The Use Of Secure Email In TRIADS

Simon Hatton

Contents

Acknowledgements………………………………………………………………… 4

1.0 Introduction………………………………………………………………… 5

1.1 What Is Secure Email?……………………………………………………… 5

2.0 How To Integrate PGP with TRIADS……………………………………….. 6

2.1 Create a PGP key pair for the examiner………………………………………. 6

2.2 Create a PGP key pair for every student…………………………………….… 6

2.3 Email encrypted/signed answers to the examiner…………………………….….. 7

2.4 Location of PGP/TRIADS files……………………………………………….. 9

2.5 Summary……………………………………………………………………. 9

3.0 Generating the examiner's PGP key………………………………………. 10

3.1 Install PGP………………………………………………………………….. 10

3.2 Key Generation Wizard……………………………………………………….. 10

3.3 Summary……………………………………………………………………. 11

4.0 Generating the students PGP keys………………………………………. 12

4.1 PGP Key Generation Command………………………………………………. 12

4.2 Using an Expect script to automatically create PGP keys…………………….. 12

4.3 Problems with the Expect script………………………………………………. 13

4.4 Using the PGP SDK to automatically create PGP keys………………………. 14

4.5 Summary……………………………………………………………………. 15

5.0 Integrating PGP with a TRIADS assessment…………………………………… 17

5.1 PGP Command Line…………………………………………………………... 17

5.2 Environment Variables used with PGP……………………………………….. 19

5.3 Running PGP from a C++ program…………………………………………… 20

5.4 Using input parameters with *pgp_triads*……………………………………… 20

5.5 Running PGP from a network drive…………………………………………... 21

5.6 Running PGP from within Authorware……………………………………….. 22

5.7 Buddy API………………………………………………………………….. 23

5.8 Incorporating the *pgp_triads* program with TRIADS………………………… 24

5.9 Not enough Random Data on PC's……………………………………………. 25

5.10 Using the PGP SDK to add random data……………………………………… 26

5.11 Summary……………………………………………………………………. 27

6.0 Running a PGP/TRIADS assessment……………………………………… 29

6.1 Preparation for the assessment………………………………………………… 29

6.2 Running the assessment……………………………………………………….. 30

6.3 Decrypting/Verifying the student's results…………………………………… 30

6.4 Summary…………………………………………………………………. 31
7.0 Conclusions and Future Improvements…………………………………………. 32
Appendix A…………………………………………………………………… 33
Appendix B…………………………………………………………………… 36
Appendix C…………………………………………………………………… 37
Appendix D…………………………………………………………………… 40
Appendix E…………………………………………………………………… 43
Appendix F…………………………………………………………………… 52
Appendix G…………………………………………………………………… 53
Appendix H…………………………………………………………………… 54
Appendix I…………………………………………………………………… 55
Appendix J…………………………………………………………………… 56
Appendix K…………………………………………………………………… 57
Appendix L…………………………………………………………………… 58

Acknowledgements

I would like to thank the following people for their continued help and support throughout this project:

JISC

The work for this project was funded by a grant from the Joint Information Systems Committee (JISC), and their support is greatly appreciated.

CIAD

The Centre for Interactive Assessment Development (CIAD), based at the University of Derby, are responsible for the development and maintenance of TRIADS. I would like to thank them for their help with the development of the TRIADS engine.

Dr. Alan Boyle

Alan Boyle is a lecturer in the Earth Sciences department at the University of Liverpool. Alan is the principal contact at the University of Liverpool for the TRIADS project. My thanks to Alan for his continued guidance and support throughout the project.

1.0 Introduction

The department of Earth Sciences at the University Of Liverpool have been running CBAs (Computer Based Assessments) since 1995. The department predominantly uses the TRIADS (TRipartite Interactive Assessment Delivery System) engine to help in the production of CBAs. TRIADS is a toolkit, built using Macromedia Authorware, specifically designed to facilitate the rapid and easy production of CBAs. Authorware is primarily used to create rich-media training applications, but it also provides the basic framework for creating CBAs.

The TRIADS system has been developed since 1992 at the University of Derby, and is currently being extended in collaboration with the University of Liverpool, and the Open University. TRIADS is developed and maintained at the University of Derby by CIAD (Centre for Interactive Assessment Development). For more information please visit http://www.derby.ac.uk/ciad/.

An assessment created with the TRIADS engine can be run on a PC, and upon completion the results can be emailed back to the examiner. The purpose of this project was to introduce the use of secure email within a TRIADS assessment.

1.1 What Is Secure Email?

Secure email offers a number of advantages over regular email. By default regular email only offers fairly weak security, however the use of secure email improves matters by providing the following benefits:

- Prevention of unauthorised reading of an email message
- Confirmation of the origin of the message
- Proof that the message has not been altered

There are a number of secure email software tools available, but for this project PGP (Pretty Good Privacy) was used. This is because PGP is currently the most widely deployed piece of software for protecting messages sent across the Internet. For a brief explanation of the concepts of PGP please see Appendix A.

The next chapter will begin by explaining how we planned to run PGP from within a TRIADS assessment.

2.0 How To Integrate PGP with TRIADS

Previously whenever an assignment was completed the student's answers were sent via email to the examiner. This only offered weak security on the email, so the solution proposed for this project was to use PGP to improve security by encrypting and signing the student's answers before they are emailed.

When the examiner received the email they will then be able to decrypt it to reveal the student's answers, and verify the signature to confirm the origin of the message and prove that it had not been altered.

Additionally any solutions that were achieved for this project had to satisfy the following conditions:

- The TRIADS assessments that are integrated with PGP would run on any Windows 2000 PC from within the University of Liverpool
- The students had to have no knowledge of the involvement of PGP within the assessment i.e. the use of PGP had to be completely hidden
- The signing/encrypting of the student's answers had to be automatic and required no input from the user

2.1 Create a PGP key pair for the examiner

In order for the student's answers to be encrypted to the examiner, the examiner generated their own public/private key pair beforehand. A key pair can be easily created by installing PGP on the examiner's PC, and then running through the key generation procedure.

2.2 Create a PGP key pair for every student

In order for the student's answers to be signed every student who was taking the assessment had their own individual public/private key pair. This presented a couple of problems, namely if the students had no knowledge of PGP being used then they cannot have been expected to create the key pair themselves. Also how can the answers be signed without asking the student to enter the passphrase for their private key?

Currently whenever CBAs are run at the University of Liverpool the students do not login to the PCs being used with their own username/password. This guards against the possibility that students could store relevant information under their own username.

Instead the examiner uses a collection of temporary usernames to login to each PC before the students arrive. An example of the form the temporary usernames may take is scese01, scese02, scese03 etc. These usernames can then be linked to seat numbers in a teaching centre, therefore allowing the examiner to know which computer each student was sat at. In addition only these temporary usernames have access to the CBA exam.

In order to link PGP with these temporary usernames a public/private key pair was created for each username beforehand. Therefore when the answers were signed PGP used the private key that had the same name as the username that was used to log onto the PC currently being used for the assessment.

The following list explains the stages that must be carried out in order for the student's answers to be correctly signed:

a) An examiner had a collection of temporary usernames, e.g. scese00 to scese99
b) The examiner created a public/private key pair for each temporary username. The passphrase for each private key is the same as the username
c) Before the students arrived for the assessment the examiner logged into each computer that was going to be used with one of the temporary usernames
d) The students then began the assessment
e) When each student completed the assessment TRIADS used PGP to sign their results using the correct private key, and passphrase. For example if the username scese33 was used by the examiner to initially logon to a PC, then the private key, and passphrase, scese33 were used to sign the student's answers on that PC.

2.3 Email encrypted/signed answers to the examiner

Once the answers have been signed using the student's private key, and encrypted using the examiner's public key, then the encrypted/signed answers were emailed to the examiner. The following list, along with Fig. 2.1, gives an example of the stages that are involved in this process:

a) A student completes the TRIADS assessment

b) TRIADS saves their answers into a temporary file

c) PGP takes the answers file, signs it with the correct student's private key, and encrypts it with the examiner's public key. This creates a new temporary file

d) TRIADS copies the contents of this new file into the body of an email message

e) TRIADS sends this email to the examiner

f) TRIADS deletes the temporary files

g) Upon receipt the examiner uses PGP to decrypt/verify the email to reveal the student's answers

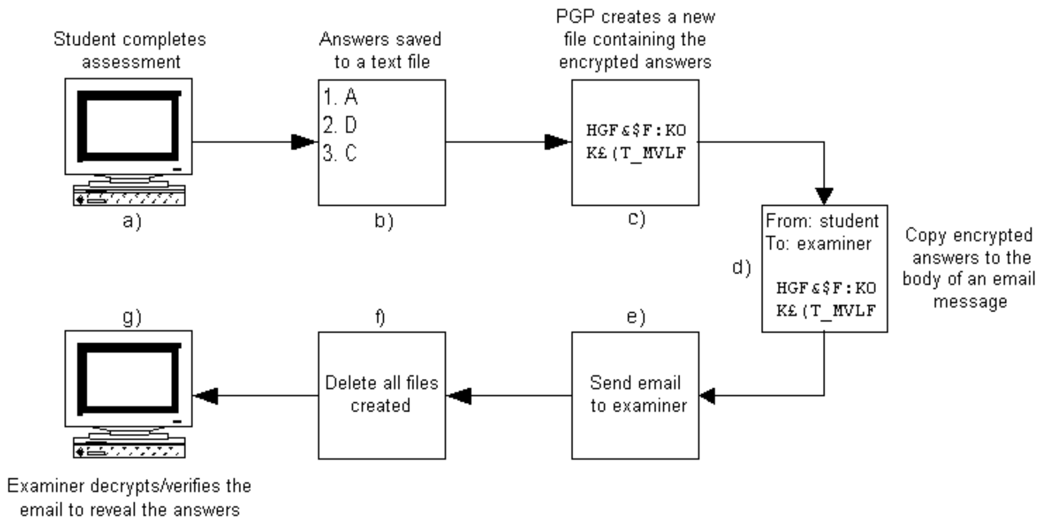

Student completes assessment
a)
Answers saved to a text file
1. A
2. D
3. C
b)
PGP creates a new file containing the encrypted answers
HGF&$F:KO
K£(T_MVLF
c)
d)
From: student
To: examiner
HGF&$F:KO
K£(T_MVLF
Copy encrypted answers to the body of an email message
e)
Send email to examiner
f)
Delete all files created
g)
Examiner decrypts/verifies the email to reveal the answers

Fig. 2.1 – Integrating PGP with a TRIADS assessment

The preceding list, along with Fig. 2.1, shows that PGP is only used in steps c) and g). TRIADS carries out all of the other operations, such as, saving the students answers to a file, emailing the answers to the examiner etc.

2.4 Location of PGP/TRIADS files

In order for PGP/TRIADS to be run from any Windows 2000 PC it was important that all files required could be easily accessed from these machines. Therefore all of the necessary files for the assessment were stored on a University network drive.

Whenever one of the temporary usernames was used to log into a PC then the login procedure automatically mapped to the network drive where the PGP/TRIADS files were stored. This meant that the assessment, and PGP, could be easily executed.

2.5 Summary

The aim of this project was to incorporate the use of PGP into a TRIADS assessment so that a student's results were signed, and encrypted, before they were emailed to the examiner.

Before an assessment took place the examiner created their own PGP key pair. A public/private key pair was also generated for each username from a list of temporary usernames. Additionally all files that are used by either PGP/TRIADS were copied onto a University network drive so that they can be accessed from any Windows 2000 PC.

On the day of the assessment, before the students arrived, the examiner logged into each PC being used with one of the temporary usernames. When a student completes the assessment their answers were signed by using the private key that had the same name as the username that was used for the CBA. The student's answers were also encrypted using the examiner's public key.

These signed/encrypted results were copied to the body of an email message, which was then sent to the examiner. Upon retrieval of the email message the examiner used PGP to decrypt the answers, and verify the student's identity.

The implementation of the concepts described in this chapter are covered in chapters 3, 4, and 5.

3.0 Generating the examiner’s PGP key

In order for the student’s answers to be encrypted, the examiner must have their own PGP key pair. In addition, the examiner must have PGP installed on the PC where they intend to decrypt and verify the emails containing the students’ answers. This chapter gives a brief description of the stages involved in the installation and setup of PGP.

3.1 Install PGP

For this project PGP version 6.5.8 for Windows 2000 was installed on the examiner’s PC. PGP also includes various plug-ins for a number of popular email programs. A plug-in is an additional piece of software that bridges the gap between PGP and an email package. The plug-in adds a couple of extra buttons to the interface of the email package to allow the user to easily access the functions of PGP. This makes decryption and verification of emails a lot easier for the examiner.

Fig. 3.1 shows an example of the decrypt/verify button that is added to the interface in Eudora when the Eudora PGP plug-in has been installed.

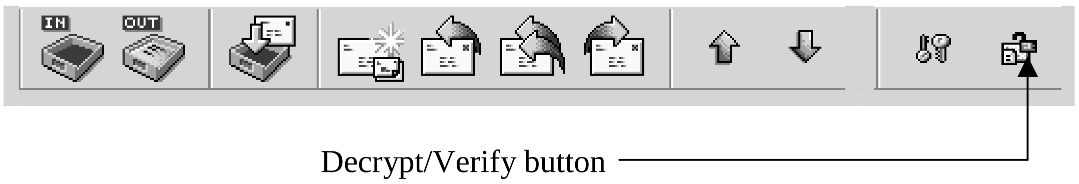


Fig. 3.1 – PGP plug-in buttons

3.2 Key Generation Wizard

Once PGP had been installed on the examiners PC then they needed to create their own key pair. After PGP has been installed the Key Generation Wizard is automatically launched, as shown in Fig. 3.2.

The Key Generation Wizard takes the user through the process of specifying the details of their key pair. The details that the user has to input are:

a) The name associated with the key pair, e.g. Fred Bloggs

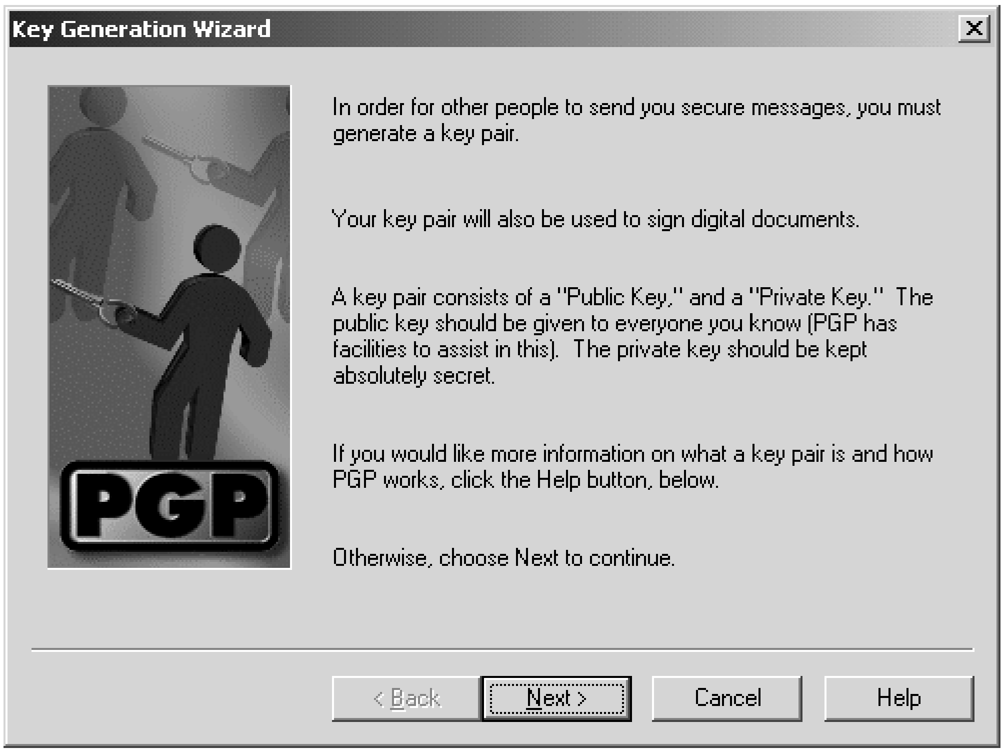
b) The email address associated with the key pair (optional), e.g. fred@bloggs.com

c) The size of the key pair. Please note that all key pairs generated for use in the project were 2048/1024 DH/DSS key pairs.

d) The date when the key pair expires

e) The passphrase used to access the private key

Once all of these details have been entered then the key pair was generated.


Key Generation Wizard

In order for other people to send you secure messages, you must generate a key pair.

Your key pair will also be used to sign digital documents.

A key pair consists of a "Public Key," and a "Private Key." The public key should be given to everyone you know (PGP has facilities to assist in this). The private key should be kept absolutely secret.

If you would like more information on what a key pair is and how PGP works, click the Help button, below.

Otherwise, choose Next to continue.

Fig. 3.2 – PGP Key Generation Wizard

PGP keys are stored in files known as keyrings. The public key is stored in a public keyring file (pubring.pkr), and the private key is stored in a private keyring file (secring.skr).

3.3 Summary

For the student’s answers to be encrypted to the examiner, the examiner must have already created their own PGP key pair. A PGP key pair was generated for the examiner by first installing PGP 6.5.8 for Windows 2000 on their PC, and then running through the Key Generation Wizard. In addition, a PGP plug-in for the examiner’s preferred email package was also installed. This made the process of decrypting/verifying emails considerably easier.

4.0 Generating the students PGP keys

In order for the student's answers to be signed each student who is taking part in the TRIADS assessment needed their own PGP key pair. However as the involvement of PGP was hidden from the students the key pairs needed to be created beforehand.

For each assessment a list of temporary usernames was used, for example, scese00 through to scese99. A PGP key pair therefore had to be created for every temporary username in this list.

The major problem was that it would take a long time for the examiner to create all the key pairs manually, especially if there are hundreds of students sitting the exam. Therefore there needed to be some form of automatic PGP key generation that would make the job of creating the student's keys as quick and as simple as possible.

4.1 PGP Key Generation Command

The PGP command line version 6.5.8 has a key generation command (`pgp -kg`). When this command is entered the user is asked a series of questions about the type of key they want to create, and once the questions have been answered then the key is generated. An example of the questions that are asked by the key generation command can be seen in Appendix B.

4.2 Using an Expect script to automatically create PGP keys

Expect is a program that lets the user control interactive applications. Interactive applications are ones where the user is prompted with a question, and they are then expected to enter a keystroke in response. By using Expect it is possible to write a script that can be used to automate these interactions i.e. the script expects certain questions to be asked, and when they are it sends the correct answer. Therefore an Expect script, *key_gen.exp*, was written that automatically ran through the PGP key generation command.

The Expect script also required some initial data, supplied by the examiner, before the keys could be created. The data required was:

a) The first and last usernames that require a key pair

b) The date when the keys should expire, e.g. 01/01/2010

In order to obtain this data from the user a C program, *genkey*, was written. This program asked the examiner to enter the name of a file where the list of temporary usernames were to be stored, the first and last usernames that required a key pair, and the date when the keys will expire. For example, the examiner inputs the data shown in Fig. 4.1.

Input Data	**Value**
Temporary Filename	TEMPKEYS.TXT
First Username	scese00
Last Username	scese99
Expiry Date	01/01/2010

Fig. 4.1 – Input data required by the *genkey* program

Given this data, *genkey* created a list of all usernames from scese00 through to scese99, and output the list into a file called “TEMPKEYS.TXT”. The program also calculated the number of days between the current date and 01/01/2010.

Finally *genkey* called the Expect script *key_gen.exp* with the name of the file containing the list of usernames, and the number of days the key pairs are valid for as input parameters. The code for the *genkey* program can be seen in Appendix C, and the code for the *key_gen.exp* script can be seen in Appendix D.

4.3 <u>Problems with the Expect script</u>

The combination of the *genkey* program and the *key_gen.exp* script was used to automatically create PGP key pairs. However there were a number of problems with this solution:

a) The *key_gen.exp* script took a long time to create the keys. For example, it took a number of hours for the program to be able to create the 100 PGP keys scese00 through to scese99.

b) The *key_gen.exp* script was unreliable, for example, when creating the keys scese00 through to scese99 it crashed on a number of occasions. This meant that the examiner had to restart the *genkey* program to create the remaining PGP keys.

c) The error checking on the user input for the *genkey* program was not sufficient i.e. if the examiner entered an invalid expiry date then the program would crash, or create an incorrect key.

d) PGP requires a certain amount of random data so that it can generate unique keys. This random data is gathered from various system sources, such as mouse position, timings, keystrokes etc.

   When using the PGP Key Generation command the user has to create this random data themselves by moving the mouse or pressing keys on the keyboard.

   In *key_gen.exp* the random data is created automatically by inputting a random number between 0 and 9, every 1 or 2 seconds. Unfortunately this method does not really offer sufficient randomness for generating PGP keys.

Therefore as a result of the problems faced with the C program and the Expect script a better solution was needed.

4.4 Using the PGP SDK to automatically create PGP keys

The PGP SDK (Software Developer's Kit) is a collection of C++ libraries that allow a developer to add PGP functions into their programs. The PGP SDK provides functions that allow the user to create their own PGP keys. Therefore a C++ program was written as an alternative to the combination of the existing *genkey* program, and the *key_gen.exp* script.

This new version of the *genkey* program was a Windows 32-bit console application that could be run from the command prompt. The program still required the user to input initial data about the first and last usernames, and the date when the keys will expire. However the need for the user to enter a temporary filename was removed, instead the filename was specified within the actual source code. The code used for the new version of the *genkey* program can be seen in Appendix E, and Fig. 4.2 shows an example of the output from the program while it is running.

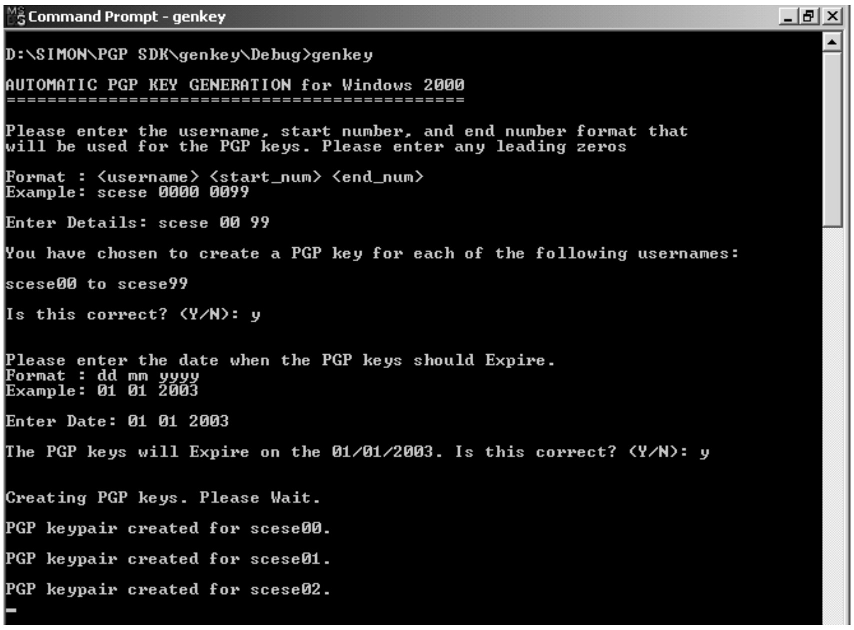
This new C++ version of the *genkey* program offered the following benefits over the Expect script:

a) The *genkey* program ran much faster. For example, it took approximately 15 minutes for the program to be able to create the 100 PGP keys.

b) The program was a lot more a stable, and did not crash when creating the PGP keys.

c) The error checking on the data input by the user is greatly improved. It was now no longer possible to enter an invalid expiry date.

d) The random data, required by PGP to create unique keys, was now automatically collected by generating random numbers between 0 and 999999. However whilst an improvement over *key_gen.exp* the generation of random numbers for the key creation was still flawed.



```
Command Prompt - genkey

D:\SIMON\PGP SDK\genkey\Debug>genkey

AUTOMATIC PGP KEY GENERATION for Windows 2000
=============================================

Please enter the username, start number, and end number format that
will be used for the PGP keys. Please enter any leading zeros

Format : <username> <start_num> <end_num>
Example: scese 0000 0099

Enter Details: scese 00 99

You have chosen to create a PGP key for each of the following usernames:

scese00 to scese99

Is this correct? (Y/N): y


Please enter the date when the PGP keys should Expire.
Format : dd mm yyyy
Example: 01 01 2003

Enter Date: 01 01 2003

The PGP keys will Expire on the 01/01/2003. Is this correct? (Y/N): y


Creating PGP keys. Please Wait.

PGP keypair created for scese00.

PGP keypair created for scese01.

PGP keypair created for scese02.
_
```

Fig. 4.2 – Output from the genkey program

4.5 Summary

In order for the student’s answers to be signed each student taking part in the TRIADS assessment required their own PGP key pair. As there are potentially hundreds of key pairs that need to be created the process needed to be as simple and as fast as possible.

Initially a combination of a C program, and an Expect script was used. Unfortunately this solution was very slow, and very unstable. An improved solution was found by writing a C++ program using the PGP SDK. This new solution was a lot faster, and more stable, than the previous Expect script.

5.0 Integrating PGP with a TRIADS assessment

This chapter describes the work that was carried out so that PGP could sign and encrypt a student's answers from within a TRIADS assessment. For this stage it was important that the following conditions were met:

- The students should have no knowledge of PGP's involvement. Therefore there should be no pop-up windows, confirmation boxes etc. PGP should run hidden in the background
- The encryption/signing of the answers file had to be completely automatic. The student must not have to enter a passphrase in order to access the private key
- The assessment/PGP must be able to run from any Windows 2000 PC anywhere in the University

5.1 PGP Command Line

In order to run PGP from within a TRIADS assessment the PGP command line version 6.5.8 for Windows 2000 was initially used. This program can take a file, sign and encrypt it, and produce an encrypted version of the original file.

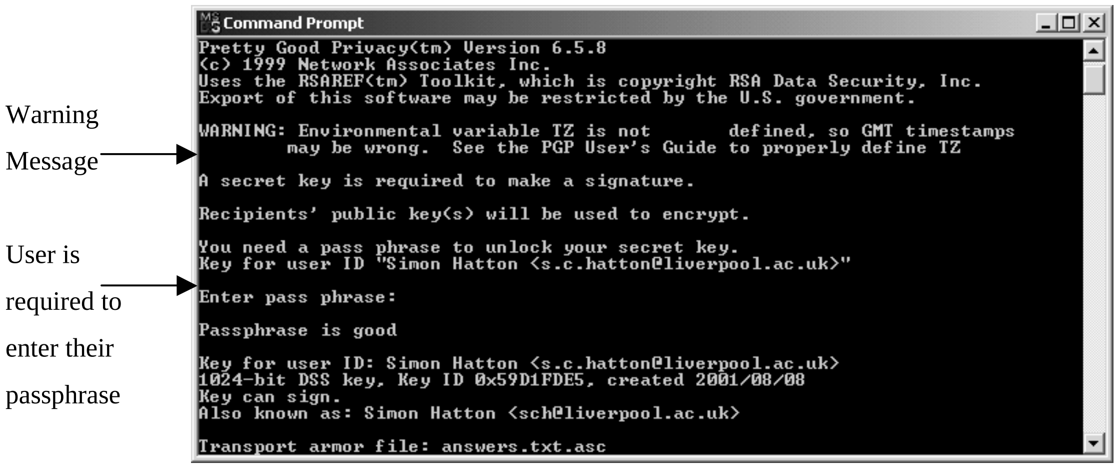
By using the PGP command line a user can specify the name of the file that they want to sign/encrypt, the private key that should be used to sign the file, and the public key used to encrypt the file. For example, below is a PGP command that takes a file called "answers.txt", signs it with the private key belonging to the user "student", and encrypts it with the public key belonging to the user "examiner".

```
pgp –seat answers.txt examiner –u student
```

This command will create a new file containing the encrypted/signed answers. A full explanation of the parameters used for this command are shown in Fig. 5.1, and the output from running the command can be seen in Fig. 5.2.

Parameter	Description
-**S**eat	Sign the input file
-s**E**at	Encrypt the input file
-se**A**t	Convert the output file to ASCII-armoured format. This format is suitable when sending the message through email channels
-sea**T**	Identifies the input file as a text file
answers.txt	Input file containing the student's answers
examiner	User ID of recipient
-u	Identify the key to use for signing
student	User ID of signing key

Fig. 5.1 – PGP command line arguments



```
Pretty Good Privacy(tm) Version 6.5.8
(c) 1999 Network Associates Inc.
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Export of this software may be restricted by the U.S. government.

WARNING: Environmental variable TZ is not       defined, so GMT timestamps
         may be wrong.  See the PGP User's Guide to properly define TZ

A secret key is required to make a signature.

Recipients' public key(s) will be used to encrypt.

You need a pass phrase to unlock your secret key.
Key for user ID "Simon Hatton <s.c.hatton@liverpool.ac.uk>"

Enter pass phrase:

Passphrase is good

Key for user ID: Simon Hatton <s.c.hatton@liverpool.ac.uk>
1024-bit DSS key, Key ID 0x59D1FDE5, created 2001/08/08
Key can sign.
Also known as: Simon Hatton <sch@liverpool.ac.uk>

Transport armor file: answers.txt.asc
```

Fig. 5.2 – PGP command line output

While the output shown in Fig. 5.2 does sign and encrypt a file there are a couple of problems with the PGP command:

a) Whenever PGP is run the following warning message is output:

   "Environmental variable TZ is not defined, so GMT timestamps may be wrong. See the PGP User's Guide to properly define TZ"

   This warning also caused the computer to output a "beep" sound. This drew attention to the fact that PGP was being used.

b) PGP asked the user to type in their passphrase so that the file can be signed. In order for the process to be automated this user interaction had to be removed

5.2 Environment Variables used with PGP

To eliminate the problems highlighted in section 5.1 the environment variables, as shown in Fig. 5.3, can be used:

Variable	**Description**
PGPPASS	The PGPPASS environment variable avoids the user being prompted for a passphrase by storing a passphrase. Example: `SET PGPPASS=hello world` This eliminates the prompt for the passphrase if the passphrase for the private key is “hello world”
TZ	PGP has a timestamp for keys and signatures in Greenwich Mean Time (GMT). The environment variable TZ can be used to specify the number of hours to add to the system time function to get GMT. Example: `SET TZ=GMT0BST` This command eliminates the warning message, and the beep, that PGP was producing every time it was run.

Fig. 5.3 – Environment variables

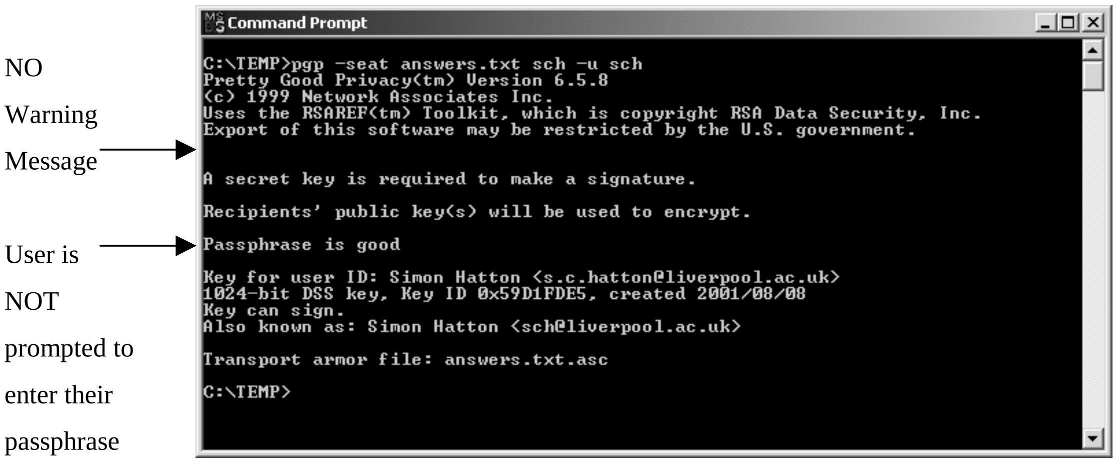
The two environment variables shown in Fig. 5.3 are initialised by typing them into a command window before running the PGP command line. For example, the 2 environment variables are typed into the command prompt, and then PGP is executed by using the command shown in section 5.1:

```
SET PGPPASS=my passphrase

SET TZ=GMT0BST

pgp –seat answers.txt examiner –u student
```

This results in PGP being executed, the output of which is shown in Fig. 5.4.



```
C:\TEMP>pgp -seat answers.txt sch -u sch
Pretty Good Privacy(tm) Version 6.5.8
(c) 1999 Network Associates Inc.
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Export of this software may be restricted by the U.S. government.

A secret key is required to make a signature.

Recipients' public key(s) will be used to encrypt.

Passphrase is good

Key for user ID: Simon Hatton <s.c.hatton@liverpool.ac.uk>
1024-bit DSS key, Key ID 0x59D1FDE5, created 2001/08/08
Key can sign.
Also known as: Simon Hatton <sch@liverpool.ac.uk>

Transport armor file: answers.txt.asc

C:\TEMP>
```

Fig. 5.4 – Improved PGP command line output

As can be seen in Fig. 5.4 the warning message, and the prompt for the passphrase, have been removed by setting the PGPPASS, and TZ, environment variables.

5.3 Running PGP from a C++ program

In order to set the PGPPASS, and TZ, environment variables before the PGP command line is run a script needed to be written to execute the 3 commands shown in section 5.2. Therefore a C++ program, *pgp_triads*, was written to carry out these actions.

Using the C++ command *_putenv* the environment variables were initialised, and the *system* command was used to launch the PGP command line from within the program.

The code used for *pgp_triads* can be seen in Appendix F. The output from running this program is shown in Fig. 5.4.

5.4 Using input parameters with *pgp_triads*

A problem with the initial version of the *pgp_triads* program was that the names of the file to encrypt, the public key, the private key, and the passphrase were all specified directly within the code. This was a problem because these details needed to be changed each time the program is run.

To avoid this problem the program was updated so that these identifiers were specified as command line arguments when the program is run. For example the following parameters were now used every time *pgp_triads* was executed:

```
pgp_triads <answers_file> <recipient> <sender>
```

This command will run *pgp_triads*, but now the file specified as the <answers_file> parameter will be signed/encrypted. The private key belonging to the user specified as the <sender> parameter will be used to sign the file, and the public key belonging to the user specified as the <recipient> parameter will be used to encrypt the file. The updated version of the program can be seen in Appendix G.

5.5 <u>Running PGP from a network drive</u>

Having written a program that could run PGP to automatically sign and encrypt a file we then had to test that the PGP command line, and *pgp_triads*, could be successfully run when stored on a University network drive.

To accomplish this the PGP command line, the *pgp_triads* program, and the PGP keyrings were copied to a network drive that could be mapped to from any Windows 2000 PC within the University.

Running the *pgp_triads* program from a network drive highlighted a problem that had not been previously encountered. Whenever the program was executed the following error message was being output, and PGP was failing to encrypt and sign the file:

```
"CMD.EXE was started with '\\U507\ES\APPS\1-ESGY105-TRIAL' as
the current directory path. UNC paths are not supported.
Defaulting to Windows directory"
```

The problem here was that the Windows directory was being used as the default directory for PGP. This caused a problem because the keyring files that PGP needs are not stored in the Windows directory so PGP couldn’t find them, and as a result the encrypted/signed document was not being created.

In order to fix this problem a new command line argument was used when *pgp_triads* was run. This new parameter specified the location where *pgp_triads*, the PGP command line, and the PGP keyrings were stored. Therefore the parameters now used to call the *pgp_triads* program are:

```
pgp_triads <current_dir> <answers_file> <recipient> <sender>
```

Additionally the PUBRING, and SECRING, command line parameters for PGP were also used to specify the exact location of the public and private keyrings files. An example of the code used for this updated version of *pgp_triads* can be seen in

Appendix H. Once these changes were made the updated version of the program was executed again, and this time the program ran successfully.

5.6 Running PGP from within Authorware

Having written a program that could be successfully run from a network drive to sign and encrypt a file, then we needed to be able to run the *pgp_triads* program from within a TRIADS assessment. Authorware had a command available called JumpOutReturn. This command can be used to launch another program from within Authorware, and keep Authorware open in the background while the other application is being executed. JumpOutReturn was therefore used to run the *pgp_triads* program from within a TRIADS assessment.

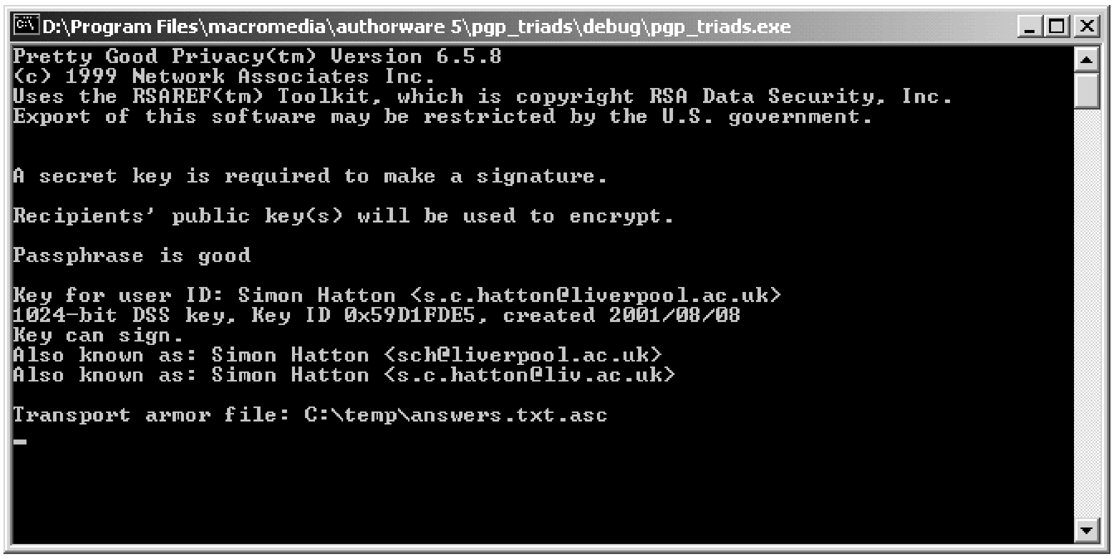
The syntax for the JumpOutReturn command is:

```
JumpOutReturn ("program", ["document"])
```

The first parameter *program* identifies the application that is to be executed. The optional second parameter *document* is used to pass any arguments to the *program* parameter. For example:

```
JumpOutReturn ("pgp_triads", "\\U507\ES\ answers.txt examiner student")
```

Running this JumpOutReturn command from within an Authorware application opens a command window, as shown in Fig. 5.5.



```
D:\Program Files\macromedia\authorware 5\pgp_triads\debug\pgp_triads.exe
Pretty Good Privacy(tm) Version 6.5.8
(c) 1999 Network Associates Inc.
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Export of this software may be restricted by the U.S. government.


A secret key is required to make a signature.

Recipients' public key(s) will be used to encrypt.

Passphrase is good

Key for user ID: Simon Hatton <s.c.hatton@liverpool.ac.uk>
1024-bit DSS key, Key ID 0x59D1FDE5, created 2001/08/08
Key can sign.
Also known as: Simon Hatton <sch@liverpool.ac.uk>
Also known as: Simon Hatton <s.c.hatton@liv.ac.uk>

Transport armor file: C:\temp\answers.txt.asc
_
```

Fig. 5.5 – Using JumpOutReturn to execute *pgp_triads*

Although the JumpOutReturn command can be used to run *pgp_triads* from within Authorware, the command window, as shown in Fig. 5.5, pops up at the front of the screen whenever the JumpOutReturn command is used. This causes a problem, as the *pgp_triads* program should run hidden in the background. Therefore, an alternative to JumpOutReturn needed to be found.

5.7 Buddy API

Following some searching on the WWW some additional functions for Authorware were discovered, collectively known as the Buddy API. The Buddy API is a library of over 100 additional commands for use with an Authorware application, and it is free to educational institutions for non-commercial use. For more information please visit http://www.mods.com.au/budapi.

By using the Buddy API it was possible to use the additional commands that it offered within the TRIADS assessment. In particular, the command baRunProgram could be used to run the *pgp_triads* program, instead of the JumpOutReturn command. The syntax for the baRunProgram command was:

```
baRunProgram ("program", state, wait)
```

The table shown in Fig. 5.6 gives an explanation of the parameters that are used by the baRunProgram command.

The following command gives an example of how the baRunProgram command can be used to execute the *pgp_triads* program from within Authorware:

```
baRunProgram ("pgp_triads \\U507\ES\ answers.txt examiner student", hidden, TRUE)
```

Using this command to launch the *pgp_triads* program instead of the JumpOutReturn command meant that the PGP command line was executed without asking the user for any input, and it was completely hidden.

Parameter	Description
program	Identifies the application that is to be executed together with any input parameters.
state	Defines how the program will run, can be one of the following: "Normal" Program runs in its usual state "Hidden" Program is not visible "Maximised" Program runs in a maximised window "Minimised" Program runs as a minimised icon
wait	Set to TRUE if Authorware is to wait for the external program to finish before continuing. Otherwise set to FALSE.

Fig. 5.6 – baRunProgram command parameters

5.8 Incorporating the *pgp_triads* program with TRIADS

Having worked out how to execute PGP automatically from a network drive, and how to run it hidden from within Authorware, the next stage was to incorporate the *pgp_triads* program into a TRIADS assessment, and then test that the assessment would run on any Windows 2000 PC in the University.

In order to run *pgp_triads* from within a TRIADS assessment a few alterations were made to the engine. These changes were made by CIADS (Centre for Interactive Assessment Development at the University of Derby), who are responsible for the development and maintenance of TRIADS.

By adding new icons to the TRIADS engine this meant that new code could be easily added so that *pgp_triads* could be setup, and executed from within an assessment. Fig. 5.7 gives an indication of where the *pgp_triads* program was setup to run from within an assessment.

The TRIADS assessment actually sends three different emails to the examiner. The first email is simply an overall score. The second message contains all of the individual question scores together with the overall score, and the third is a complete transcript of the student's answers. Obviously all three of these messages had to be encrypted and signed.

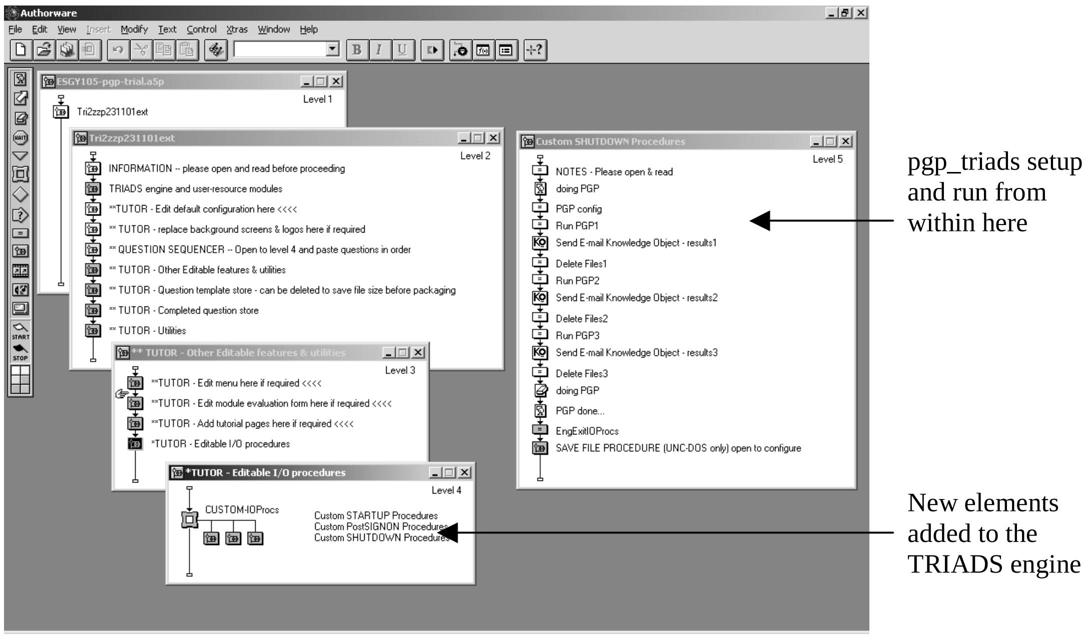

Authorware
File Edit View Insert Modify Text Control Xtras Window Help
ESGY105-pgp-trial.a5p
Level 1
Tri2zzp231101ext
Tri2zzp231101ext
Level 2
INFORMATION -- please open and read before proceeding
TRIADS engine and user-resource modules
**TUTOR - Edit default configuration here <<<<
** TUTOR - replace background screens & logos here if required
** QUESTION SEQUENCER -- Open to level 4 and paste questions in order
** TUTOR - Other Editable features & utilities
** TUTOR - Question template store - can be deleted to save file size before packaging
** TUTOR - Completed question store
** TUTOR - Utilities
** TUTOR - Other Editable features & utilities
Level 3
**TUTOR - Edit menu here if required <<<<
**TUTOR - Edit module evaluation form here if required <<<<
**TUTOR - Add tutorial pages here if required <<<<
*TUTOR - Editable I/O procedures
*TUTOR - Editable I/O procedures
Level 4
CUSTOM-IOProcs
Custom STARTUP Procedures
Custom PostSIGNON Procedures
Custom SHUTDOWN Procedures
Custom SHUTDOWN Procedures
Level 5
NOTES - Please open & read
doing PGP
PGP config
Run PGP1
Send E-mail Knowledge Object - results1
Delete Files1
Run PGP2
Send E-mail Knowledge Object - results2
Delete Files2
Run PGP3
Send E-mail Knowledge Object - results3
Delete Files3
doing PGP
PGP done...
EngExitIOProcs
SAVE FILE PROCEDURE (UNC-DOS only) open to configure
pgp_triads setup and run from within here
New elements added to the TRIADS engine

Fig. 5.7 – Incorporating pgp_triads with the TRIADS engine

Using the updated TRIADS engine the examiner was able to write the code that would be used to setup, and execute the *pgp_triads* program. An example of the code used in TRIADS to setup PGP can be seen in Appendix I.

Once the parameters had been defined then the *pgp_triads* program was executed. An example of the code used in TRIADS to launch the program can be seen in Appendix J. After the *pgp_triads* program was executed TRIADS emailed the encrypted/signed results to the examiner.

Finally, once the email had been sent then TRIADS deleted any temporary files created by the program, an example of the code for this can be seen in Appendix K.

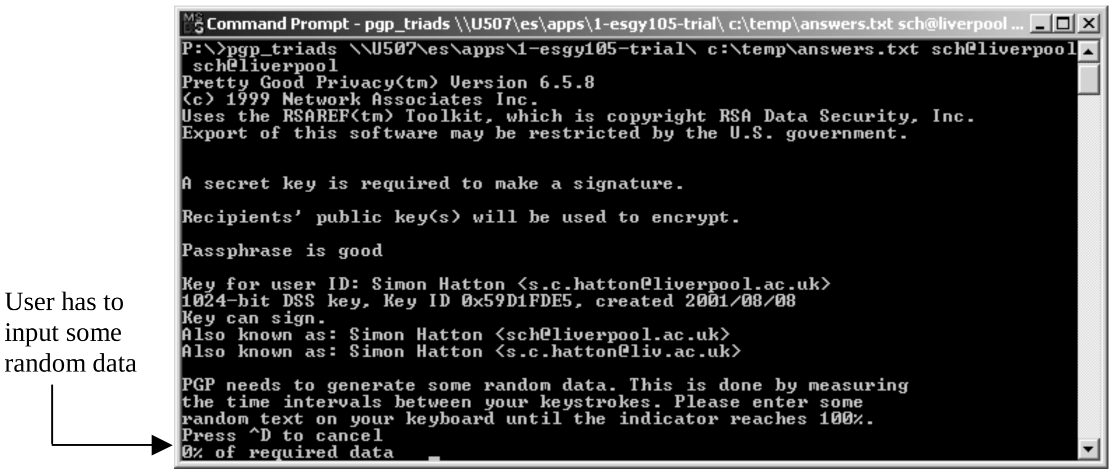
Now that *pgp_triads* had been setup to run from within a TRIADS assessment the final stage was to ensure that the assessment could be run on any Windows 2000 PC within the University.

5.9 <u>Not enough Random Data on PC's</u>

In order to test the *pgp_triads* program, and the PGP command line, a test TRIADS assessment was generated that would call PGP to encrypt/sign the student's answers.

To test this assessment in a University teaching centre, all required files (TRIADS assessment file, PGP command line, the *pgp_triads* program, and the public/private keyrings) were copied to a network drive.

The assessment appeared to run correctly on the teaching centre PCs, but afterwards the examiner had been sent 3 blank email messages instead of the student’s encrypted/signed results. Further investigation into the problem revealed the *pgp_triads* program was expecting some input from the user, as shown in Fig. 5.8.


Command Prompt - pgp_triads \\U507\es\apps\1-esgy105-trial\ c:\temp\answers.txt sch@liverpool ...
User has to input some random data

```
P:\>pgp_triads \\U507\es\apps\1-esgy105-trial\ c:\temp\answers.txt sch@liverpool
 sch@liverpool
Pretty Good Privacy(tm) Version 6.5.8
(c) 1999 Network Associates Inc.
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Export of this software may be restricted by the U.S. government.


A secret key is required to make a signature.

Recipients' public key(s) will be used to encrypt.

Passphrase is good

Key for user ID: Simon Hatton <s.c.hatton@liverpool.ac.uk>
1024-bit DSS key, Key ID 0x59D1FDE5, created 2001/08/08
Key can sign.
Also known as: Simon Hatton <sch@liverpool.ac.uk>
Also known as: Simon Hatton <s.c.hatton@liv.ac.uk>

PGP needs to generate some random data. This is done by measuring
the time intervals between your keystrokes. Please enter some
random text on your keyboard until the indicator reaches 100%.
Press ^D to cancel
0% of required data
```

Fig. 5.8 – Not enough random data

The problem was that PGP required a certain amount of random data before it could encrypt/sign a file. This was similar to the way that PGP required random data in order to generate unique keys. This problem had not been previously encountered because the PGP command line had always been run on a PC that had PGP installed on it. It was only in testing on a teaching centre PC, that did not have PGP installed locally, that the problem become apparent.

This was a potentially serious problem. We didn’t want to install a local copy of PGP onto every machine, nor did we want the students to have to input random data so their answers could be encrypted.

5.10 Using the PGP SDK to add random data

In order to fix the problem highlighted in section 5.9, there needed to be some way to increase the random pool of data on the teaching centre PCs, so that PGP could sign and encrypt the student’s answers automatically.

One possible solution was to use the PGP SDK. In particular there are a number of functions available that allow a developer to check the amount of data in the random pool, and increase it automatically where necessary.

At first we thought it would be possible to add code to increase the global random pool of data to the existing *pgp_triads* program, and then call the PGP command line.

However this solution did not appear to work as it resulted in the same output as shown in Fig. 5.9. Therefore the only way to increase the random pool of data, and encrypt/sign the answers file, was to include all of the PGP functionality in one *pgp_triads* program. The code for the new version of the *pgp_triads* program can be seen in Appendix L.

Before trying to sign and encrypt the file containing the answers, the program checked to see if the random pool contained enough data. If it didn't then the random pool was increased. Once this had been done then the answers file was signed, and encrypted.

This final version of the *pgp_triads* program was successfully tested on a number of different teaching centre PCs. Each time the trial assessment was run the *pgp_triads* program encrypted and signed the answers, and TRIADS then sent the encrypted answers in the body of an email to the examiner. The execution of the program also required no input from the user, and it remained completely hidden. Upon receipt of the emails, the examiner was able to decrypt the answers, and verify the student's identity.

5.11 Summary

In order to execute PGP from within a TRIADS assessment the PGP command line 6.5.8 was originally used. A C++ program, *pgp_triads*, was written that setup and launched the PGP command line automatically.

In order to execute PGP from within a TRIADS assessment, so that it was completely hidden from the students, the command baRunProgram was used from the Buddy API. Additionally some alterations were made to the TRIADS engine by CIAD to allow the setup, and execution, of the *pgp_triads* program.

However when it came to executing the PGP command line from within a TRIADS assessment on a teaching centre PC the user was required to enter some random data so that the answers could be signed and encrypted.

Therefore the PGP SDK was used as an alternative to the PGP command line. Using the SDK an updated version of the *pgp_triads* program was written that ensured there was sufficient random data to be able to sign and encrypt the answers. The program then signed, and encrypted, the file containing the student's answers.

Once *pgp_triads* had finished then the student's signed/encrypted results were sent to the examiner by TRIADS in the body of an email message. Upon receipt the examiner was able to successfully decrypt and verify the email to reveal the student's answers.

Now that the implementation was completed the final stage was to run a PGP/TRIADS assessment with a group of students. The details of such a test are covered in the next chapter.

6.0 Running a PGP/TRIADS assessment

Having come up with a method of using PGP to encrypt, and sign, a student's answers to an assessment, it was time to test out the solution with an actual TRIADS assessment.

The assessment used was an Earth Sciences exam held on the 14th January 2002. In total there were 80 students who were taking the exam, which comprised of 84 questions.

After each student completed the assessment, their results were signed and encrypted. The results were then sent in the body of an email message to the examiner.

6.1 Preparation for the assessment

Before the assessment was run the following preparations had to be made:

- The examiner installed PGP onto their PC and created their own PGP key pair. They also installed a PGP email plug-in so that they can easily decrypt/verify the messages.
- The program *genkey* was used to create the temporary PGP key pairs for the students. In total 100 PGP key pairs were created for the usernames scese00 through to scese99. All key pairs were set to expire on the 14th February 2002. The program created 2 new keyring files containing all of the keys (pubring.pkr, and secring.skr)
- The examiner imported their public key into the public keyring file (pubring.pkr) that was created by the *genkey* program. This was because in order for the student's answers to be encrypted PGP must have access to the examiner's public key
- A TRIADS assessment was created that called the *pgp_triads* program with the correct parameters in order to encrypt and sign the student's answers
- All files required by the TRIADS assessment, including the *pgp_triads* program, and the keyring files (pubring.pkr, and secring.skr), created by the *genkey* program were copied into a directory on a University network drive.

6.2 Running the assessment

On the day of the exam the following stages were carried out:

- The examiner logged into every computer that would be used for the assessment, using the temporary usernames, before allowing any students into the examination room
- The students were then admitted to the teaching centre, and began the assessment
- Once the assessment was completed the *pgp_triads* program automatically encrypted, and signed, the student's answers, and then emailed them to the examiner.

6.3 Decrypting/Verifying the student's answers

Once the assessment was completed the examiner returned to their PC to decrypt/verify the email messages that they had been sent. The following stages were involved in the decryption/verification process:

- The examiner altered the PGP settings on their PC so that PGP will use the public keyring file (pubring.pkr) that was created by the *genkey* program. This is because in order for the student's signatures to be verified PGP must have access to the student's public keys.
- The examiner was then able to decrypt each message using their private key, and verify the signature using the student's public keys. As 80 students took part in the assessment, and there are 3 emails sent from each student, this meant that the examiner had 240 messages to decrypt/verify. The first email simply summarises the overall score. The second contains all of the individual question scores together with the overall score. The third is a complete transcript of the student's responses, which is useful if the student claims to have done better than the computer recorded. For processing marks, the second set of emails is most appropriate. Decrypting these 80 emails, appending them together, importing into a spreadsheet, sorting rows and then deleting all extraneous material took approximately 10 minutes.

6.4 Summary

The assessment chosen to test the use of PGP with TRIADS was very successful. Once all preparations had been made then the assessment ran smoothly without any problems. The students answers were encrypted, and signed, using the correct keys, and the whole process was done without any indication to the user. When the examiner came to reveal the answers there were no problems with the decryption/verification process.

7.0 Conclusions and Future Improvements

The main objective of the project was to incorporate the use of PGP within a TRIADS assessment, and this was successfully achieved. The process of running PGP with TRIADS required a small amount of preparation work, but once all the preparations had been made then the assessment ran without any problems.

Future improvements on this project might include:

- There is a potential security weakness with the student's keys because the passphrase used for the private key is the same as the username associated with the key.

  A possible improvement would be to incorporate some form of simple transformation algorithm, between the username and passphrase, into the *genkey* and *pgp_triads* programs. Therefore the passphrase will not be the same as the username, but the correct passphrase can still be derived from each username.

- Speeding up the decryption/verification process for the examiner. One option that we have considered could be to send the students results in PGP/MIME format, instead on inline PGP. However as a lot of email programs cannot handle PGP/MIME messages then there could be an option in the *pgp_triads* program as to whether the results will be sent as inline PGP, or PGP/MIME.
- The possibility of encrypting the students answers to more than one examiner.
- Update the *genkey* program from a Windows console application to a Windows GUI application.
- The random number generation method used in the *genkey* and *pgp_triads* programs need improving.

A.1 PGP (Pretty Good Privacy)

PGP is an email and file encryption program originally developed by Phil Zimmerman in 1991.

PGP uses a technique called public-key cryptography, whereby each user generates their own public and private key, known as a key pair. The two keys are related, but they are not deducible from each other. If the public key is used to encrypt a message then the private key is used to decrypt it, and vice-versa.

The public key can be made available to as many people as possible; it is freely distributable. For users to be able to share secure email they must have access to other users' public keys.

The private key however is exactly what the names suggests, private, and it should be kept secret by the user. As added protection the private key is encrypted using a passphrase. Whenever the private key is required the corresponding passphrase must be entered first. Therefore if someone else were to obtain a user's private key it would be useless to them without the correct passphrase. For more information see The International PGP Homepage at http://www.pgpi.org.

A.1.1 Public and Private Keyrings

PGP creates two keyring files on the users computer for storage of PGP keys. The public keyring (pubring.pkr) contains the users own public key, and a collection of other people's public keys. In order to send an encrypted message, or verify a signature, the user must have the other users' public key stored in their public keyring.

The private keyring (secring.skr) contains the users own private key. The private key is encrypted with a passphrase that has to be entered every time the user wishes to sign a message, or decrypt a message that they have been sent.

A.1.2 Encrypting a Message with PGP

Encrypting a message provides the following benefit:

- Prevention of unauthorised reading of an email message

For example, Alice wants to send an encrypted message to Bob. To accomplish this the following steps are carried out:

a) Alice types a message for Bob

b) PGP encrypts the message using Bob’s public key

c) The encrypted message is then sent to Bob

d) Upon arrival PGP decrypts the message by using Bob’s private key

e) As Bob is the only person who knows his private key, then he is the only person who can decrypt the message

Fig. A.1 shows an example of how keys are used to encrypt/decrypt an email message.

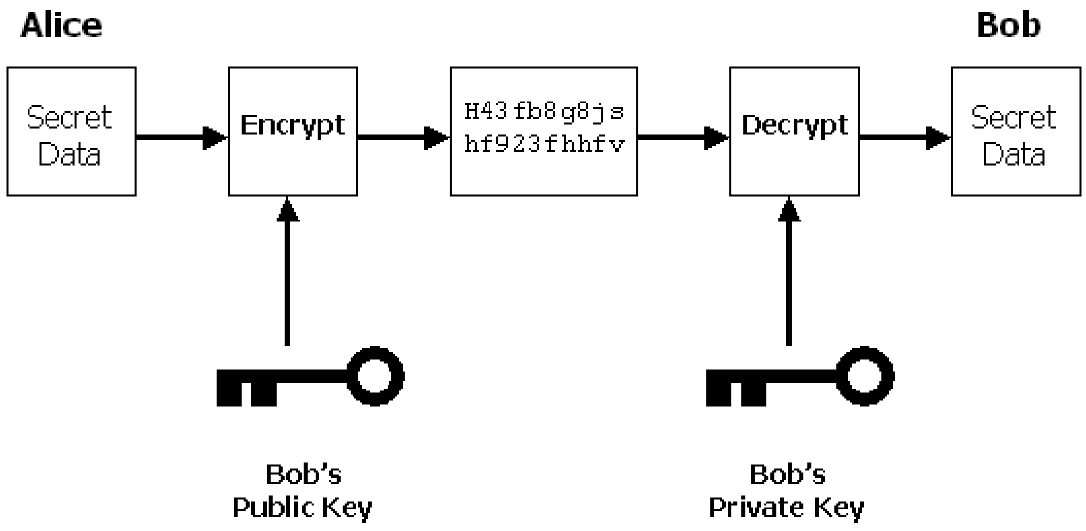


Fig. A.1 – Encrypting a Message

A.1.3 <u>Signing a Message with PGP</u>

Signing a message provides the following benefits:

- Confirmation of the origin of a message
- Proof that the message has not been altered

For example, Alice wants to send a signed message to Bob. To accomplish this the following steps are carried out:

a) Alice writes a message for Bob

b) PGP runs the message through a mathematical function

c) A value is generated by the function

d) PGP encrypts this value using Alice’s private key, this creates a digital signature

e) The digital signature is put at the end of the message, and it is then sent to Bob

f) Upon arrival PGP verifies the digital signature by using Alice’s public key. This reveals the original value generated by the mathematical function

g) As Alice is the only person who knows her private key then only she could have created the digital signature

h) PGP again runs the message through a mathematical function

i) If the value generated at Bob’s end is different to the one sent in the digital signature by Alice then the contents of the message has been altered

Fig. A.2 shows an example of how keys are used to sign/verify an email message.

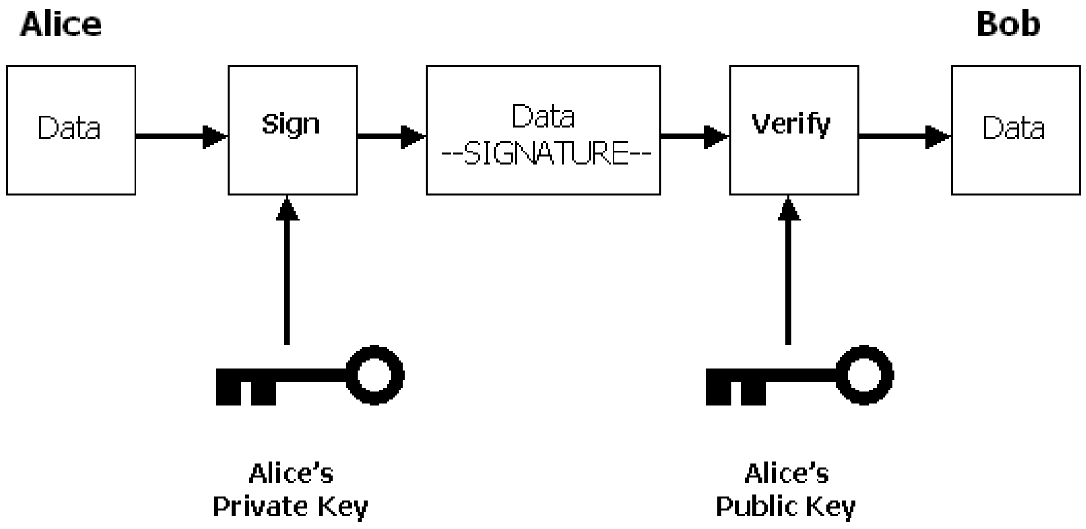


Fig. A.2 – Signing a Message

A.1.4 <u>Signing & Encrypting a Message with PGP</u>

Signing and encrypting of a message both offer different advantages. However combining the two methods will provide the benefits of both techniques. When using both methods the message is signed first, and then it is encrypted. Once the email is received it will be decrypted first, and then the signature will be verified.

Appendix B

```
// Output of the pgp -kg command
// Answers to the questions have been underlined

Choose the public key algorithm to use with your new key
1) DSS/DH (a.k.a. DSA/ElGamal) (default)
2) RSA
Choose 1 or 2: 1

Choose the type of key you want to generate
1)  Generate a new signing key (default)
2) Generate an encryption key for an existing signing key
Choose 1 or 2: 1

Pick your DSS "master key" size:
1) 1024 bits - Maximum size (Recommended)
Choose 1 or enter desired number of bits: 1

Generating a 1024-bit DSS key

You need a user ID for your public key. The desired form for this user ID is your
name, followed by your E-mail address enclosed in <angle brackets>, if you have an E-
mail address.
For example: John Q. Smith <jqsmith@nai.com>
Enter a user ID for your public key: scese00

Enter the validity period of your signing key in days from 0 - 10950
0 is forever (the default is 0): 30

You need a pass phrase to protect your DSS secret key. Your pass phrase can be any
sentence or phrase and may have many words, spaces, punctuation, or any other
printable characters.
Enter pass phrase: scese00
Enter same pass phrase again: scese00
PGP will generate a signing key. Do you also require an encryption key? (Y/n) Y

Pick your DH key size:
1) 1024 bits - High commercial grade, secure for many years
2) 2048 bits - "Military" grade, secure for foreseeable future
3) 3072 bits - Archival grade, slow, highest security
Choose 1, 2, 3, or enter desired number of bits: 2

Enter the validity period of your encryption key in days from 0 - 30
0 is forever (the default is 0): 30


Note that key generation is a lengthy process.

PGP needs to generate some random number data. This is done by measuring the time
intervals between your keystrokes. Please enter some random text on your keyboard
until the indicator reaches 100%.
Press ^D to cancel

Enough, thank you
****.........................*****.........

Make this the default signing key? (Y/n) n

Key generation completed.
```

Appendix C

```
/*
 Author          Simon Hatton
 Filename        genkey.c
 Date Started    16/10/2001
 Last Updated    04/01/2002
 See Also        key_gen.exp

 Description     This file is a C program that generates a file containing a
                 list of all users who need PGP key pair. The program also
                 calculates how long the PGP keys should be valid for.

                 This number, along with the file are used by the expect script
                 key_gen.exp. It is within this expect script that the PGP
                 keys are actually created.

                 This program has been written so that temporary keys can be
                 created automatically for use with TRIADS assessments.
*/

#include <stdio.h>
#include <string.h>
#include <time.h>

int main(int argc, char *argv[])
{
  char start_str[10],
       end_str[10],
       username[10],
       out_file[50],
       buffer[100];

  int start_num,
      end_num,
      counter,
      end_day,
      end_month,
      end_year;

  size_t end_len;

  time_t rawtime,
         start_sec,
         end_sec;

  struct tm * timeinfo;
  double dif;
  FILE *pgpkeys;

// Ask the user to enter the output filename and the format that the temporary
// usernames will consist of.

  printf ("\nAUTOMATIC PGP KEY GENERATION\n");
  printf ("===========================\n\n");
  printf ("Please enter the filename to store the list of usernames: ");
  scanf ("%s", out_file);
  printf ("\nPlease enter the username, start number, and end number format that will
           be used\nfor the PGP keys. Please enter any leading zero's\n\n");
  printf ("Format : <username> <start_num> <end_num>\n");
  printf ("Example: scese 0000 0100\n\n");
  printf ("Enter Details: ");
  scanf ("%s %s %s", username, start_str, end_str);

// Convert the start_num and end_num parameters into integers.
```

```
  start_num = atoi (start_str);
  end_num = atoi (end_str);

// Calculate the length of the number format for the usernames.

  end_len = strlen (end_str);

// Open the output file.

  pgpkeys = fopen (out_file, "w");

// Output each username into a file. Place the correct number of leading
// zero's in each username.

  for (counter = start_num ; counter <= end_num ; counter++)
  {
    switch (end_len)
    {
      case 1  : fprintf (pgpkeys, "%s%01d\n", username, counter);
                break;
      case 2  : fprintf (pgpkeys, "%s%02d\n", username, counter);
                break;
      case 3  : fprintf (pgpkeys, "%s%03d\n", username, counter);
                break;
      case 4  : fprintf (pgpkeys, "%s%04d\n", username, counter);
                break;
      case 5  : fprintf (pgpkeys, "%s%05d\n", username, counter);
                break;
      default : printf ("\nLine 92. The number you have input has more than 5
                         digits.\nYou need to add a new switch statement that can
                         handle the number of digits\nfor your required
                         username.\n");
                fclose (pgpkeys);
                exit (1);
                break;
    }
  }

  fclose (pgpkeys);

  printf ("\nThe file \"%s\" has been created. This contains all usernames
           that\nrequire a PGP keypair.\n\n", out_file);

// Get the current date and time.

  time (&rawtime);

// Set the time to 00:00, for the current day.

  timeinfo = localtime (&rawtime);
  timeinfo->tm_hour = 0;
  timeinfo->tm_min = 0;
  timeinfo->tm_sec = 0;

// Calculate the number of seconds between January 1st 1970 and 00:00 on
// todays date.

  start_sec = mktime (timeinfo);

// Ask the user to enter the date for the PGP keys to expire on.

  printf ("Please enter the date when the PGP keys should Expire\n");
  printf ("Format : dd mm yyyy\n");
  printf ("Example: 12 03 2002\n\n");
  printf ("Enter Date: ");
  scanf ("%d %d %d", &end_day, &end_month, &end_year);

// Calculate the number of seconds between January 1st 1970 and 00:00 on
// the date that the PGP will expire.

  time (&rawtime);

  timeinfo = localtime (&rawtime);
  timeinfo->tm_mday = end_day;
  timeinfo->tm_mon = end_month - 1;
  timeinfo->tm_year = end_year - 1900;
  timeinfo->tm_hour = 0;
  timeinfo->tm_min = 0;
  timeinfo->tm_sec = 0;

  end_sec = mktime (timeinfo);

// Calculate the time difference (in seconds) between todays date, and the date
// when the keys will expire.

  dif = difftime (end_sec, start_sec);

// Convert the time difference (in seconds) into days.

  dif = (((dif / 60) / 60) / 24);

  counter = dif;

// Launch the expect script key_gen.exp, with the output filename and the
// number of days as input parameters.

  sprintf (buffer, "key_gen.exp %s %d", out_file, counter);

  system (buffer);

  return 0;
}
```

Appendix D

```
#!/usr/local/bin/expect --

# Author          Simon Hatton
# Filename        key_gen.exp
# Date Started    22/10/2001
# Last Updated    26/11/2001
# See Also        genkey.c
# Description     This file is an expect program that automatically generates
#                 PGP key pairs. The program reads from a file of usernames.
#                 For each username in the file a PGP key pair is generated.
#
#                 The expect script takes two input parameters which are the
#                 file containing the list of the usernames, and the number of days
#                 that the key is valid. These parameters are generated in the program
#                 genkey.c
#
#                 This program has been written so that temporary keys can be
#                 created automatically for use with TRIADS assessments.


# Set the send_slow command to avoid any problems with input being sent before
# the program is ready for it.

set send_slow {1 2}

set arg1 [lindex $argv 0]
set arg2 [lindex $argv 1]

# Open the input file for reading.

set keyfile [open $arg1 r]

# Extract a username from the file, store it in the variable user. If there
# are no usernames left to extract then exit the loop.

while {[gets $keyfile user] != -1} {

# Check variable initially set to 0.

  set check 0

# Execute the pgp command line with the key generation (-kg) parameter.

  spawn pgp -kg

# Select the DSS/DH algorithm to use with the new key.

  expect "Choose 1 or 2:"
  send -s "1\r"

# Select to generate a new signing key.

  expect "Choose 1 or 2:"
  send -s "1\r"

# Pick the DSS 'master key' size of 1024 bits.

  expect "Choose 1 or enter desired number of bits:"
  send -s "1\r"

# Enter a user ID for the public key. This will be the value of the variable
# user that was extracted from the file username.

  expect "Enter a user ID for your public key:"
  send -s "$user\r"

# Enter the validity period of the signing key in days.

  expect "0 is forever (the default is 0):"
  send -s "$arg2\r"

# Enter a pass phrase. This will be the value of the variable user.

  expect "Enter pass phrase:"
  send -s "$user\r"

  expect "Enter same pass phrase again:"
  send -s "$user\r"

# Choose to create an encryption key as well as a signing key.

  expect "encryption key? (Y/n)"
  send -s "Y\r"

# Select the DH key size of 2048 bits.

  expect "Choose 1, 2, 3, or enter desired number of bits:"
  send -s "2\r"

# Enter the validity period of the encryption key in days.

  expect "0 is forever (the default is 0):"
  send -s "$arg2\r"

  set random_number [expr int(rand()*10)]
  expect "required data"
  send -s "$random_number\r"

# Keep entering random numbers every 1 or 2 seconds until the indicator
# reaches 100%.

  while {$check == 0} {

# Enter a random number between 0 and 9 inclusive.

    set random_number [expr int(rand()*10)]
    expect "required data" {send "$random_number\r"}

# Set the timeout to 1 or 2 seconds.

    set random_time [expr int(rand()*2) + 1]
    set timeout $random_time

# If enough random numbers have been entered, then set the check variable to
# 1 and exit the loop.

    expect "Enough, thank you" {set check 1}
  }

# Disable the timeout.

  set timeout -1

# Don't make the new key the default signing key.

  expect "Make this the default signing key? (Y/n)"
```

```
# Reset the timeout back to the default of 10 seconds.

  set timeout 10

  send -s "n\r"

  expect "Key generation completed"

  wait
}

# Close the input file.

close $keyfile
```

Appendix E

```
/*
Author         Simon Hatton
Date Started   08/01/2002
Last Updated   13/05/2002
Description    This program automatically generates PGP key pairs. This program
               was written to be used with a TRIADS assessment so that each
               individual PGP key pair didn't have to be created manually.

               The user types in the usernames that they wish to create PGP
               keypairs for, and the program automatically creates a key pair
               for each one.

               For a full explanation of the commands used in this program please
               refer to the PGP SDK Reference Guide, which can be found at:

               http://www.pgpi.org/products/sdk/c++/pgpsdk/

               NOTE: The following Random Number Generation files and code used in
               this program were obtained from http://www.agner.org/random/

               randomc.h
               mersenne.cpp
*/


#include <iostream>
#include "stdafx.h"
#include "genkey.h"
#include "headers\pgpKeys.h"
#include "headers\pgpErrors.h"
#include "headers\pgpRandomPool.h"
#include "headers\pgpUtilities.h"
#include "headers\pgpEncode.h"

// RNG (Random Number Generation) files obtained from http://www.agner.org/random/

#include "headers\randomc.h"    // define classes for RNG
#include "headers\mersenne.cpp" // Members of class TRandomMersenne

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#define PUBLIC_KEY_SIZE    2048
#define PRIVATE_KEY_SIZE   1024
#define PRIVATE_ALGORITHM  kPGPPublicKeyAlgorithm_DSA
#define PUBLIC_ALGORITHM   kPGPPublicKeyAlgorithm_ElGamal

/////////////////////////////////////////////////////////////////////////////
// The one and only application object

CWinApp theApp;

using namespace std;

COleDateTimeSpan daysValid;          // Time span for how long the keys are valid
FILE *pgpkeys;                       // File containing a list of each username

PGPContextRef  context     = kInvalidPGPContextRef;  // Default context
PGPKeySetRef   theKeyRing  = kInvalidPGPKeySetRef;   // Identifier for the keyring
PGPFileSpecRef pubFileRef  = kInvalidPGPFileSpecRef; // Identifier for pubring.pkr
PGPFileSpecRef privFileRef = kInvalidPGPFileSpecRef; // Identifier for secring.skr
```

```
int Shutdown(const int);                            // Function to shutdown PGP

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
  int nRetCode = 0;
  void getPGPKeyDetails();      // Function to get details of PGP keys
  void createPGPKeys();         // Function to generate PGP keys

// initialize MFC and print and error on failure
  if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
  {
// TODO: change error code to suit your needs
    cerr << _T("Fatal Error: MFC initialization failed") << endl;
    nRetCode = 1;
  }
  else
  {
    getPGPKeyDetails();
    cout << "\n\nCreating PGP keys. Please Wait.\n";
    createPGPKeys();
    nRetCode = Shutdown(0);
    cout << "\nDONE\n";
  }
  return nRetCode;
}

// Function to get the details about the PGP keys the are to be generated

void getPGPKeyDetails()
{
  char user[256],                // User ID for each key pair e.g. scese
       startString[50],          // String of the first User ID e.g. 000
       endString[50],            // String of the final User ID e.g. 099
       endDayString[50],         // String of the day that each key pair expires
       endMonthString[50],       // String of the month that each key pair expires
       endYearString[50];        // String of the year that each key pair expires

  int startNum,                  // Number of the first User ID e.g. 000
      endNum,                    // Number of the final User ID e.g. 099
      lenEndNum,                 // Length of the endString variable
      userCount,                 // Loop counter
      counter,                   // Loop counter
      lenUserCount,              // Length of the userCount variable
      leadingZerosCounter,       // Counter for the number of leading zeros
      endDay,                    // Day that each key pair expires
      endMonth,                  // Month that each key pair expires
      endYear;                   // Year that each key pair expires

  char answer;                   // Answer to any (Y/N) questions

  CString strEndNum,             // CString of the final User ID e.g. 099
          strUserCount,          // CString of the UserCount loop counter
          strLeadingZeros;       // CString containing any leading zeros

  COleDateTime today,            // The current date and time
               endOfToday,       // 23:59:59 on todays date
               endOfExpiryDate;  // 23:59:59 on the date when the keys expire

// Open an output file for the list if usernames to be written to

  pgpkeys = fopen("TEMPKEYS.TXT", "w");

  cout << "\nAUTOMATIC PGP KEY GENERATION for Windows 2000\n";
  cout << "=============================================";

// Ask the user to enter the format that the temporary usernames will consist of

  do
  {
    answer = 0;
    cout << "\n\nPlease enter the username, start number, and end number format
    cout << "that\nwill be used for the PGP keys.";
    cout << "Please enter eny leading zeros\n\n";
    cout << "Format : <username> <start_num> <end_num>\n";
    cout << "Example: scese 0000 0099\n\n";
    cout << "Enter Details: ";
    cin >> user >> startString >> endString;

// Convert the startString, and endString, variables into integers

    startNum = atoi (startString);
    endNum = atoi (endString);

// Check that startString variable is a valid integer

    for (counter = 0 ; counter < strlen (startString) ; counter++)
    {
      if ((!isdigit (startString[counter])))
        answer = 'N';
    }

    if (answer == 'N')
      cout << "\nERROR: The <start_num> parameter MUST be a valid integer";

// Check that the endString variable is a valid integer

    else
    {
      for (counter = 0 ; counter < strlen (endString) ; counter++)
      {
        if ((!isdigit (endString[counter])))
          answer = 'N';
      }

      if (answer == 'N')
        cout << "\nERROR: The <end_num> parameter MUST be a valid integer";
    }

    if (answer != 'N')
    {

// If the start number isn't the same length as the end number then re-enter the data

      if (strlen (startString) != strlen (endString))
      {
        cout << "\nERROR: The <start_num> and <end_num> parameters MUST be the same";
        cout << " length";
        answer = 'N';
      }

// If the start number is greater than the end number then re-enter the data

      else if (startNum > endNum)
      {
        cout << "\nERROR: The <start_num> parameter MUST be less than the <end_num>";
        cout << " parameter";
        answer = 'N';
      }

// Else ask the user to confirm that the username details are correct

      else
      {
        cout << "\nYou have chosen to create a PGP key for each of the following";
        cout << "usernames:\n\n";
        cout << user << startString << " to " << user << endString;
        cout << "\n\nIs this correct? (Y/N): ";

        cin >> answer;
        cin.ignore (80, '\n');
        answer = toupper(answer);
      }
    }
  } while (answer != 'Y');

// Calculate the length of the endString variable

  strEndNum.Format("%s", endString);
  lenEndNum = strEndNum.GetLength();

// Output each username into the temp file. Place the correct number of leading zeros
// in each username

  counter = 0;

  for (userCount = startNum ; userCount <= endNum ; userCount++)
  {
    strLeadingZeros = "";
    strUserCount.Format("%d", userCount);
    lenUserCount = strUserCount.GetLength();
    leadingZerosCounter = lenEndNum - lenUserCount;

    while (leadingZerosCounter > 0)
    {
      strLeadingZeros += '0';
      leadingZerosCounter--;
    }

    fprintf (pgpkeys, "%s%s%d\n", user, strLeadingZeros, userCount);
  }

  fclose(pgpkeys);

// Ask the user to enter the date when each PGP key pair will expire

  do
  {
    cout << "\n\nPlease enter the date when the PGP keys should Expire.\n";
    cout << "Format : DD MM YYYY\n";
    cout << "Example: 01 01 2003\n\n";
    cout << "Enter Date: ";
    cin >> endDayString >> endMonthString >> endYearString;

// Check that the date is in the format DD MM YYYY

    if ((strlen (endDayString) != 2) || (strlen (endMonthString) != 2) ||
        (strlen (endYearString) != 4))
    {
      cout << "\nERROR: Please enter the date in the format DD MM YYYY";
      answer = 'N';
    }

// Check that the Day value is a valid integer

    else if ((!isdigit (endDayString[0])) || (!isdigit (endDayString[1])))
    {
      cout << "\nERROR: Invalid Day value entered";
      answer = 'N';
    }

// Check that the Month value is a valid integer

    else if ((!isdigit (endMonthString[0])) || (!isdigit (endMonthString[1])))
    {
      cout << "\nERROR: Invalid Month value entered";
      answer = 'N';
    }

// Check that the Year value is a valid integer

    else if ((!isdigit (endYearString[0])) || (!isdigit (endYearString[1])) ||
             (!isdigit (endYearString[2])) || (!isdigit (endYearString[3])))
    {
      cout << "\nERROR: Invalid Year value entered";
      answer = 'N';
    }
    else
    {

// Convert the endDayString, endMonthString, and endYearString, variables into
// integers

      endDay = atoi(endDayString);
      endMonth = atoi(endMonthString);
      endYear = atoi(endYearString);

// Get the current date and time

      today = COleDateTime::GetCurrentTime();

// Set the endOfToday variable to 23:59:59 on the current date

      endOfToday.SetDateTime(today.GetYear(), today.GetMonth(), today.GetDay(), 23,
                             59, 59);

// Set the endOfExpiryDate variable to 23:59:59 on the Expiry date

      endOfExpiryDate.SetDateTime(endYear, endMonth, endDay, 23, 59, 59);

// If an invalid date has been entered then re-enter the date

      if (endOfExpiryDate.GetMonth() == -1)
      {
        cout << "\nERROR: You have entered an Invalid Date";
        answer = 'N';
      }

// If a date less than today has been entered then re-enter the date

      else if (endOfExpiryDate <= endOfToday)
      {
        cout << "\nERROR: The keys must expire AFTER today";
        answer = 'N';
      }

// Else ask the user to confirm that the date is correct

      else
      {
        printf ("\nThe PGP keys will Expire on the %02d/%02d/%02d. ");
        printf ("Is this correct? (y/N): ", endDay, endMonth, endYear);
        cin >> answer;
        cin.ignore (80, '\n');
        answer = toupper(answer);
      }
    }
  } while (answer != 'Y');

// Calculate the difference between the current date and the expiry date

  daysValid = endOfExpiryDate - endOfToday;
}

// Function to create each PGP key

void createPGPKeys()
{
  char const *pubKeyRing  = "pubring.pkr";   // Public keyring file
  char const *privKeyRing = "secring.skr";   // Private keyring file

  CFileFind finder;
  BOOL fileAvailable;

  char username[100];                        // User ID for each key pair

  long int seed = time(0);                   // Random seed
  TRandomMersenne randNumGenerator(seed);    // Instance of random number generator
  long int randomInteger;                    // Random integer number

  PGPError     err    = kPGPError_NoErr;     // Resultant errors
  PGPKeyRef    theKey = kPGPInvalidRef;      // Key to create
  PGPSubKeyRef subKey = kPGPInvalidRef;      // Subkey to create
  PGPUInt32    entropyNeeded;                // Random data required to create key

  void DisplayPGPError(PGPError *, const char *);  // Function to display errors

// Initialise the PGP SDK libraries

  err = PGPsdkInit();
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Initialising the PGP SDK");
    Shutdown (1);
  }

// Create a new PGP context. Most PGP functions require a valid PGPContext

  err = PGPNewContext (kPGPsdkAPIVersion, &context);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the PGP Context");
    Shutdown (1);
  }

// Create file descriptors for the public and private keyring files
```

```
  err = PGPNewFileSpecFromFullPath (context, pubKeyRing, &pubFileRef);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the \"pubring.pkr\" File Descriptor");
      Shutdown (1);
  }

  err = PGPNewFileSpecFromFullPath (context, privKeyRing, &privFileRef);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the \"secring.skr\" File Descriptor");
      Shutdown (1);
  }

// Open the public and private keyring files. If this program is run in a directory
// where there are already existing pubring.pkr and secring.skr files, then these
// files will be updated. Otherwise new pubring.pkr and secring.skr files
// will be created.

  err = PGPOpenKeyRingPair (context, kPGPKeyRingOpenFlags_Create +
                            kPGPKeyRingOpenFlags_Mutable, pubFileRef,
                            privFileRef, &theKeyRing);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Opening the key ring");
    Shutdown (1);
  }

// Open the file containing the list of all usernames that need a PGP key pair

  pgpkeys = fopen("TEMPKEYS.TXT", "r");

// Create a PGP key pair for each username listed in the file

  fscanf (pgpkeys, "%s", username);

  while (!feof(pgpkeys))
  {

// Calculate the amount of random data required to generate a PGP key pair

    entropyNeeded = PGPGetKeyEntropyNeeded (
                     context,
                     PGPOKeyGenParams (context, PRIVATE_ALGORITHM, PRIVATE_KEY_SIZE),
                     PGPOKeyGenFast   (context, TRUE),
                     PGPOLastOption   (context));

    entropyNeeded = entropyNeeded +
                       PGPGetKeyEntropyNeeded (
                       context,
                       PGPOKeyGenParams (context, PUBLIC_ALGORITHM, PUBLIC_KEY_SIZE),
                       PGPOKeyGenFast   (context, TRUE),
                       PGPOLastOption   (context));

// Make sure the Random Pool contains enough data. If it does not then increase
// the random pool until it contains the required amount of data. The random pool
// is increased by adding a random number between 0 and 999999.

    if (PGPGlobalRandomPoolGetEntropy() < entropyNeeded)
    {
      do
      {
        randomInteger = randNumGenerator.IRandom(0, 999999);
        PGPGlobalRandomPoolAddKeystroke (randomInteger);
      } while (PGPGlobalRandomPoolGetEntropy() < entropyNeeded);
    }

// Create the Signing Key part

    err = PGPGenerateKey (
           context,
           &theKey,
           PGPOKeySetRef     (context, theKeyRing),
           PGPOKeyGenParams (context, PRIVATE_ALGORITHM, PRIVATE_KEY_SIZE),
           PGPOKeyGenName    (context, username, strlen(username)),
           PGPOPassphrase    (context, username),
           PGPOExpiration    (context, (int)daysValid.GetTotalDays()),
           PGPOKeyGenFast    (context, TRUE),
           PGPOLastOption    (context));

    if (IsPGPError (err))
    {
      DisplayPGPError (&err, "Creating the Signing Key");
      Shutdown (1);
    }

// Create the encryption key part

    err = PGPGenerateSubKey (
           context,
           &subKey,
           PGPOKeyGenMasterKey (context, theKey),
           PGPOKeyGenParams    (context, PUBLIC_ALGORITHM, PUBLIC_KEY_SIZE),
           PGPOPassphrase      (context, username),
           PGPOExpiration      (context, (int)daysValid.GetTotalDays()),
           PGPOKeyGenFast      (context, TRUE),
           PGPOLastOption      (context));

    if (IsPGPError (err))
    {
      DisplayPGPError (&err, "Creating the Encryption Key");
      Shutdown (1);
    }

// Commit changes to the keyring files

    PGPCommitKeyRingChanges(theKeyRing);

// Delete all backup keyring files that have been created

    fileAvailable = finder.FindFile("*ring-bak-*.*kr");

    while (fileAvailable)
    {
      fileAvailable = finder.FindNextFile();
      remove (finder.GetFileName());
    }

    cout << "\nPGP keypair created for " << username << ".\n";

// Read the next username from the temp file

    fscanf (pgpkeys, "%s", username);
  }

  fclose (pgpkeys);
```

```
// Reload the public and private keyrings

  PGPReloadKeyRings (theKeyRing);
}

// Function to display a PGP error message

void DisplayPGPError(PGPError *e, const char *errMesg)
{
  char errorString[256];

  PGPGetErrorString(*e, sizeof(errorString), errorString);
  cout << "PGP ERROR: " << errorString << "\n" << errMesg << endl;
}

// Function to shutdown the PGP SDK

int Shutdown (const int error)
{

// De-allocate all assigned key sets, file descriptions, etc.

  if (PGPFileSpecRefIsValid (pubFileRef))
    PGPFreeFileSpec (pubFileRef);

  if (PGPFileSpecRefIsValid (privFileRef))
    PGPFreeFileSpec (privFileRef);

  if (PGPKeySetRefIsValid (theKeyRing))
    PGPFreeKeySet (theKeyRing);

// Release the PGP context

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

// Shutdown the PGP SDK library

  PGPsdkCleanup();

// Delete the temporary file containing the list of all of the usernames

  remove ("TEMPKEYS.TXT");

// If a PGP error was detected then exit the program here

  if (error == 1)
    exit(1);

  return 0;
}
```

Appendix F

```
/*
Author         Simon Hatton
Program        pgp_triads.exe
Description    This program is used in conjunction with a TRIADS assessment. The code
               takes a file containing the answers to an assessment, and then creates
               a new signed and encrypted file of the answers. The program is called
               from within an Authorware application.
*/

#include <iostream>

using namespace std;

int main()
{

// Set the PGP timestamp variable. Setting this will remove the warning message, and
// the beep, that are output when PGP is run

  _putenv ("TZ=GMT0BST");

// Set the PGP passphrase environment variable. The passphrase is the same as the
// currently logged in username. Setting this will remove the prompt where the user
// has to enter their passphrase

  _putenv ("PGPPASS=student");

// Run the PGP command line to sign a file called answers.txt, using the private key
// of the currently logged in username, and encrypt the file using the public key
// that belongs to the examiner

  system ("pgp -seat answers.txt examiner -u student");

  return 0;
}
```

Appendix G

```
/*
Author        Simon Hatton
Program       pgp_triads.exe
Description   This program is used in conjunction with a TRIADS assessment. The code
              takes a file containing the answers to an assessment, and then creates
              a new signed and encrypted file of the answers. The program is called
              from within an Authorware application with the following parameters:

              pgp_triads.exe <answers_file> <recipient> <sender>
*/

#include <iostream>

using namespace std;

int main (int argc, char *argv[])
{
// String variable that stores system commands to be executed

  char command[255];

// Set the PGP timestamp variable. Setting this will remove the warning message, and
// the beep, that are output when PGP is run

  _putenv ("TZ=GMT0BST");

// Set the PGP passphrase environment variable. The passphrase is the same as the
// <sender> input parameter. Setting this will remove the prompt where the user has
// to enter their passphrase

  sprintf (command, "PGPPASS=%s", argv[3]);
  _putenv (command);

// Set the PGP command line to sign and encrypt the file specified in the
// <answers_file> parameter. Sign the file using the private key belonging to the
// <sender> parameter, and encrypt the file using the public key that belongs to the
// <recipient> parameter

  sprintf (command,
           "pgp -seat %s %s -u %s", argv[1], argv[2], argv[3]);

// Execute the PGP command line

  system (command);

  return 0;
}
```

Appendix H

```
/*
Author        Simon Hatton
Program       pgp_triads.exe
Description   This program is used in conjunction with a TRIADS assessment. The code
              takes a file containing the answers to an assessment, and then creates
              a new signed and encrypted file of the answers. The program is called
              from within an Authorware application with the following parameters:

              pgp_triads.exe <current_dir> <answers_file> <recipient> <sender>
*/

#include <iostream>

using namespace std;

int main (int argc, char *argv[])
{
// String variable that stores system commands to be executed

  char command[255];

// Set the PGP timestamp variable. Setting this will remove the warning message, and
// the beep, that are output when PGP is run

  _putenv ("TZ=GMT0BST");

// Set the PGP passphrase environment variable. The passphrase is the same as the
// <sender> input parameter. Setting this will remove the prompt where the user has
// to enter their passphrase

  sprintf (command, "PGPPASS=%s", argv[4]);
  _putenv (command);

// Set the PGP command line to sign and encrypt the file specified in the
// <answers_file> parameter. Use the public and private key rings files stored in the
// <current_dir> parameter. Sign the file using the private key belonging to the
// <sender> parameter, and encrypt the file using the public key that belongs to the
// <recipient> parameter

  sprintf (command,
      "%spgp -seat +PUBRING=%s\\pubring.pkr +SECRING=%s\\secring.skr %s %s -u %s",
       argv[1], argv[1], argv[1], argv[2], argv[3], argv[4]);

/*
For example if the following parameters are used to call the program:

pgp_triads \\U507\ES\ c:\temp\answers.txt examiner student

Then the contents of the command variable will be:

"\\U507\\ES\pgp -seat +PUBRING=\\U507\ES\pubring.pkr +SECRING=\\U507\ES\secring.skr
 c:\temp\answers.txt examiner –u student"

*/

// Execute the PGP command line

  system (command);

  return 0;
}
```

Appendix I

```
-- This code sets up the variables in TRIADS that will be used to run PGP.

-- Config data for SMTP knowledge object
PGPsmtpserver := "mail1.liv.ac.uk"

-- Get the current working directory. This will be the <current_dir> parameter for
-- pgp_triads
PGPpath := FileLocation

-- The name of the file to be encrypted. This will be the <answers_file> parameter
-- for pgp_triads.
TRIADSfile := "answers.txt"

-- Get the username currently logged onto the PC from the environment variable
-- "USERNAME". This will be the <sender> parameter for pgp_triads. This will also be
-- the passphrase used to access the students private key.
PGPsender := baEnvironment ("USERNAME")

-- Use the following email address to identify the examiners public key. This will be
-- the <recipient> parameter for pgp_triads.
PGPrecipient := "s.c.hatton@liverpool.ac.uk"

-- Subject headers for the 3 encrypted/signed emails that are sent to the examiner.
PGPsubjectheader1 := "ESGY105 trial test results 1 " ^ EngUserFileName
PGPsubjectheader2 := "ESGY105 trial test results 2 " ^ EngUserFileName
PGPsubjectheader3 := "ESGY105 trial test results 3 " ^ EngUserFileName

-- Store the current "USERPROFILE" environment variable in the variable userdir.
-- Then set the "USERPROFILE" environment variable to be "C:\temp".
-- This is because whenever the PGP command line is executed it will create a PGP
-- sub-directory within the directory specified in the "USERPROFILE" directory.
-- This way we know that the PGP sub-directory will always be created in "C:\temp".

PGPTempPath := "C:\\temp"
userdir := baEnvironment ("USERPROFILE")
del_result := baSetEnvironment ("USERPROFILE", PGPTempPath)

-- Path for temp files used by PGP
PGPTextFilePath := "C:\\temp\\"

-- Name of output file to be created by PGP.
PGPencrypted := TRIADSfile ^ ".asc"

-- Text to be encrypted
PGPtext1 := EngRecordForFile[1]
PGPtext2 := EngRecordForFile[2]
PGPtext3 := EngRecordForFile[3]
```

```
-- This code copies the data entered by the student into a temporary file called
-- "C:\temp\answers.txt". This file is then encrypted and signed by the pgp_triads
-- program to produce a ciphertext file called "C:\temp\answers.txt.asc". The
-- contents of this new file is then copied into an Authorware variable.

-- Copy the text entered by the user into the file "C:\temp\answers.txt".
WriteExtFile (PGPTextFilePath ^ TRIADSfile, PGPtext1)

-- Execute the program pgp_triads.exe. This program sets up, and runs, the PGP
-- command line. Afterwards the ciphertext file "answers.txt.asc" is created.
-- pgp_triads.exe <current_dir> <answers_file> <recipient> <sender>

baRunProgram ("\"" ^ PGPpath ^ "pgp_triads.exe\" \"" ^ PGPpath ^ "\\\" \"" ^
              PGPTextFilePath ^ TRIADSfile ^ "\" " ^ PGPrecipient ^ " " ^ PGPsender,
              "hidden", TRUE)

-- Copy the contents of "C:\temp\answers.txt.asc" into the variable PGPCipherText.
PGPCipherText := ReadExtFile (PGPTextFilePath ^ PGPencrypted)
```

```
-- This code deletes all files created by the pgp_triads program.

-- Delete the files "answers.txt", and "answers.txt.asc"

DeleteFile (PGPTextFilePath ^ TRIADSfile)
DeleteFile (PGPTextFilePath ^ PGPencrypted)

-- Delete the directory "C:\temp\Appication Data\PGP". This temporary directory is
-- created each time PGP is executed.

del_result := baDeleteXFiles (PGPTempPath ^ "\\Application Data\\PGP", "*.*")
del_result := baDeleteFolder (PGPTempPath ^ "\\Application Data\\PGP")
del_result := baDeleteFolder (PGPTempPath ^ "\\Application Data")

-- Reset the USERPROFILE environment variable to what is was originally before PGP
-- was run.

del_result := baSetEnvironment ("USERPROFILE", userdir)
```

Appendix L

```
/*
Author          Simon Hatton
Date Started    11/12/2001
Last Updated    23/03/2002
Description     This program is used in conjunction with a TRIADS assessment. The
                code takes a file containing the answers to an assessment, and then
                creates a new signed and encrypted file of the answers. The program
                is called from within an Authorware application with the following
                parameters:

                pgp_triads.exe <current_dir> <answers_file> <recipient> <sender>

                For a full explanation of the commands used in this program please
                refer to the PGP SDK Reference Guide, which can be found at:

                http://www.pgpi.org/products/sdk/c++/pgpsdk/

                NOTE: The following Random Number Generation files and code used in
                this program were obtained from http://www.agner.org/random/

                randomc.h
                mersenne.cpp
*/

#include <iostream>
#include "headers\pgpKeys.h"
#include "headers\pgpErrors.h"
#include "headers\pgpRandomPool.h"
#include "headers\pgpUtilities.h"
#include "headers\pgpEncode.h"

// RNG (Random Number Generation) files obtained from http://www.agner.org/random/

#include "headers\randomc.h"     // define classes for RNG
#include "headers\mersenne.cpp"  // Members of class TRandomMersenne

using namespace std;

PGPContextRef  context          = kInvalidPGPContextRef;  // Default context
PGPKeySetRef   ourKeyRing       = kInvalidPGPKeySetRef;   // KeySet for key ring
PGPFilterRef   filter           = kInvalidPGPFilterRef;   // KeySet search filter
PGPKeySetRef   foundPubKey      = kInvalidPGPKeySetRef;   // KeySet for public keys
PGPKeySetRef   foundPrivKey     = kInvalidPGPKeySetRef;   // KeySet for private keys
PGPFileSpecRef inFileRef        = kInvalidPGPFileSpecRef; // File ref for input file
PGPFileSpecRef outFileRef       = kInvalidPGPFileSpecRef; // File ref for output file
PGPFileSpecRef pubFileRef       = kInvalidPGPFileSpecRef; // File ref for public keys
PGPFileSpecRef privFileRef      = kInvalidPGPFileSpecRef; // File ref for priv keys
PGPKeyIterRef  keyListIterator = kInvalidPGPKeyIterRef;  // KeySet iterator

// Function to display a PGP error message.

void DisplayPGPError (PGPError *e, const char *errMesg)
{
  char errorString[256];

  PGPGetErrorString (*e, sizeof (errorString), errorString);
  printf ("PGP ERROR: %s\n%s\n", errorString, errMesg);
}

// Function to shutdown PGP.

int Shutdown (const int error)
{
```

```
// De-allocate all assigned key sets, file descriptions, filters etc.

  if (PGPKeyIterRefIsValid (keyListIterator))
  {
    PGPFreeKeyIter (keyListIterator);
  }

  if (PGPFileSpecRefIsValid (pubFileRef))
  {
    PGPFreeFileSpec (pubFileRef);
  }

  if (PGPFileSpecRefIsValid (privFileRef))
  {
    PGPFreeFileSpec (privFileRef);
  }

  if (PGPFileSpecRefIsValid (inFileRef))
  {
    PGPFreeFileSpec (inFileRef);
  }

  if (PGPFileSpecRefIsValid (outFileRef))
  {
    PGPFreeFileSpec (outFileRef);
  }

  if (PGPFilterRefIsValid (filter))
  {
    PGPFreeFilter (filter);
  }

  if (PGPKeySetRefIsValid (foundPubKey))
  {
    PGPFreeKeySet (foundPubKey);
  }

  if (PGPKeySetRefIsValid (foundPrivKey))
  {
    PGPFreeKeySet (foundPrivKey);
  }

  if (PGPKeySetRefIsValid (ourKeyRing))
  {
    PGPFreeKeySet (ourKeyRing);
  }

// Release the PGP Context.

  if (PGPContextRefIsValid (context))
  {
    PGPFreeContext (context);
  }

// Shutdown the PGPsdk library.

  PGPsdkCleanup();

// If a PGP error was detected then exit the program here.

  if (error == 1)
  {
    exit (1);
  }
  return 0;
}

// Entry point for the program.

int main (int argc, char *argv[])
{
// Initialise variables

  char const *currentDir    = argv[1];    // Current directory
  char const *inFileName    = argv[2];    // Plaintext file (C:\temp\answers.txt)
  char const *receiver      = argv[3];    // Name of public key used to encrypt
  char const *sender        = argv[4];    // Name of private key used to sign
  char const *keyPassphrase = argv[4];    // Passphrase for the private key

  char pubKeyRing[500];                   // Location of public key ring
  char privKeyRing[500];                  // Location of private key ring
  char outFileName[500];                  // Output file

  long int seed = time(0);                // Random seed
  TRandomMersenne randNumGenerator(seed); // Instance of random number generator
  long int randomInteger;                 // Random integer number

  PGPError      err           = kPGPError_NoErr;         // Resultant errors
  PGPKeyListRef foundKeysList = kInvalidPGPKeyListRef;  // KeyList search results
  PGPKeyRef     signingKey    = kInvalidPGPKeyRef;      // Signing key

  void DisplayPGPError (PGPError *, const char *);  // Function to display errors
  int  Shutdown (const int);                        // Function to shutdown PGP

// Initialise the name of the output file. This will be the same as the input file,
// but with the extension ".asc" added.
// For example if inFileName = C:\temp\answers.txt, then outFileName =
// C:\temp\answers.txt.asc

  strcpy (outFileName, inFileName);
  strcat (outFileName, ".asc");

// Initialise the pubKeyRing, and privKeyRing string variables, to point to the
// public and private key ring files.

  strcpy (pubKeyRing, currentDir);
  strcat (pubKeyRing, "pubring.pkr");

  strcpy (privKeyRing, currentDir);
  strcat (privKeyRing, "secring.skr");

// Initialise the PGPsdk libraries.

  err = PGPsdkInit();
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Initialising the PGP SDK");
    Shutdown (1);
  }

// Create a new PGPContext. Most PGP functions require a valid PGPContext.

  err = PGPNewContext (kPGPsdkAPIVersion, &context);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the PGP context");
    Shutdown (1);
  }

// Make sure the Random Pool contains enough data. If it does not then increase
// the random pool until it contains the required amount of data. The random pool
// is increased by adding a random number between 0 and 999999.

  if (!PGPGlobalRandomPoolHasMinimumEntropy())
  {
    PGPUInt32 minEntropy;
    minEntropy = PGPGlobalRandomPoolGetMinimumEntropy();
    do
    {
      randomInteger = randNumGenerator.IRandom(0, 999999);
      PGPGlobalRandomPoolAddKeystroke (randomInteger);
    } while (PGPGlobalRandomPoolGetEntropy() < minEntropy);
  }

// Create file descriptions for the public and private keyring files.

  err = PGPNewFileSpecFromFullPath (context, pubKeyRing, &pubFileRef);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the \"pubring.pkr\" File Descriptor");
    Shutdown (1);
  }

  err = PGPNewFileSpecFromFullPath (context, privKeyRing, &privFileRef);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the \"secring.skr\" File Descriptor");
    Shutdown (1);
  }

// Create file descriptions for the "answers.txt" input file, and for the
// "answers.txt.asc" output file.

  err = PGPNewFileSpecFromFullPath (context, inFileName, &inFileRef);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the \"answers.txt\" File Descriptor");
    Shutdown (1);
  }

  err = PGPNewFileSpecFromFullPath (context, outFileName, &outFileRef);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the \"answers.txt.asc\" File Descriptor");
    Shutdown (1);
  }

// Open the public and private Key Ring files.

  err = PGPOpenKeyRingPair (context, kPGPKeyRingOpenFlags_None,
                            pubFileRef, privFileRef, &ourKeyRing);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Opening the key ring");
    Shutdown (1);
  }
```

```
// Create a filter to look for the key pair of the message sender.

  err = PGPNewUserIDStringFilter (context, sender, kPGPMatchSubString, &filter);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the Private Key filter");
    Shutdown (1);
  }

// Search for the message senders key pair.

  err = PGPFilterKeySet (ourKeyRing, filter, &foundPrivKey);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Searching for the Private Key");
    Shutdown (1);
  }

// Use the first matching key pair.

  err = PGPOrderKeySet (foundPrivKey, kPGPAnyOrdering, &foundKeysList);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Getting the Private Key (1)");
    Shutdown (1);
  }

  err = PGPNewKeyIter (foundKeysList, &keyListIterator);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Getting the Private Key (2)");
    Shutdown (1);
  }

  err = PGPKeyIterNext (keyListIterator, &signingKey);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Getting the Private Key (3)");
    Shutdown (1);
  }

// Create a filter to look for the key pair of the message recipient

  err = PGPNewUserIDStringFilter (context, receiver, kPGPMatchSubString, &filter);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Creating the Public Key filter");
    Shutdown (1);
  }

// Search for the message recipients key pair.

  err = PGPFilterKeySet (ourKeyRing, filter, &foundPubKey);
  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Searching for the Public Key");
    Shutdown (1);
  }
```

```
// Sign and encrypt the file C:\temp\answers.txt, and output the results into
// the file C:\temp\answers.txt.asc.

  err = PGPEncode (context,
             PGPOArmorOutput (context, TRUE),
             PGPOSignWithKey (context, signingKey,
                 PGPOPassphrase (context, keyPassphrase),
                 PGPOLastOption (context)),
             PGPOEncryptToKeySet (context, foundPubKey),
             PGPOInputFile (context, inFileRef),
             PGPOOutputFile (context, outFileRef),

             PGPOLastOption (context));

  if (IsPGPError (err))
  {
    DisplayPGPError (&err, "Encrypting and Signing the file");
    Shutdown (1);
  }

  Shutdown (0);

  return 0;
}
```