

```
//do the GPG magic

gpgOutput = [context encryptedSignedData: gpgData
                      forRecipients: gpgRecipient
                      allRecipientsAreValid: nil
];
```

This message signs and encrypts `gpgData` (taken from input). It is encrypted to `gpgRecipient` and signed with the key we added in the previous code segment. The result is put in `gpgOutput`, another `GPGData` instance.

The third parameter of `encryptedSignedData:forRecipients:allRecipientsAreValid:` is a pointer to a `BOOL` that, after this method returns, points to a `BOOL` that is YES if all of the recipients are valid. Here we choose to ignore it, but it's important because you can only encrypt to valid recipients. If you try to sign to an invalid recipient (a recipient that you don't trust), you won't get an error, but you won't get any output either. In real programs you should check this value and notify the user if encryption fails or take preemptive steps to ensure that it will always be YES (i.e. only allow the user to select valid recipients).

```
//display output

[output setValue:[gpgOutput string]];
```

This part displays the data. We turn `gpgOutput` into a string, which only works because we armored the output. If we had not, then `gpgOutput` could not convert to an `NSString`, although you can still convert `gpgOutput` to an `NSData` instance without armor.

```
NS_HANDLER
    NSRunAlertPanel(@"Oops, you broke it.", @"%@",
                    nil, nil, nil, localException);
NS_ENDHANDLER
```

`GPGME.framework` can raise a variety of errors. You'll get errors when there is something wrong with the data, the passphrase is incorrect, or some other strange thing happened. Look at `GPGME/GPGContext.h` for a list of the errors that `GPGContext` can raise. Other objects can also raise errors; see their header files for details. If you do not catch these errors then your application will crash or, at least, fail to complete cryptographic tasks every time something goes wrong.

```
[context release];
[gpgData release];
[gpgRecipient release];
}
```

At the beginning of this method we initialized all of our variables to `nil`. Now we take advantage of that to send those variables that are not to be autoreleased a release message. This is a useful technique because `NS_DURINGs` can cause memory leaks if you don't carefully manage your data. A novice mistake with error handling in Cocoa is to put all memory allocation and deallocation inside the error handling code, but this can lead to memory leaks if an error occurs between the allocation and deallocation code.

You're not crazy; this is a duplicate discussion. I feel, however, that correct memory management is important enough to warrant repeating the point.

```
- (NSString *) context: (GPGContext *)context
    passphraseForKey: (GPGKey *)key
        again: (BOOL)again
{
    return [passphrase stringValue];
```

```
}
```

```
@end
```

This is the delegate method for telling our `GPGContext` the passphrase (remember when we set the passphrase delegate to self earlier?). We will use this method's arguments with `GPGAppKit` in a later example, but for now we return the value of the passphrase text field. `GPGContext` will ask for the passphrase three times for any given command, just as `gpg` would on the command line. In this case, even if it is incorrect, we just keep returning the same passphrase, which, if it is wrong, will eventually raise a no passphrase error. Later we will see how to handle each try individually.

## 2.7 Exercises

1. Change the output `NSTextField` to an `NSTextView`.
2. Add a way for the user to specify multiple recipients.
3. Check if all recipients are valid and report the result to the user.

# Chapter 3

## Simple GPGAppKit Example

In the last chapter we learned the basics of GPGME. In this chapter we will expand on that knowledge, using GPGAppKit to build user-friendly interfaces to GPGME features in applications.

### 3.1 Creating SimpleGPGAppKitExample

Start ProjectBuilder and create a new Cocoa application project called SimpleGPGAppKitExample. If you are saving the examples from this book together, remember to create the project in the appropriate directory.

### 3.2 User Interface

When SimpleGPGAppKitExample opens in ProjectBuilder, open MainMenu.nib. In InterfaceBuilder, construct an interface like the one in Figure 3.1 made of four NSTextFields and one NSButton.

Although SimpleGPGAppKitExample and SimpleGPGMEExample have similar interfaces, SimpleGPGAppKitExample initially asks only for the input data to sign and encrypt because we will use GPGAppKit to get the rest of the information we need (recipient, signer, and passphrase). In fact, most applications using GPGAppKit only need to provide a way to input and output data; GPGAppKit does the rest.



Figure 3.1: SimpleGPGAppKitExample User Interface

### 3.3 Architecture

In InterfaceBuilder, create a subclass of NSObject called GPGAppKitController. It has two NSTextField outlets—input and output—and one action named signAndEncrypt:. Instantiate GPGAppKitController and connect its outlets to the appropriate text fields. Connect the “Sign and Encrypt” button’s target to signAndEncrypt: on GPGAppKitController. Create the files for GPGAppKitController and add them to SimpleGPGAppKitExample.

We are done with InterfaceBuilder for this example.

### 3.4 Libraries

Back in ProjectBuilder, link to the GPGME and GPGAppKit frameworks. From the Project menu, choose Add Frameworks... and find GPGME.framework as you did for SimpleGPGMEExample. Then repeat the process for GPGAppKit.framework. Both should have been built in the EmbDeployment style (if not, go back and do it). Then add a copy build phase to copy both frameworks into SimpleGPGAppKitExample’s Framework directory.

As argued in §2.4(Embedding GPGME), you should embed these frameworks inside your applications.

To use GPGAppKit, you must link to GPGME.framework in your application. GPGAppKit depends on GPGME but does not copy GPGME into itself. Since GPGAppKit is of little or no use without GPGME, this is a feature to reduce the size of your applications.

## 3.5 Code

Begin by modifying GPGAppKitController.h to import the GPGME and GPGAppKit frameworks. Add the second and third import statements so that GPGAppKitController.h looks like this:

```
/* GPGAppKitController */

#import <Cocoa/Cocoa.h>
#import <GPGME/GPGME.h>
#import <GPGAppKit/GPGAppKit.h>

@interface GPGAppKitController : NSObject
{
    IBOutlet NSTextField *input;
    IBOutlet NSTextField *output;
}
- (IBAction)signAndEncrypt:(id)sender;
@end
```

Now fill in GPGAppKitController.m as below. We will discuss what this code does in the next section.

```
#import "GPGAppKitController.h"

@implementation GPGAppKitController

- (IBAction)signAndEncrypt:(id)sender
{
    //from GPGME
    GPGContext *context = nil;
    GPGRecipients *gpgRecipient = nil;
    GPGKey *gpgSigner = nil;
    GPGData *gpgData = nil;
    GPGData *gpgOutput = nil;

    //from GPGAppKit
    GPGSingleKeySelectionPanel *keySelectionPanel = nil;

    NS_DURING
        //initializers
        context = [[GPGContext alloc] init];
```

```
gpgData = [[GPGData alloc] initWithString: [input stringValue]];
gpgRecipient = [[GPGRecipients alloc] init];
keySelectionPanel = [GPGSingleKeySelectionPanel panel];

//get the recipients

[keySelectionPanel setMinimumKeyValidity: GPGValidityMarginal];
[keySelectionPanel setListsSecretKeys: NO];
[keySelectionPanel setPrompt: @"Choose recipient"];

[keySelectionPanel runModalForKeyWildcard: nil
                    usingContext: context];

[gpgRecipient addName:
 [[keySelectionPanel selectedKey] fingerprint]];

//get the signers

[keySelectionPanel resetToDefaults];
[keySelectionPanel setMinimumKeyValidity: GPGValidityUltimate];
[keySelectionPanel setListsSecretKeys: YES];
[keySelectionPanel setPrompt: @"Choose signer"];

[keySelectionPanel runModalForKeyWildcard: nil
                    usingContext: context];

gpgSigner = [keySelectionPanel selectedKey];

//set up the context

[context addSignerKey: gpgSigner];
[context setPassphraseDelegate: self];
[context setUsesArmor: YES];

//do the GPG magic

gpgOutput = [context encryptedSignedData: gpgData
                    forRecipients: gpgRecipient
                    allRecipientsAreValid: nil
                    ];

//display output

[output setStringValue: [gpgOutput string]];
```



```

NS_HANDLER
    NSRunAlertPanel(@"Oops, you broke it.", @"%@",
                    nil, nil, nil, localException);
NS_ENDHANDLER

[context release];
[gpgData release];
[gpgRecipient release];
}

- (NSString *) context: (GPGContext *)context
  passphraseForKey: (GPGKey *)key
    again: (BOOL)again
{
    GPGPassphrasePanel *ppanel = [GPGPassphrasePanel panel];

    if (again)
    {
        [ppanel runModalWithPrompt:
         [NSString stringWithFormat: @"Try again: %@", %@,
          [key userID], [key shortKeyID]
        ]
        ];
    }
    else
    {
        [ppanel runModalWithPrompt:
         [NSString stringWithFormat: @"Enter passphrase: %@", %@,
          [key userID], [key shortKeyID]
        ]
        ];
    }

    return [ppanel passphrase];
}

@end

```

The program is done. Compile and run it. See what kind of errors you can generate, especially by experimenting with GPGAppKit's dialogs.

## 3.6 Code Analysis

Because GPGAppKitController.h and GPGMEController.h are very similar, there is no need to reanalyze some sections of the code. Below we know when sections of the code have been skipped. Points addressed in the previous chapter are not repeated here.

First we skip the beginning of GPGAppKitController's implementation to the first new line:

```
//from GPGAppKit
GPGSingleKeySelectionPanel *keySelectionPanel = nil;
```

We have added only one variable, keySelectionPanel, to the method variables needed in GPGMEController's signAndEncrypt:. Using GPGAppKit does not change how you apply cryptographic functions to data, only how you gather information from the user.

```
NS_DURING
    //initializers
    context = [[GPGContext alloc] init];
    pgpData = [[GPGData alloc] initWithString: [input stringValue]];
    pgpRecipient = [[GPGRecipients alloc] init];
    keySelectionPanel = [GPGSingleKeySelectionPanel panel];
```

The last line creates a GPGSingleKeySelectionPanel, which provides a dialog, as seen in Figure 3.2, with a pop-up list of keys to choose from. To initialize keySelectionPanel we send the class GPGSingleKeySelectionPanel a panel message. Unless you have a good reason not to, you should always use this class method to create instances of GPGSingleKeySelectionPanel because the alloc init sequence will not create a GPGSingleKeySelectionPanel instance ready for immediate use. See the source code of GPGAppKit for details.

```
//get the recipients

[keySelectionPanel setMinimumKeyValidity: GPGValidityMarginal];
[keySelectionPanel setListsSecretKeys: NO];
[keySelectionPanel setPrompt: @"Choose recipient"];

[keySelectionPanel runModalForKeyWildcard: nil
                    usingContext: context];

[pgpRecipient addName:
 [[keySelectionPanel selectedKey] fingerprint]];
```

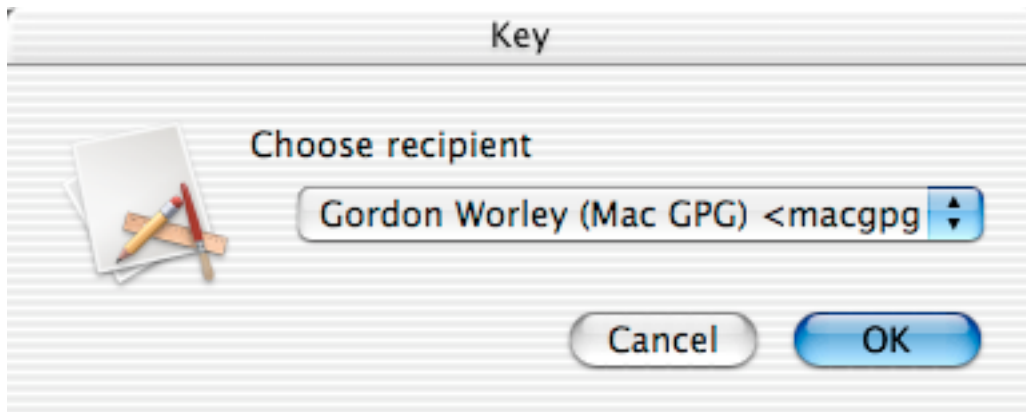


Figure 3.2: A GPGSingleKeySelectionPanel in action

Here we use `keySelectionPanel` to ask the user for a recipient, but first we must set criteria that specify what keys to display to the user. We want to show only keys with at least marginal validity because less valid keys are invalid recipients and will cause encryption to fail. Rather than inform the user later that they picked an invalid recipient, we can show them only choices that will have the desired effects. This follows the secure interface design principles described in [2].

We also specify not to list secret keys, i.e. to list public keys. You can only encrypt to public keys, so it makes little sense to list secret keys when asking for recipients.

Next we set the prompt to explain the choice the user will make with `keySelectionPanel`: selecting a recipient. It is important in secure systems that you let the user know clearly and simply what every choice means.[2]

The method `runModalForKeyWildcard:usingContext:` creates a modal dialog version of `keySelectionPanel`, though you can also display it as a sheet and as a relative modal dialog (a.k.a. a modal sheet). The first parameter is a key search pattern and is ‘optional’: you can pass it nil if you want to list all keys meeting the other criteria. The second argument, though, must be a valid `GPGContext`. Otherwise `keySelectionPanel` will not display any keys and may raise errors.

The method `selectedKey` returns the users choice as a `GPGKey` instance. We use its fingerprint to add it as a recipient because fingerprints rarely collide.

```
//get the signers

[keySelectionPanel resetToDefaults];
[keySelectionPanel setMinimumKeyValidity: GPGValidityUltimate];
[keySelectionPanel setListsSecretKeys: YES];
[keySelectionPanel setPrompt: @"Choose signer"];
```

```

        [keySelectionPanel runModalForKeyWildcard: nil
                        usingContext: context];

    gpgSigner = [keySelectionPanel selectedKey];

```

To save memory, we reuse `keySelectionPanel` to get the signer. First we reset its options to default, as they were when `keySelectionPanel` was initialized. Without `resetToDefaults` some of the key search criteria from the last use might bleed over into this use. Since the user is looking for a signing key, we ask for secret keys with ultimate validity. This does not check, though, if the key is actually capable of signing, which is left as an exercise to the reader.

The rest of this method is identical to the method of the same name in `GPGMEController` from Chapter 2 because `GPGAppKit` only affects *how* you get information from the user, not *what* information you get from the user.

```

- (NSString *) context: (GPGContext *)context
  passphraseForKey: (GPGKey *)key
    again: (BOOL)again
{
    GPGPassphrasePanel *ppanel = [GPGPassphrasePanel panel];

    if (again)
    {
        [ppanel runModalWithPrompt:
            [NSString stringWithFormat: @"Try again: %@, %@",
                [key userID], [key shortKeyID]
            ]
        ];
    }
    else
    {
        [ppanel runModalWithPrompt:
            [NSString stringWithFormat: @"Enter passphrase: %@, %@",
                [key userID], [key shortKeyID]
            ]
        ];
    }

    return [ppanel passphrase];
}

@end

```

This method uses a `GPGPassphrasePanel` instance to ask the user for the passphrase. `GPGPassphrasePanel` provides a simple prompt and `NSSecureTextField` to enter the passphrase for the key (see Figure 3.3).

We ignore the first argument, a `GPGContext` instance, because we have no use for it in our example. The second argument, a `GPGKey` instance representing the key that the first argument needs a passphrase for, provides information to tell the user what key a passphrase is needed for. Because many users have multiple keys that can sign or may be signing a message using multiple keys, it is necessary to identify which key passphrase is requested. It is also good secure user interface design, because it helps the user know exactly what to do, rather than forcing them to make guesses.

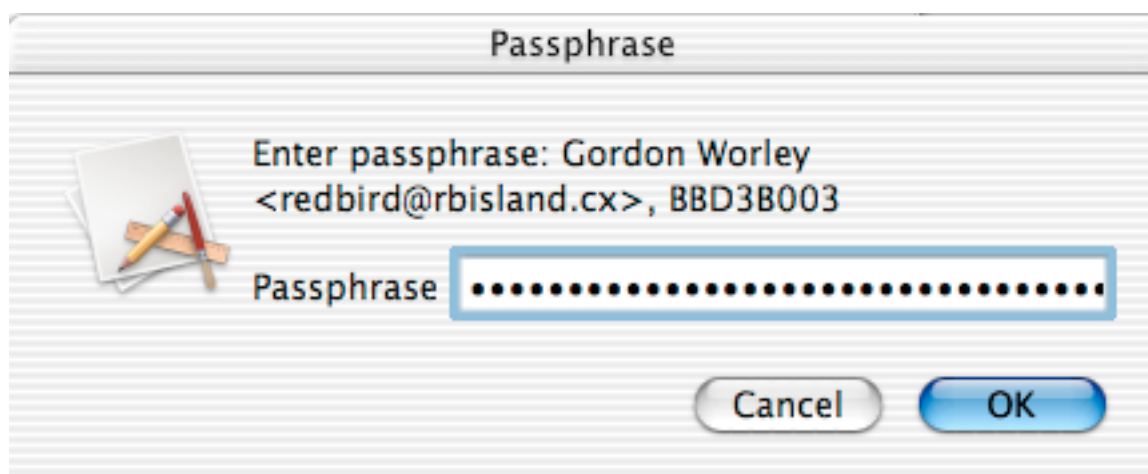


Figure 3.3: A `GPGPassphrasePanel` in action

Changing this method to use `GPGPassphrasePanel` also gives the user multiple tries to enter their password correctly, although this feature is not exclusive to `GPGPassphrasePanel`: you could write your own passphrase request interface that allowed for multiple tries. We use the third argument to change the message displayed to the user, to let them know if they are trying to enter their passphrase again after an unsuccessful try or if they are entering their passphrase for the first time. If you do not say that the user failed to enter their passphrase correctly when asking the second or third time, they may believe that it was a program rather than user error producing the additional requests.

## 3.7 Exercises

1. Let the user specify multiple recipients and signers.
2. Add a button to reverse the process: take cipher text as input, decrypt it to output.
3. Only display choices for signer key(s) that are capable of signing.



# Bibliography

- [1] Koch, Werner. "GPGME". nuP.org. 20 October 2002.  
<[http://www.gnupg.org/\(en\)/related\\_software/gpgme/index.html](http://www.gnupg.org/(en)/related_software/gpgme/index.html)>. (3 June 2003).
- [2] Yee, Ka-Ping. "User Interaction Design for Secure Systems". 3 December 2002.  
<<http://www.sims.berkeley.edu/~ping/sid/uidss.pdf>>. (17 June 2003).

# About the Author

Gordon Worley is the founder and former administrator of the Mac GPG Project. He is a senior at the University of Central Florida, earning a B.S. in Computer Science. He works at UCF's Writing Center and on various grants. He intends to earn a Ph.D. in computer science.

You can e-mail him at [redbird@mac.com](mailto:redbird@mac.com) or visit his Website at <http://homepage.mac.com/redbird/>.



# Colophon

This book was written using i $\text{\TeX}$ Mac and typeset with  $\text{\LaTeX}$ . Any oddities in the typesetting are the fault of the author (please report them or, better yet, suggest how to fix them).

Animals not on the cover of this book include a gnu, a duck, a platypus, a lama, a bald eagle, a golden retriever, a school of tilapia, a sloth, an etropus, and a tree frog.

# Index

You may be wondering where the index is. Sorry, no index is currently available, but it will exist in a future edition of this book.