INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

**A PRACTICAL SOLUTION TO
INTEROPERABLE SECURE E-MAIL
OVER THE INTERNET**

By

Kevin B Chafe

A thesis submitted to the Faculty of Graduate Studies
and Research in partial fulfilment of the
requirements for the degree of

Master of Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

April 2000

© Kevin Chafe, 2000

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-52387-X

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

A PRACTICAL SOLUTION TO INTEROPERABLE SECURE E-MAIL OVER THE INTERNET

A thesis submitted by Kevin B. Chafe M.Eng.,
M.Sc., B. Eng., P.Eng. in partial fulfilment of the
requirements for the degree of Master of
Engineering

**Chair, Department of Systems and Computer
Engineering**

Thesis Supervisor, Professor Mihail Devetsikiotis

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
April 2000

ABSTRACT

Secure e-mail is possible using one of a number of incompatible secure enveloping protocols and cryptographic algorithms. The goal is to provide secure e-mail without restricting the solution to a single protocol and algorithm suite. The solution must also be practical from a commercial perspective. This thesis presents a mail user agent architecture that will provide practical, secure interoperability between security enveloping protocols at the application level. A working prototype was developed and is described. The architecture is based on the IDUP-GSS-API; a security service API that allows MUAs to utilise different protocols in a way that also solves many of the implementation concerns associated with alternative monolithic architectures. An in-depth explanation of cryptographic principles, some history on the development of various secure enveloping protocols and a survey of the current state of the art with respect to secure APIs and infrastructures provide a background for this work.

ACKNOWLEDGMENTS

The author wishes to express his appreciation to Professor Mihail Devetsikiotis and Dr. Carlisle Adams for their assistance and encouragement in preparing for and writing this thesis. Similar gratitude is passed to Professors Tony Bailetti and John Callahan for introducing the author to the Internet. This inauguration was the result of participation in their web-based classroom in 1995: an event characterised by the usual trials and tribulations of Internet neophytes but whose personal impact was not fully appreciated until later in time.

One other individual for whom I am thankful is Dr. David Balenson of Network Associates Incorporated (NAI). The working prototype developed in this thesis would not have been feasible without his support and NAI's permission to extend this demonstration in the spirit of open and continuing co-operation between government sponsored research projects and the academic community.

Last, but most of all, I would also like to give special thanks to and acknowledge my wife, Ann, for her support and understanding throughout this entire program as well as our young daughter Breanna.

TABLE OF CONTENTS

Abstract	iii
Acknowledgments	iv
Table of Contents	v
List Of Tables.....	vii
List Of Figures	viii
Table of Acronyms.....	ix
Chapter 1 INTRODUCTION.....	1
1.1 Contributions	4
1.2 Thesis Outline.....	6
Chapter 2 PRINCIPLES OF CRYPTOGRAPHY AND ELECTRONIC MAIL.....	7
2.1 Electronic Mail	8
2.1.1 Brief History	8
2.1.2 Multipurpose Internet Mail Extensions (MIME)	18
2.2 Cryptography	23
2.2.1 Cryptographic Principles	27
2.2.2 Symmetric Ciphers	38
2.2.3 Asymmetric Ciphers.....	40
2.2.4 Hybrid Cryptosystems	46
2.3 Secure Enveloping Protocols.....	48
Secure Multiparts For MIME	50
2.4 Cryptosystem/Protocol Interoperability	51
2.4.1 Commercial Technologies and the Standards Development Cycle	52
2.4.2 De Facto Standards	55
2.4.3 Adopting A Single Industry Standard.....	56
2.4.4 Internet Engineering Task Force (IETF).....	61
Chapter 3 CURRENT STATE OF SECURE E-MAIL	64
3.1 Custom Development and/or Integration.....	68
Need For APIs.....	69
3.2 Proprietary APIs (Toolkits) For A Given Protocol	76
3.3 De Facto Secure Enveloping Protocol Standard	77
3.3.1 S/MIME.....	80
3.3.2 Open PGP	81
3.4 Trust Models and Digital Certificates	83
3.4.1 S/MIME Trust Model.....	86
3.4.2 PGP Trust Model	88
3.5 Summary of Secure E-Mail	89
Chapter 4 CURRENT STATE OF SECURE E-MAIL INTEROPERABILITY	91
4.1 Security Service and Cryptographic APIs.....	92
4.1.1 Common Data Security Architecture (CDSA).....	95

4.1.2 International Cryptography Experiment (ICE)	105
4.1.3 Internet Engineering Task Force (IETF)	115
4.2 Summary of Interoperable Secure E-Mail	123
Chapter 5 THE CRYPTOGRAPHIC INTEROPERABILITY PROBLEM	124
Problem Statement	125
Chapter 6 AN INTEROPERABLE SECURE E-MAIL PROTOTYPE	130
6.1 Architecture Model	130
6.2 Interoperable Architecture Model	133
6.3 IDUP Prototype	135
6.3.1 Overview of IDUP-S/MIME Plug-in	139
6.3.2 IDUP-S/MIME Plug-in Architecture	141
6.3.3 IDUP-S/MIME Plug-in Use	143
6.3.4 IDUP-S/MIME Plug-in Mechanism	145
6.3.5 Overview of Prototype Plug-in	145
6.3.6 Prototype Architecture	147
6.3.7 Prototype IDUP PGP Mechanism/CSP	148
6.3.8 Prototype Plug-in Use	149
6.3.9 Source Code Modifications	149
6.3.10 Summary of Prototype Plug-in	153
Chapter 7 CONCLUSIONS AND RECOMMENDATIONS	157
7.1 Conclusions	157
7.2 Summary of Contributions	159
7.3 Future Work	160
References	162
Appendices	170
Appendix A – A Complex Multipart MIME Message	171
Appendix B – The RSA Public Key Cryptosystem	173
Appendix C – Cryptographic API Criteria	185
Appendix D – Secure Enveloping Protocols	189
MIME Object Security Services (MOSS)	189
Secure MIME (S/MIME)	193
Pretty Good Privacy (PGP)	198
Message Security Protocol (MSP)	206
Appendix E – Public Key Cryptography Standards (PKCS)	222
Appendix F – Comparison of Cryptographic Toolkits	227
Appendix G – Source Code Modifications For Prototype	231
Glossary	281

LIST OF TABLES

Table.....	Page
Table 1 - Comparing POP and IMAP	15
Table 2 - Comparing Mail Access Modes	17
Table 3 - Domain and Codomain of a Simple Function	29
Table 4 - Codomain of a one-way function.....	32
Table 5 - Codomain for a trapdoor one-way function	34
Table 6 - CSP Algorithms	140
Table 7 - Plaintext converted to $\mathbb{Z}_{26}$	175
Table 8 - Encrypting plaintext into ciphertext using the RSA algorithm.....	176
Table 9 - $\mathbb{Z}_{26}$ converted back to plaintext	177

LIST OF FIGURES

Figure	Page
Figure 1 - Internet E-Mail Message (Formatted as per RFC 822).....	9
Figure 2 - Internet E-Mail.....	11
Figure 3 - X.400 Message Handling System Model	11
Figure 4 - E-Mail Message MIME Formatted	22
Figure 5 - One-Way Function	31
Figure 6 - Simple Cipher or Encryption Scheme	36
Figure 7 - Cryptosystem	38
Figure 8 - Composition of Ciphers	39
Figure 9 - Public Key Cryptosystem.....	41
Figure 10 - E-Mail Security Services - Confidentiality, Integrity and Authentication.....	46
Figure 11 - E-Mail Security Services - Evidence.....	47
Figure 12 - Cryptographic Awareness Property.....	75
Figure 13 - Strict Hierarchical Trust Model.....	87
Figure 14 - A Simplistic PGP Certificate Path.....	89
Figure 15 - The Common Data Security Architecture	96
Figure 16 - DARPA ICE Project Architecture.....	107
Figure 17 - ICE Project Recommended CAPI Suite	110
Figure 18 - DARPA ICE Project Architecture, Conceptual Components	114
Figure 19 - DARPA ICE Project Secure E-Mail Demonstration Architecture	114
Figure 20 - Current Architecture Model	131
Figure 21 - IDUP-based Architecture	133
Figure 22 - Plug-in Architecture	142
Figure 23 - Plug-in User Interface	144
Figure 24 - Modified Plug-in Architecture.....	147
Figure 25 - Signed Message Using MOSS Protocol.....	192
Figure 26 - Signed Message using S/MIME Protocol.....	197
Figure 27 - Signed Message Using PGP and Secure Multiparts (i.e. PGP/MIME)	205
Figure 28 - X.400 Message Using MSP	206
Figure 29 - Internet Message Using MSP	210

TABLE OF ACRONYMS

API	Application Programming Interface
ARPANET	Advanced Research Project Agency Network
ASN.1	Abstract Syntax Notation, version one
CAPI	Cryptographic Application Programming Interface
CAT WG	Common Authentication Technologies Working Group
CDSA	Common Data Security Architecture
CMS	Cryptographic Message Syntax
CSP	Cryptographic Service Provider
CSSM	Common Security Services Manager
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
ESS	Extended Security Services (S/MIME, version 3)
EDIINT	Electronic Data Interchange - Internet Integration
GUI	Graphical User Interface
GP	General Purpose - referring to a subset of the IDUP interface
IDEA	International Data Encryption Algorithm
IDUP-GSS-API	Independent Data Unit Protocol - Generic Security Service - Application Programming Interface
IETF	Internet Engineering Task Force
IMAP	Internet Mail Access Protocol
MD5	Message Digest, version five
MIME	Multipurpose Internet Mail Extensions
MOSS	MIME Object Security Services
PGP	Pretty Good Privacy
MISSI	Multilevel Information System Security Initiative
MS	Mail Store
MSP	Message Security Protocol
MTA	Mail Transfer Agent
MUA	Mail User Agent
NSA	National Security Agency
PKCS	Public Key Cryptography Standards
PKIX	Public Key Infrastructure (X.509)
POP	Post Office Protocol
RSA	Rivest Shamir Adleman
SE/EV	Sign, Encrypt/Evidence - referring to a subset of the IDUP interface
S/MIME	Secure MIME
SHA-1	Secure Hash Algorithm, version one
SPI	Security Provider Interface

SSAPI
TCB
TIS

Security Service Application Programming Interface
Trusted Computing Base
Trusted Information Systems (Incorporated)

Chapter 1 INTRODUCTION

Computer network communications are broadly divided into two categories: on-line (real time) and store-and-forward communications. A financial transaction at a bank's teller machine uses an on-line environment with the bank's mainframe or equivalent. A connection is established with the mainframe throughout the entire transaction; similar to a telephone call between a business and a client. Store-and-forward communications, on the other hand, do not establish a connection between source and destination. They are characterised by a series of independent connections between intermediary nodes in the communications network. The data is stored at each intermediary node then forwarded onto the next node until it reaches its destination; similar in concept to mailing a letter through Canada Post and its intermediary post offices or distribution centres. Electronic mail (e-mail) uses store-and-forward communications. The difference between these two types of communications is fundamental to the way in which security services are provided. Providing secure services directly between two entities over an existing communications channel is one thing. It is quite another to provide the same services without being able to communicate directly with the other parties and having to rely on untrusted intermediary nodes for the delivery of this information.

Secure Internet communication services are commercially available. However, they are at a point in their evolution where they are required universally across the Internet: independent of the many and diverse technologies implemented across the individual component networks. Although similar interoperability efforts have been accomplished in other areas of computer networking, secure communications and its underlying cryptographic technologies are not as flexible or tolerant, as are many other aspects of computer networking.

The basic task of secure communications is to ensure the validity (data origin and integrity) and confidentiality of data communications. Implicit in providing these services is the concept of trust. It is of little use to provide these secure services without an acceptable level of trust. For example, electronic financial transactions would not be possible if the identity of the originator was *probably* Mr. Brown. Cryptographic algorithms and protocols are used as one of the mechanisms to provide this level of trust. Cryptography enhances secure communications by minimising the risks associated with accidental or intentional misinformation. The cost of providing this trust is the relative inflexibility rooted in the cryptographic primitives of the cryptosystem employed.

This “inflexibility”, from one perspective, is a principal design goal in any cryptosystem used to support secure communications. Indeed, the role of secure communications in providing data integrity services is to detect and reject any kind of modifications to the data. Designing cryptosystems that provide this required functionality excludes any possibilities of interoperability directly between or across cryptosystems.

Regardless of the fact that none of the cryptosystems were originally designed with interoperability as an objective, their very nature prohibits achieving such an objective in favour of the confidence or trust attained. Cryptosystems utilise algorithms that restrict the interoperability of encryption and decryption transformations to specific mathematical functions operating over certain mathematical groups or fields. This is further restricted by the use of a unique cryptographic variable or key. It is precisely this lack of interoperability or the fact that it is so difficult to participate in this exchange, without the specific algorithm and cryptographic variable, that provides users of secure services with the confidence that these communications are legitimate.

Consequently, interoperability and security are often seen as mutually exclusive, opposing objectives. One is striving for an open and implicitly trusting communications environment designed to facilitate connectivity, while the other creates a very closely controlled communications environment designed to formally and explicitly enforce trust mechanisms.

The ultimate goal is to enable comprehensive and interoperable security services across the Internet. We shall see that it is possible to utilise a high-level Security Service Application Programming Interface (SSAPI) together with an API-based security infrastructure to provide the required security services while simultaneously providing interoperability - *at the user level*. This can also be described as providing transparent security protocols at the application layer.

The building blocks of this capability are *APIs*, *mechanisms* and *implementations*. APIs are interface specifications defining generic security services that must be provided. Mechanisms are also specifications but detail how the API services are to be accomplished. Mechanisms are not generic in that they specify how a given enveloping protocol performs these services. Mechanisms also specify to a lower level of detail than APIs and prescribe such things as formats and options within the given protocol. Implementations are the actual software or hardware modules used to execute these services. These APIs and mechanisms will be the vehicles required for industry to implement this capability; providing secure interoperability across the Internet in a way that is independent of communications protocols, individual network operating system characteristics as well as the actual cryptographic algorithms used to provide these secure services.

The Internet Engineering Task Force (IETF) and more specifically the Common Authentication Technologies (CAT) Secure Working Group, has recently developed the SSAPI upon which the Internet and its industry will be able to build these universal security services.

1.1 Contributions

The importance of secure Internet e-mail is well known in the industry. The public is becoming increasingly security conscious, especially in regards to communications over the

Internet. Much work is being done to resolve outstanding security issues and promote the use of secure technologies. A number of different and incompatible protocols have been proposed, defined and even standardised to provide secure e-mail. Literature abounds in the general field of cryptography in sharp contrast to what it was just 20 years ago. Surprisingly, the literature contains very little related specifically to providing secure application layer *interoperability* between these secure e-mail protocols. This thesis has tried to amend this deficiency by making the following contributions:

- a. Surveyed the literature and identified a void with respect to application layer interoperability between secure protocols
- b. Presented possible explanations for this deficiency, attributing the commercial environment in which cryptography is now being developed
- c. Provided a *detailed* background and explanation of fundamental e-mail and cryptographic principles
- d. Identified research and development efforts in the area of SSAPIs, Cryptographic APIs (CAPIs), and API-based security architectures
- e. Summarised the current state of the art with respect to secure e-mail as well as secure e-mail interoperability
- f. Advanced the state of secure interoperability by describing how this can be accomplished at the application layer using a SSAPI such as IDUP-GSS-API
- g. Proposed a practical solution to secure, interoperable e-mail for the Internet
- h. Validated this solution with a working, multi-protocol secure e-mail prototype

1.2 Thesis Outline

This thesis is organised into seven chapters. The first is this introduction, followed by background material in which e-mail concepts are explained and secure e-mail enveloping technologies are described. The third and fourth chapters explore the current state of secure e-mail and issues related to secure e-mail interoperability. Interoperable secure e-mail may be accomplished in different ways. Solutions range from deploying a single technology, which avoids the requirement for application layer *interoperability* or transparent security protocols, to deploying a generic application programming interface that will support all the secure e-mail enveloping technologies and provide true application layer interoperability across the Internet. Chapter Five summarises the secure interoperability problem addressed in this thesis and why it is worthwhile solving. The next chapter presents a model showing how the previously described technologies can be used to construct a Mail User Agent (MUA) architecture that goes beyond the current state of secure e-mail; to create an interoperable secure e-mail agent capable of providing comprehensive and transparent security protocols across the Internet. This is followed by a description of the prototype used to validate this architectural model. Finally, Chapter Seven concludes this thesis and summarises the results of this work.

Chapter 2 PRINCIPLES OF CRYPTOGRAPHY AND ELECTRONIC MAIL

This chapter presents the fundamentals of e-mail followed by an introduction to cryptography. With a basic understanding of cryptographic principles and ciphers, we explore how cryptography is used to provide security services to e-mail in general. Moving up one level of abstraction, above the cryptographic primitives or ciphers, secure enveloping protocols and how they are used to support secure communications are explored. Borrowing from the physical equivalent, these secure protocols “envelope” the e-mail content much like the physical envelope provides confidentiality for a letter. This level, the protocol level, is both the boon and the bane of secure e-mail interoperability. On one hand it is clear that the protocol level can be used to provide the required secure e-mail interoperability. On the other hand there are four primary secure enveloping protocols; none of which are compatible with respect to any other.

Technical problems with interoperability between these secure enveloping protocols are then introduced together with other non-technical factors that complicate achieving secure interoperability. These other factors remind the reader that commercial security is both a business and a technology. The commercial development environment in which these technologies are being developed is briefly characterised in an effort to provide a deeper

insight into their evolution and help explain why the industry has not yet implemented a truly interoperable secure e-mail solution. Consequently these cryptographic technologies repeat the same cycle as most other commercial technologies: beginning with a number of incompatible products and eventually looking for a way to have them interoperate.

2.1 Electronic Mail

2.1.1 Brief History

One of the original goals of the ARPANET, the grandmother of the Internet, was to enable scientists to share information. Originally using services such as FTP, these services eventually lost favour to E-mail, which became the preferred means of inter-personal communications. The backbone and core e-mail system of the Internet has existed since 1969 when ARPANET was born [1]. Originally, e-mail was simply short, text based messages and by 1977 employed several informal standards. Eventually, it became necessary to develop a single, more formal standard for e-mail. In 1982 two e-mail standards were established: Internet Standard # 10 known as the "Standard for the Format of ARPA Internet Text Messages" [2] and Internet Standard # 11, the "Simple Mail Transport Protocol" (SMTP) [3]. These standards reflect the state of e-mail technologies at that time. E-mail was viewed simply as three basic parts - the SMTP envelope and the message contents. The message content itself was further divided into two parts: the headers and the body [4]. Note the SMTP envelope is not the same envelope or concept referred to in the term secure enveloping

protocol. The SMTP envelope is used by the Message Transfer System for routing or delivering the message. Once delivered, the headers and the body (i.e. the message content) are used by the mail user agent and/or presented to human end users. The structure of the body or text of the message was arbitrary and left up to the user but great efforts were made to specify the SMTP envelope and header fields to ensure consistency for routing and processing of these messages across the APRA/Internet and subsequent delivery to the user. Figure 1 shows a simple e-mail message formatted as per Internet Standard # 10 or more commonly known as RFC 822. Note this figure illustrates the message content consisting of both headers and message body. It does not illustrate the SMTP envelope that is outside or "on top of" the message content. The SMTP envelope is not usually seen by the end user. The same message will be used throughout this chapter to illustrate formatting and differences between secure enveloping protocols.

Figure 1 - Internet E-Mail Message (Formatted as per RFC 822)

There are two basic e-mail architectures: X.400 and Internet Mail. Many of the terms and concepts used in Internet mail are the same as those used in the X.400 standard because the Internet architecture is based on the X.400 model. X.400 is the ITU nomenclature for the international standards designed to provide e-mail services. It is also known as Message Handling Systems (MHS) standards. Although this thesis deals with Internet mail, the common X.400 e-mail model is presented in parallel with the Internet architecture.

Figure 2 - Internet E-Mail is a diagram showing basic message processing and transport across the Internet [5]. Figure 3 - X.400 Message Handling System Model is the X.400 equivalent. Except for the specific reference to Internet protocols (SMTP) and message formatting (RFC822, MIME) these figures are based on the same model. E-mail processing can be roughly categorised as three processes involving the same number of mail system components. The three mail processes are sending, routing and receiving. The three components are the Mail User Agent (MUA), the Mail Transfer Agent (MTA) and the Mail Store (MS). The MUA is the mail program or e-mail software package that executes on the end users workstation. It is sometimes referred to as the mail client. The MUA does most of the work in terms of creating, editing, displaying, interacting with the end user, sending e-mail directly and accessing messages sent to the user.

Figure 2 - Internet E-Mail

Figure 3 - X.400 Message Handling System Model

In creating a message, the user inserts the address of the recipient into the header field within the message content. The MUA then copies this address, as well as the originator's address, into the associated SMTP envelope. When the user sends the e-mail message, the MUA makes a call to its associated MTA. Acting as an SMTP client, the MUA posts the message to the MTA. At this point the syntax of the message and validity of the destination addresses are verified. Once accepted, the MTA is responsible for delivery of the message. The role of the originating MTA then is to route or transfer the message to the delivering MTA that services the destination MUA. MUAs interface with human end users whereas MTAs interface with other computers (other MTAs, MUAs or MSs). MTAs are configured to serve one or more explicit mail domains as well as relay messages onward towards the correct delivering MTA. Users supported by a given MTA share these domains which are reflected in their user e-mail addresses. The goal of the RFC822 and RFC821 standards was to enable this inter-MTA/MUA function across the entire Internet thereby providing Internet-wide e-mail services.

As an interesting aside, until 1999 most of the MTAs throughout the Internet had no form of access control. Characteristic of its roots, the Internet user community was assumed trusted, or at least ethical. Anyone wishing to configure their MUA appropriately could use the transfer services of any other MTA - there was no need for access control as users were trusted to use the correct MTA. Unfortunately, experience forced system administrators to protect their MTAs as they were being abused. Computing resources financed and intended for specific users were being usurped by others. To make matters worse, these open MTAs

were also used as a means to launch e-mail attacks on third parties and conceal the true source of the attack. It would appear to the target that these open MTAs hosted the attack and could thus be held liable for damages. To protect the organisations sponsoring these MTAs, system administrators now restrict these mail transfer or relaying services to legitimate users.

Once the e-mail message gets to the correct destination MTA, the message is delivered to the MUA. In many instances, the MUA is not connected to the MTA at all times. In this case, the message is delivered to the MS. The purpose of the MS is similar to a mailbox in the physical equivalent of government postal services. It provides interim storage until the recipient is able to accept them. Utilising a MS frees the MTA from having to provide interim storage as well as many of the other functions associated with managing user mail boxes and interfacing with MUAs for mail delivery. The message will wait here until the user checks and downloads this message onto their local workstation using another protocol such as the Post Office Protocol (POP). The purpose of this protocol is to give the end user access to their electronic mailbox and the ability to download these messages for processing by the MUA on the user's workstation. This process, where the e-mail is downloaded, stored and processed on the user's workstation, is called "offline" processing.

Another alternative to POP is the Internet Mail Access Protocol (IMAP). IMAP provides a richer functionality in terms of being able to manipulate messages on the mail store without having to download and process them on the local workstation. IMAP provides a form of remote access allowing users to manage their mailboxes on the MS, from their

workstations. This is convenient for users that access their mail from a variety of workstations and/or MUAs. It prevents these users from having to scatter their mail across these various workstations/MUAs. E-mail processing using the IMAP is called either online or disconnected processing. Disconnected processing refers to a situation where connectivity between the IMAP server and the MUA is broken. This would prevent online processing; however IMAP does not prevent the MUA from downloading messages for local processing on the workstation. In disconnected mode, the user can process mail locally until IMAP connectivity is restored. At this point the remote (IMAP server) and local (workstation) mail stores are updated accordingly.

The following two tables compare these two mail access protocols as well as the three mail access modes: offline, online and disconnected [6]. This comparison does not assume the use of end-to-end security services. Use of these secure e-mail services would negate any concerns regarding security and privacy for all three access modes.

Criteria	POP Protocol	IMAP Protocol
Protocol status	Standards are complete and technology is mature. Clients and servers are widely implemented and deployed.	Progressing in the IETF standards process.
Support for offline mode	Designed for offline mode. Widely supported by clients, servers, and ISPs. Access from only one computer.	Possible but not widely implemented. Provides some advantages, such as downloading attachments one at a time.
Support for online mode	Possible in an extremely limited way but is generally not implemented or used.	Designed for online mode. Allows mail access from multiple computers connected to the network. Some IMAP clients and servers are currently available. Need for Application Configuration Access Protocol (ACAP).
Support for disconnected/Resynchronisation mode	Extremely basic, allowing limited resynchronisation of one mailbox per server.	Allows mail access from various computers that are intermittently connected to the network. Few clients support this mode, but development is under way. Need for Application Configuration Access Protocol.

Table 1 - Comparing POP and IMAP

Mail Access Modes			
Criteria	Offline	Online	Disconnected Resynchronisation
Mail access	Mail is downloaded to the client and deleted from the server.	Mail is stored on the server, with temporary copies downloaded to the client for reading and processing.	Mail is synchronised between the mail server and the various computers you may use to access mail.
Client/server connection	Brief, connected only when downloading mail.	Connection maintained for the entire mail session.	Connection required when loading and desynchronising stored mail.
Performance	Speed limited only by the speed of your desktop.	Speed limited by the speed of the mail server, network, and/or modem.	Speed limited only by desktop computer, except during desynchronising when speed is affected by network and server performance.
Security and privacy	You have complete control of your e-mail and are solely responsible for its security.	The ISP or IS manager has complete control of your e-mail and is responsible for its security.	You are responsible for securing each computer on which mail is stored.
Backup	You are responsible for backup.	The IS manager is responsible for backup.	Backup is nearly inherent, since the e-mail is always stored on multiple computers.
Advantages	Server-independent features and performance. Reduced dial-up and connection costs due to limited need for client/server connection.	Access your e-mail from different platforms anywhere on the network. Supports shared mailbox capabilities.	Access from multiple computers, without requiring a continuous network connection. Performance when processing e-mail is not dependent on the server or network connection. Supports shared mailboxes.

Disadvantages	Mail access available from only one computer.	Requires network connection, and is dependent on server and network performance. Dial-up costs are higher than in offline mode.	Resynchronisation may be slow and complicated to configure and operate. Technology is neither mature nor well understood.
Recommended users	Users who read mail from only one computer.	Users who read mail from multiple computers, all of which have network access at all times.	Users who read mail from multiple computers in multiple locations.

Table 2 - Comparing Mail Access Modes

The humble beginning of the Internet e-mail backbone is reflected in the fact that SMTP alone does not support multimedia file transport. It was designed to carry only 7 bit ASCII text and limits each line to 1000 characters [3]. Most other types of traffic today, such as multimedia or binary files, are in 8 bit non-ASCII formats. This severely restricts the use of Internet e-mail to transport anything but basic text. One solution is to upgrade the entire Internet backbone with mail gateways or SMTP servers that support both 7 and 8 bit data types. Although this movement has begun, the sheer size of the Internet, its geographic dispersion and the number of "owners" of these supporting computers makes this a rather daunting solution: at least to the point where one can depend upon every intermediate server being capable of supporting 8 bit data types.

A more acceptable solution is to somehow use the existing backbone to transport these various data types. Enter Multipurpose Internet Mail Extensions (MIME). MIME specifies a way to encapsulate these other data types into e-mail in such a way as they can be

transported via the Internet's SMTP backbone. The result is e-mail can now be embellished with all the modern multimedia. E-mail users can also attach any type of file to their e-mail messages and mail the entire "package".

2.1.2 Multipurpose Internet Mail Extensions (MIME)

Stated very briefly: MIME [7, 8, 9, 10, 11] specifies a format that provides a structure for the body of the e-mail message. This structure dictates where to find the components or "content-types" of the message body such as the data (e-mail message) as well as specific headers and control information needed to process the MIME structure. The control information, such as the content-types, identifies the type of data that is in the envelope. Entities that are MIME compliant (MUA, mail gateways or servers) now know where to look (format), what is there (content-type) and can now process this data accordingly.

The content-types used in this thesis include "plain/text", "multipart/mixed" and "application/x". Content-types are specified to ensure the receiver can unambiguously determine how the coded message was originally intended to be interpreted. Plain/text indicates that the contents of this body part are plain (unformatted) text. The default character set is US-ASCII meaning that unless explicitly stated otherwise a plain/text message is just an ordinary text-based e-mail message like they were in the "old" (pre-MIME) days. Multipart/mixed denotes that the body of this message has multiple parts or sub-bodies nested within it. The mixed tag indicates that these sub-bodies are themselves of different content-

types. It is this ability to nest or encapsulate multiple parts within a single message body that allows users to “attach” (i.e. encapsulate) files to e-mail messages. As it is now possible to have different parts within a message, something was needed to separate or identify these different parts. For this reason the “boundary” parameter is mandatory whenever a multipart content-type is used. Application/x is used to transmit data to be used by locally installed programs. For example, if a file created with Microsoft PowerPoint version 7 were attached to an e-mail message, the file would be tagged as “application/x-powerpoint7”. Upon receipt the MUA; if configured properly and assuming Power Point 7 was installed in the recipients machine, would hand off this file to that application for presentation to the user. There are many other content-types and sub-types that are specified in RFC 1521, as well as subsequent documents that have extended the number and types. Those types related to security are identified later under the title of Secure Multiparts for MIME.

MIME also specifies two different processes to convert 8 bit data into 7 bit data and back again. These conversions are called “quote-printable” and “base64”. These conversions are what enable SMTP to apparently transport 8 bit data. In actual fact 8 bit data is converted to 7 bit data by the MUA, transported via SMTP, and then converted back into 8 bit data after it is received at the destination. The reason there are two different conversion processes is because it was not possible to create a single process with both a human readable output and a highly compact or efficient encoding. Where the input data was predominantly, but not entirely, 7 bit data; a human readable output was desired to allow non MIME-compliant MUAs to present this data to the user. MIME-compliant MUAs were, of course, not as

prevalent in 1993 when this specification was accepted, as they are today. For the purposes of this thesis we will assume henceforth that all MUAs are MIME compliant. Also note that data that is already mostly ASCII text (7 bit) is often encoded to quote-printable to ensure data integrity. This “mail mangling” frequently occurs when mail passes through character-translating or line-wrapping gateways. The message is literally “processed” by these gateways one character at a time and may or may not be modified. Quote-printable and base64 are encoded in a way that greatly reduces the chance of these alterations.

Base64 encoding caters to the other requirement where a compact or efficient encoding scheme was needed and it was not important to be human readable. Note that the output of both these conversions is still ASCII, to be compatible with mail transport protocols such as SMTP. Base64, however, uses more efficient mappings, as it is not limited by the readability requirement. This encoding is also more appropriate for binary or “octet-stream” data as it is labelled in MIME.

Appendix A provides a complicated example of a multipart MIME encoded message. A simpler example is used here to help illustrate the MIME protocol: Kevin wants to send his classmates a short message with a copy of the slides that he is preparing for a class presentation. The presentation was done with version 7 of the Power Point application. Using MIME, Kevin’s MUA would create a MIME message as illustrated in Figure 4 - E-Mail Message MIME Formatted. Note the following points:

- a. The "MIME-Version: 1.0" header field within the RFC 822 header is a mandatory field for MIME messages and tells the MUA that this e-mail is a MIME message.
- b. The body of this message contains multiple parts delineated with the "--unique-boundary" indicator as denoted by the "Content-type: multipart/mixed; boundary='unique-boundary'" control information also in the RFC 822 header.
- c. The first body part is a simple text message from Kevin to his classmates.
- d. The second body part is the attached Power Point file containing the actual slides for presentation.
- e. This message is insecure. It does not engage any security services.

RFC-822 Header

From: Kevin Chafe<chafe@sce.carleton.ca>
To: Class<class_list@sce.carleton.ca>
Date: 20 May 99 01:10:57 CST
Subject: Crypto Presentation Slides
MIME-Version: 1.0
Content-Type: multipart/mixed;
 boundary="unique-boundary"

RFC-1521 (MIME) Body

--unique-boundary
Content-Type: text/plain
(blank line)
Hello everyone, take a look at these attached
slides will you? They are in PowerPoint 7 format.
Let me know what you think.
Thanks
Kevin
(blank line)
--unique-boundary
Content-Type: application/x-powerpoint7
Content-Transfer-Encoding: Base64
(blank line)

Attached PowerPoint7 File

(blank line)
--unique-boundary--

Figure 4 - E-Mail Message MIME Formatted

Using MIME the Internet e-mail backbone or SMTP network can now support multimedia files. Unfortunately, MIME does not provide any security services for e-mail and has; indirectly, increased the need for these services now that all manner of text and multimedia files are supported by SMTP. On the other hand, the same framework that MIME uses to support various media types can be extended to support secure enveloping protocols. These protocols can and do provide the necessary security services to e-mail. These protocols are built upon underlying cryptographic services. Before investigating these secure enveloping protocols a discussion of cryptography is given as a background to understanding how electronic messaging is secured using these cryptographic services.

2.2 Cryptography

Cryptography has numerous definitions, dependent on the context in which it is being presented. From a mathematical perspective, Paul van Oorschot et. al. state:

"Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication." [12]

Whitfield Diffie and Martin Hellman define cryptography similarly in their 1976 paper, "New Directions in Cryptography" [13]:

"Cryptography is the study of 'mathematical' systems for solving two kinds of security problems: privacy and authentication. A privacy system prevents the extraction of information by unauthorized parties from messages transmitted over a public channel, thus assuring the sender of a message that it is being read only by the intended recipient. An authentication system prevents the unauthorized injection of messages into a public channel, assuring the receiver of a message of the legitimacy of its sender."

In their 1979 paper, "Privacy and Authentication: An Introduction to Cryptography", it is defined as:

"... the use of transformations of data intended to make data useless to one's opponents. Such transformations provide solutions to two major problems of data security: the privacy problem... and the authentication problem...".

From a pragmatic perspective, Bruce Schneier, presents the following introduction:

"The process of disguising a message in such a way as to hide its substance is encryption. ... The process of turning ciphertext back into plaintext is decryption. The art and science of keeping messages secure is cryptography, and it is practiced by cryptographers." [14]

In any case, it is evident that cryptography is used to secure electronic messages. By securing electronic messages, we mean services are applied to the messages that enable the entities communicating to ascertain if certain threats have been subverted. In using these services, communicating parties are assured that the communication is legitimate. The term "services" is taken from the OSI reference model. To facilitate the understanding of how cryptography and electronic messaging are integrated; and clarify some of the terms used within this field of study, some general concepts are briefly discussed here.

The overall data or computer communications process is broken down into a series of seven basic steps or layers as they are represented in the OSI model. Each layer within the model has its own role and communicates with the adjacent layer in the model. Higher layers are dependent upon the lower layers to perform their layer-specific functions. Lower layer-specific functions are prerequisites to the subsequent processing at the higher layers. The exact functions required will vary with the type of communication being processed. Not all of

the functions within each layer are required for every communications process. Layers are called to perform these functions through a concept called primitives. Primitives are used within the OSI model to specify the functions to be performed within a given layer. In other words, layers communicate to their adjacent layers through primitives. These primitives specify the function required and pass the relevant data and control information between the layers. In practice, the realisation of these primitives varies with its implementation. A programmatic structure commonly used is a subroutine call. Any given layer will be aware of the subroutines available in the adjacent lower layer. When required, the given layer will prepare the required subroutine call and request the lower layer process the request. In providing this functionality to the given layer, the lower layer (service provider) is said to provide services to the higher layer (service user).

In much the same way, albeit at a much higher conceptual level, cryptography is used to provide a number of security services to electronic messaging. To simplify the analogy the reader may think of cryptography as a conceptual layer directly below the “electronic messaging” layer. The services provided by cryptography to electronic messaging can be categorised into four primary services:

- a. Confidentiality. Also known as secrecy or privacy, this service is employed to prevent unauthorised entities from being able to read or understand the message plaintext content.
- b. Data Integrity. This service provides a way for the receiver of a message to verify

that the message is the same as what was originally sent and intended by the sender. It prevents the unauthorised modification of the message data by a third party by providing a means for the receiver to detect any unauthorised manipulation. Conversely, this service provides the originator with the assurance that the message content will not be altered either accidentally or intentionally without the receiver's knowledge.

- c. Authentication. A service aimed at confirming an identity. In its most basic form, it provides the receiver with a means to confirm the identity of the message sender. In store and forward communications such as electronic messaging this is called data origin authentication. It prevents one party from masquerading or sending messages while claiming to be someone else.
- d. Non-repudiation. This service prevents an entity (typically the sender of an e-mail) from falsely denying that a message was indeed sent. If a dispute were to arise, it is critical to be able to prove the existence and exact content of a specific communication to a trusted third party.

It should be noted that cryptography is not limited to the above services. Nor are these services a complete listing of all the services required in the OSI security architecture. The focus here is on the primary services required for the connectionless or store-and-forward communications used by electronic messaging. To understand how cryptography is able to

offer these services, the following section offers a rudimentary background in cryptographic principles and concepts. The author has attempted to balance this presentation between rigorous theoretical mathematics and overly simplistic black box diagrams. Once these founding cryptographic principles are presented, the aforementioned security services will be summarised in light of the cryptographic primitives used to provide them.

2.2.1 Cryptographic Principles

A cryptosystem is a general term used to refer to a set of cryptographic primitives that provide information security services [12]. Cryptographic primitives are specific steps or transformations that collectively implement the required cryptographic functionality. Normally, cryptographic algorithms are used to transform the plaintext to and from ciphertext. These algorithms are capable of many different transformations. To be useful, both of the communicating parties must know which particular transformation was done. The algorithms are designed to accept a special parameter called a key. The key is used to specify which unique transformation to use to encrypt the message; and therefore, with knowledge of this key, enable the second party to decrypt the plaintext. A cryptosystem can be thought of as the family of all the transformations possible from the algorithm(s) used, the set of all plaintext, all ciphertext and all keys.

The core of a cryptosystem is its encryption and decryption functions. Not to be confused with the terminology used in the OSI security architecture, these are functions in the

mathematical sense and provide a mapping or transformation from one set to another set.

The next few pages borrow from Van Oorschot et. al. [12] to explain the fundamental mathematical principles upon which cryptography is built.

A cryptosystem provides the means to secure communications but is too general without specifying a unique key. Once a key is specified a subset of the cryptosystem has been identified. This subset of corresponding encryption and decryption functions is called an encryption scheme or more commonly a cipher.

Mathematically, a cryptosystem can be described as $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ where:

- a. $\mathcal{P}$ is a finite set of possible plaintexts (the plaintext space)
- b. $\mathcal{C}$ is a finite set of possible ciphertexts (the ciphertext space)
- c. $\mathcal{K}$ is a finite set of possible keys (the key space)
- d. There is an encryption function or transformation denoted $\{\mathcal{E}_e \mid e \in \mathcal{K}\}$ and a corresponding decryption function $\{\mathcal{D}_d \mid d \in \mathcal{K}\}$ such that for each $e \in \mathcal{K}$, there is a unique $d \in \mathcal{K}$ such that $\mathcal{D}_d = \mathcal{E}_e^{-1}$ or $\mathcal{D}_d(\mathcal{E}_e(p)) = p$ for every plaintext, $p \in \mathcal{P}$. Note that e and d are known as a key pair. Note also that e and d could be the same, that is $e = d$.

A function is defined by two sets X and Y and a rule f or mapping which assigns each element in X to one element in Y . Standard notation for describing a function is $f: X \rightarrow Y$. The set X is called the domain of the function and Y is called the codomain. Any given element in Y ($y \in Y$) is called the image of the specific element in X from which it is mapped. This is written as $y = f(x)$ where $x \in X$ and y is the image of x . Conversely, given y then x can be thought of as the preimage of y where $f(x) = y$. The set of all the elements in Y that have at least one preimage in X is called the image of the entire function, f , and is written $\text{Im}(f)$. The range of f is synonymous with $\text{Im}(f)$, the image of f . These terms will be used shortly to define the properties required of cryptographic functions.

The following is an example of a simple function. Set $X = \{1,2,3,4,5,6,7,8,9,10\}$ and $f =$ the remainder when x^2 is divided by 11 or $f(x) = x^2 \bmod 11$. The domain and codomain of this function can be written as:

X	1	2	3	4	5	6	7	8	9	10
$f(x)$	1	4	9	5	3	3	5	9	4	1

Table 3 - Domain and Codomain of a Simple Function

Therefore, $\text{Im}(f) = \{1,3,4,5,9\}$.

Clearly, cryptographic functions are much more complex. This next example, although not a cryptographic function, illustrates how easily the complexity of functions can change. Set $X = \{1,2,3,4,5,\dots, 10^{50}\}$ and $f(x) = x^2 \bmod (10^{50}+1)$. In this case, it is impractical to list the elements of the domain much less $f(x)$.

In addition to their complexity, cryptographic functions are also bijections and one-way functions. To define a bijection the mathematical concepts of “one-to-one” and “onto” are used. A one-to-one function is a function that has for each element in the codomain, at *most* one element in the domain. Note that this definition does not preclude one or more elements in Y from having no associated image in X . Given a function $f: X \rightarrow Y$, f is onto if $\text{Im}(f) = Y$. In other words, a function is onto if each element in the codomain Y has at least one element or preimage in the domain X . Conversely, a function is onto if there are no elements in the codomain without a preimage. A bijection is a function that is both one-to-one and onto.

Given that a function is a bijection, say from X to Y ($f: X \rightarrow Y$), it is intuitive that there is an inverse function mapping the same elements but in the reverse direction; say from Y to X ($g: Y \rightarrow X$). Thus, for each $y \in Y$, $g(y) = x$ for all $x \in X$ where $f(x) = y$. This inverse relationship is denoted as $g = f^{-1}$ and g is called the inverse function of f . Figure 5 - One-Way Function depicts f and g where $X = \{a,b,c,d,e\}$ and $Y = \{1,2,3,4,5\}$. Here, $f(a) = 3$, $f(b) = 1$, $f(c) = 5$ and $g(1) = b$, $g(2) = e$, $g(3) = a$.

Figure 5 - One-Way Function

Diffie and Hellman [13] defined a one-way function as a function that is straightforward to compute in one direction and practically impossible to compute in the reverse direction. It is interesting to note that both the idea of one-way functions and leveraging the difficulty of the integer factorisation problem for this very purpose was published in 1874 by W.S. Jevons [15]. This is exactly how the RSA cryptosystem was designed in 1978; one hundred years after Jevons and one year after Diffie and Hellman outlined the conceptual basis of public-key cryptography [16].

In their paper, Diffie and Hellman describe the basis of what became known as the Diffie-Hellman key exchange protocol. This protocol is not based on the integer factorisation problem. Enhancing work previously done by R. Merkle [17], they described how two individuals could establish a secret key using a public channel. This protocol is based on the fact that it is easy to compute $Y = a^X \bmod q$ but very difficult to calculate the logarithm of Y where $X = \log_a Y \bmod q$ for certain fields or sets of integers. The security of this key exchange protocol is based on the difficulty of computing discrete logarithms over a finite field

with a prime number of elements. This is a well-known mathematical problem and often referred to as the discrete log problem.

If one were to assume a cryptanalyst had unlimited time and computing power, there is only one cryptosystem that is unconditionally secure: the one-time pad¹. All other ciphertext could theoretically be decrypted, even if by a brute force attack. A Brute Force attack is an exhaustive search of the key space for a given cipher. A computer is programmed to try every possible key until the ciphertext is eventually decrypted successfully. This is usually a very resource and time intensive effort – so much so that alternative cryptanalytic techniques are developed to avoid having to resort to a brute force attack.

Thus a one-way function is not expected to be unconditionally secure, rather it must but be computationally secure: meaning sufficiently difficult to thwart all “reasonable” cryptanalytic attacks. The following function is a simple example demonstrating the notion of a one-way function. For $X = \{1,2,3,... 10\}$ and $f(x) = 5^x \bmod 23$, the codomain can be calculated and explicitly identified as:

X	1	2	3	4	5	6	7	8	9	10
$f(x)$	5	2	10	4	20	8	17	16	11	9

Table 4 - Codomain of a one-way function

¹ The one-time pad is a stream cipher where the plaintext is simply XORed with the key. The recipient has an identical key, which is simply XORed with the ciphertext to reveal the plaintext. Its perfect secrecy is based on the fact that the key is a truly random sequence and only used once. Disadvantages of the one-time pad are the key must be the same size as the plaintext and its key management: each recipient must have the correct key and use it only once. A more practical, albeit less secure, approach is to use a pseudorandomly generated key. This is called the Vernam cipher and was invented in 1917 for telegraph messages. It became the first automatic or on-line encryption machine and was used extensively throughout WWII.

Given the above, it is easy to calculate y for each value of x . It is not as easy, however, to find the correct value of x such that $f(x) = 10$ - without reference to the above table. Note another characteristic of one-way functions; there are a few elements in the range where it is relatively easy to find their pre-image. For example, given $f(x) = 5$ it is not difficult to deduce that the preimage $x = 1$. A domain with a large number of elements and carefully chosen function would complicate this process dramatically.

Although it is difficult to mentally compute the inverse of this function, it would be a simple task given the correct techniques and a computer. One-way functions used for cryptography are based on mathematical problems using numbers large enough to burden the computational power available today. Calculating their inverse is said to be computationally infeasible. It is often easier to simply mount a brute force attack in an attempt to discover the key, then it is to calculate the preimage values by solving and going through the inverse function. Thus, the key becomes the "weakest" link. For this reason the size of the key or more importantly the associated key space, is of critical importance in determining a cipher's resistance against a brute force attack.

The last concept introduced here is the trapdoor one-way function. Although they could not specifically identify a trapdoor one-way function, Diffie and Hellman proposed that trapdoor one-way functions exist and could be used as the basis for public key cryptography. The examples they provided were conceptual and had too many limitations to be practical for an actual public key cryptosystem. Nonetheless, they explained that a trapdoor one-way

function is similar to a regular one-way function in that it remains computationally infeasible to find the plaintext; the image, from the ciphertext; the range (i.e. to invert the function). The difference is that given some additional trapdoor information (the key) it becomes easy to compute this inverse and thus to decrypt the message. This is the basis of today's RSA public key cryptosystem. The following example provides a tangible illustration.

Given a function $f(x) = x^e \bmod n$, where $e=3$ and $n=2773$ yields $f(x) = x^3 \bmod 2773$. Like the previous one-way function, computing the image for any specific element in the domain is straightforward. The following table lists some examples:

x	5	10	15	20
f(x)	125	1000	1102	2454

$$\begin{aligned}
 \text{Where } f(5) &= 5^3 \bmod 2773 = 125 \\
 f(15) &= 15^3 \bmod 2773 = 3375 \bmod 2773 \\
 &= 1 \cdot 2773 + 1102 \\
 &= 1102 \bmod 2773
 \end{aligned}$$

Table 5 - Codomain for a trapdoor one-way function

Computing the inverse of this function is much more difficult. That is given the same function, where $n=2773$, $e=3$ and $f(x) = 1760 \bmod 2773$; find x .

Knowing that $f(x) = y = x^3 \bmod 2773$, we could solve for y as follows:

$$y^{1/3} \bmod 2773 = (x^3)^{1/3} \bmod 2773 = x \bmod 2773$$

or more generally, where $f(x) = y = x^e \bmod n$

$$y^{(e)^{-1} \bmod 2773} = x^{(e)(e)^{-1} \bmod 2773} = x^1 \bmod 2773$$

In this case, where $e=3$, this equates to finding the modular cube root of y with modulus 2773. Up to this point this is a typical one-way function. It is very difficult to find the inverse of the function or determine the pre-image that maps to a given image value. However, given some secret trapdoor information, it becomes very easy to find the inverse or pre-image for any image value. The trapdoor information is the value of the multiplicative inverse of e over the modulus n or the value of $(e)^{-1} \bmod n$. Here, $(e)^{-1} \bmod 2773 = 1779 \bmod 2773$ is the trapdoor information for the function used in this example. Thus, we can now calculate any inverse for a given image. The following expression calculates the pre-image for $y = 1760 \bmod 2773$:

$$\begin{aligned} x \bmod 2773 &= y(e)^{-1} \bmod 2773 = 1760 (e)^{-1} \bmod 2773 = 1760^{1779} \bmod 2773 \\ &\approx 5.848 * 10^{5773} \bmod 2773 = 25 \bmod 2773 \end{aligned}$$

Thus, for $y = f(x) = 1760 \bmod 2773$, we found $x = 25$. Further background and an explanation of how the trapdoor information itself is determined is provided at Appendix B - The RSA Public Key Cryptosystem.

Using these concepts, an encryption scheme or cipher can be devised to provide confidentiality services. Let $P = \{p_1, p_2, p_3\}$ be the set of plaintext messages or values. Let $C = \{c_1, c_2, c_3\}$ be the set of ciphertext values. Using functions having the characteristics as previously described each plaintext value will be mapped or encrypted to a unique ciphertext

value according to a key e . There are $3! = 6$ different possible mappings from P to C , each is a bijection related to the general encryption function but unique in its exact mapping which is dictated by the key. The keyspace is therefore $K = \{1,2,3,4,5,6\}$ or one for each possible bijection. The exact transformation is denoted E_k where k is the particular key used to specify a particular element of the set. The set of all these mappings may be thought of as a family of transformations, all generated from the same function. This function has a parameter, the key, which is used to generate and identify each unique bijection from P to C within the family of possibilities. A corresponding key, d , determines the reverse bijection from C to P . In some ciphers these corresponding keys are the same, that is $e = d$. In any case, the encryption and decryption mappings are inverses; that is $D(E(p)) = p$ for all elements of P . Because P is small this concept can be illustrated graphically as shown in Figure 6 - Simple Cipher or Encryption Scheme.

Figure 6 - Simple Cipher or Encryption Scheme

In practice P , or the possible combinations of plaintext, are practically infinite. Similarly, the number of possible mappings is related to the size of the key space. For example, a 128 bit key (2^{128}) creates a key space of approximately 340,000,000,000,000,000,000,000,000,000,000 possible keys. Algorithms and keys are used here much the same as the simple graphic is used above. They provide a means to identify and perform the encryption and decryption transformations.

Another point to note is that the key is not only valuable as a means to identify the particular mapping used for a given encryption; but also to maintain the usefulness of the cipher even after the cryptosystem is publicly known. A combination lock is the classic analogy used to illustrate this notion. There are, in reality, two types of combination locks: those with a fixed combination and others with combinations that can be changed by authorised users. Both types of combination lock are very effective at physically securing something, only because the combination is generally unknown. Should the combination somehow become known, the fixed combination lock would be useless and have to be replaced. The same is not true of the other type of combination lock. In this case, the physical mechanism remains useful; it is simply a matter of changing the combination so that the previously divulged combination can no longer be used to open the lock. The same is true of cryptosystems.

Figure 7 - Cryptosystem illustrates the physical manifestation of a generic cryptosystem where a piece of computer hardware and software is used to execute the

algorithm upon the plaintext, thereby encrypting the data and producing the ciphertext output. The key is used to determine the exact transformation computed.

Figure 7 - Cryptosystem

2.2.2 Symmetric Ciphers

Ciphers are classified as either symmetric or asymmetric. A symmetric cipher is an encryption scheme as previously defined with the additional characteristic that it is easy to determine the second key of the key pair given knowledge of the other. In most symmetric ciphers these two keys are actually the same. Thus the same key is used for both encrypting and decrypting the plaintext. Symmetric ciphers are also called single-key ciphers, secret-key, one-key or conventional ciphers.

Symmetric ciphers can be further classified into block and stream ciphers denoting the size of the input strings. A stream cipher can be thought of as a block cipher whose block size

is one bit. Stream ciphers are much more immune to transmission errors as each bit is encrypted individually. A bit error in a block cipher will effect the entire block and perhaps the entire message being encrypted. Most symmetric ciphers are block ciphers employing one or a combination of substitution or transposition techniques. Substitution replaces the original symbols with different symbols. For example, the plaintext characters or symbols for "A" could be replaced with the symbol "G". Transposition ciphers permute or move the original symbols to different relative locations within the block or string of symbol input. Product ciphers combine both these techniques to form more complex ciphers. Further, there may be more than two functions multiplied together. From a mathematical perspective, product ciphers implement a composition of functions. This concept is illustrated graphically in Figure 8 - Composition of Ciphers.

Figure 8 - Composition of Ciphers

Examples of symmetric ciphers are the Data Encryption Standard (DES), MARS, RC6, Rijndael, Serpent and Twofish². Symmetric ciphers can be orders of magnitude faster than asymmetric ciphers. Their biggest disadvantage is the key management required. Both parties have to somehow communicate securely to get the symmetric key before they can use the cipher! If they can already communicate securely then why do they need the cipher? The answer is bandwidth or the volume of data and the ability to leverage the results of this initial secure exchange. Symmetric keys are small amounts of data in comparison to the amounts of data exchanged in day to day business or human communications. The overhead and cost of protecting these keys is justified based on their ability to protect large volumes of other data. The mechanisms used to secure symmetric keys would be impractical if employed to protect general data communications.

2.2.3 Asymmetric Ciphers

An asymmetric cipher is a cipher that uses a key pair where each key is different. One key is published in the public domain while the other, the private key, remains secret to all but the owner(s) of the key. Unlike symmetric ciphers, knowledge of one of these keys does not facilitate the determination of the other key. This is not theoretically correct; but from a practical perspective it is very difficult for someone to break the cipher given knowledge of

² The latter five ciphers are the finalists in the US Government's competition and evaluation for the cipher that will replace DES. DES was adopted as a Federal Information Processing Standard (FIPS 46-2) in 1977. The winner will be called AES - the Advanced Encryption Standard. The AES program began on 12 September 1997 with these finalists announced on 9 August 1999. The standard should be completed by June 2001. Detailed information may be found at <http://www.nist.gov/aes>.

one of the keys. Trapdoor one-way functions are used to create this apparent isolation between the two keys. This unique characteristic gives asymmetric ciphers a distinct advantage over symmetric ciphers with respect to key management and the key distribution problem. Asymmetric ciphers are most often called public key ciphers because one of the keys can be made public without jeopardising the security of the cipher. Figure 9 shows a public key cryptosystem. It is identical to the symmetric system with the exception of the key pair. See Appendix B for a detailed explanation of the mathematical background and concepts behind the RSA public key cipher.

Figure 9 - Public Key Cryptosystem

There are some major differences in the application of public key cryptosystems.

Public Key ciphers can be used to provide:

- a. Confidentiality: Confidentiality is achieved in a way not unlike symmetric ciphers.

The main difference is that public keys are specific to each recipient. Thus, a single message going to x different recipients will need to be encrypted x times (once with each private key). Each recipient would then use their private key to decrypt the

copy of the message that was encrypted with their public key. This is clearly an inefficient process compounded by the relatively slow speed of public key ciphers.

- b. Integrity. A new cryptographic primitive, called a hash value or message digest, is employed to create an identity for a digital object. Hash values are used to verify an object's integrity. In an e-mail context the recipients can use this capability to verify the e-mail was not intentionally or accidentally altered. Cryptographic hash functions are used to generate a unique value or message digest for each message. A hash function is a one-way function whose input may be any arbitrary values but the output is a fixed length, unpredictable and unique value for every different input. The purpose of hash functions is to generate a concise, fixed-length, unique and irreversible identity value for each unique input value. Informally, a hash value or message digest can be thought of as a fingerprint for a digital object. Hash functions are not asymmetric ciphers. They do not constitute a cryptosystem; as there is no inverse function to "decrypt" the "ciphertext". They are used in conjunction with an asymmetric cipher as described under digital signatures.
- c. Digital Signatures. As the name suggests, digital signatures provide a means to bind an individual's digital identity (i.e. their private key) with a given digital object (e.g. an electronic document). Symmetric ciphers have no equivalent, as there is no unique identity for each user; all participants in a given communication share the same symmetric key. With digital signatures the signer wishes to show everyone else

that he or she has signed an object. Two identities are needed here: the identity of the individual signing the object as well as the identity of what is being signed, that is the object itself. Use of a given private key is accepted as authentication of the individual's identity. For the object being signed, its hash value is used - the same hash value that is used to verify a message's integrity. With a signed message, it is actually the hash value that is signed or "encrypted" with the private key.

Applying public key ciphers to digital signatures is the reverse of the process for confidentiality. An object is signed by encrypting or mapping the object's hash value from P, the plaintext space; into C, the ciphertext space, using the signer's private key. The public key can then be used to decrypt or invert this mapping from C back to P. It is disconcerting to think of "encrypting" an object into a ciphertext space under a key that is publicly available. In this context the "encryption" function is better thought of as a one-way mapping where only one individual can create the mapping but anyone else is able to reverse or verify it.

To verify a digital signature the recipient must generate his or her own local hash value by inputting the received plaintext message through the same hash function. The resultant value is then compared to the value that was "decrypted" using the public key of the signer. If these values match, it confirms that the reverse transformation was successful; the corresponding private key was used to sign the object and therefore who signed the object as well as that the integrity of the object

is preserved. This is analogous to a written signature in that an individual's hand written signature is assumed to be unique and difficult to forge, yet it can be verified by anyone that knows what the signature looks like. The signature identifies whom and the printed text that accompanies the signature identifies what is being asserted. Hand written signatures and their paper medium are subject to a variety of physical vulnerabilities threatening both integrity and abuse of an individual's identity through misuse of the signature itself. Procedural countermeasures are used to minimise this threat primarily in the form of legislation against fraud, false representation, impersonation and the like³.

As a point of clarification, there are two types of digital signature schemes: signatures with appendix and signatures with message recovery. Signatures with message recovery differ in that the message is altered to a specific form as part of the signature process. The cleartext message is not transmitted with the signature as the message can be recovered during the signature verification process. Unfortunately, this scheme is also prone to existential forgery attacks (possible to produce some messages that verify Alice's signature correctly where in fact Alice has had no part in creating or signing the message). Signature schemes with appendix require the actual

³ Bill C-6 (formerly C-54) entitled the Personal Information Protection and Electronic Documents Act is scheduled to come into force on 1 January 2001. It was accepted into the House of Commons on 4 April 2000 and is currently awaiting Royal Assent. The act provides a legal foundation for digital signatures, privacy issues and electronic commerce. This legislation is the equivalent to that already in place for hand written signatures with the exception that the vulnerabilities between these alternative forms of identification vary and therefore require different procedural countermeasures. Further information may be found at <http://strategis.ic.gc.ca/privacy>.

(cleartext) message as input to the verification process as previously described. The latter signature scheme is the most commonly used.

- d. Key Distribution. Perhaps the greatest single advantage of asymmetric ciphers is that key distribution is so much easier relative to symmetric ciphers. The trapdoor one-way characteristic allows the public key to be openly distributed. Thus the key distribution problem shifts from one of confidentiality to one of availability. A secure channel is no longer needed for key distribution. The issue becomes one of simply ensuring the correct public key is available to those that need it. The exact manner in which these public keys are distributed varies and can range from simply including a copy in outgoing messages to utilising central databases and directory services which provide public keys when requested. The critical point here is that the correct public key is used, otherwise vulnerabilities may be exploited such as a man-in-the-middle attack⁴. Public key certificates are the mechanism used to ensure the public keys requested are correct. Certificates are copies of the public key for a given identity together with a trusted third party endorsement that this key is correct for the stated identity. The trusted third party is called a Certificate Authority.

⁴ A man-in-the-middle attack can occur if the attacker can dupe Alice and Bob into using the attacker's public key instead of each other's. The man-in-the-middle will then intercept everything Bob sends to Alice, decrypt it using his own private key, read it and possibly modify the message and then encrypt it again; this time using the correct public key for Alice. The same can be done for messages from Alice to Bob. Bob and Alice will have no indication that their messages are being intercepted and possibly modified.

2.2.4 Hybrid Cryptosystems

A hybrid cryptosystem is simply a mix of symmetric and asymmetric ciphers. The motivation behind hybrid cryptosystems is to take advantage of the best features of each type of cipher to produce an even better overall cryptosystem. Symmetric ciphers are still used to encrypt message data (including attachments) because of their speed advantage. Asymmetric ciphers are used for all other services, including the distribution of the symmetric key that was used to encrypt the message data.

Graphical descriptions of the hybrid cryptosystem architecture and the e-mail security services commonly used in today's secure e-mail agents are shown below. These illustrations also serve as a summary of these security services and how the underlying cryptographic primitives previously described are employed to provide these services to e-mail [5].

Figure 10 - E-Mail Security Services - Confidentiality, Integrity and Authentication

Note that these services may be combined. For example, a message may be both sealed (encrypted) for privacy as well as digitally signed. Another service: named "Signed Receipt" is also shown. Using a signed receipt, an evidence service, known as non-repudiation with proof of delivery will protect against any attempts by the recipient to falsely deny receiving the message and its contents.

Signed Receipt

Non-repudiation (Recipient)

Figure 11 - E-Mail Security Services - Evidence

The illustrations above convey the fact that there are many steps and parameters that must be conveyed to the e-mail recipient, over and above the e-mail content itself. The message digest, various public or symmetric keys, timestamps, the names of the algorithms used and initialisation values or parameters for these algorithms are examples. This is where cryptographic protocols are used. A *protocol* is a specific sequence of steps involving two or

more entities designed to accomplish a task. In a messaging context a protocol is a sequence of steps to exchange messages between two parties. A *cryptographic protocol* is a protocol that uses cryptography to achieve a specific security objective. Needham and Abadi describe a cryptographic protocol as a protocol where whole or part of some or all of the messages exchanged is encrypted [18]. The following section introduces the cryptographic e-mail or secure enveloping protocols used to support secure e-mail.

2.3 Secure Enveloping Protocols

Secure e-mail is e-mail that has had some or all of the previously mentioned security services provided through the careful application of cryptographic algorithms. A secure enveloping protocol is a way to encapsulate or wrap e-mail using these cryptographic primitives and then communicate the required cryptographic and control parameters to the destination together with the message. The destination will then use this information to remove the applied security services and process the message. Using the classic letter and envelop metaphor, enveloping protocols can be viewed as a means to envelope the e-mail message, providing security services similar to those provided by the physical envelop. Once the letter has reached its destination, the receiver simply has to remove the envelope and read the letter. For e-mail this means the MUA must be able to recognise the “envelope” and then be able to remove it properly and reproduce the unprotected message. At this point the message can be processed by the MUA in the same manner as all other, non-secure e-mail.

In 1996 there were four primary security enveloping protocols competing in the industry. They were MIME Object Security Services (MOSS), Message Security Protocol (MSP), Pretty Good Privacy (PGP) and Secure MIME (S/MIME). These protocols are completely incompatible with one another. For reasons suggested in Section 2.4.1, these protocols were developed independently by different organisations and, in some cases, for different markets. MOSS, PGP and S/MIME cater to the general or business security community or market. MSP is regarded as the most complicated, although richest, protocol in terms of the security services it provides. MSP was developed for the US Military for use on their Defence Message System network. It was developed by the military because the other alternatives were insufficient for military messaging requirements. It is available to the public and may be used as an alternative commercial security enveloping protocol.

Each of these secure enveloping protocols is outlined in Appendix D. Following this is a 1996 comparison of these four enveloping protocols; labelled Protocol Comparison - Part One. This comparison [19] is the work of the IETF Electronic Data Interchange - Internet Integration (EDIINT) Working Group whose goal in creating the referenced document was to draft a functional specification for interoperable secure EDI over the Internet. The last part of Appendix D is a more current comparison of the PGP and S/MIME protocols.

Secure Multiparts For MIME

Secure Multiparts for MIME were developed specifically to let secure enveloping protocols take advantage of the MIME mechanism. Secure Multiparts are not a protocol in themselves but rather a MIME structure. Although it was developed jointly with MOSS and is intrinsic to MOSS, the Secure Multiparts structure is designed to be used by any secure protocol. RFC 1847 [20] specifies Secure Multiparts for MIME and defines two new MIME extensions (i.e. content-types) for this purpose. There are: multipart/signed and multipart/encrypted.

Each multipart content-type consists of two body parts; one containing the data and the other containing the cryptographic control information. The order of these body parts is explicitly defined in the specification. In the multipart/signed object, the first body part contains the message that was signed. The second body part contains the control information such as identifying the secure enveloping protocol, the hash and signature algorithms used, public keys and the actual signature data. An example of this construct is given in Figure 27 - Signed Message Using PGP and Secure Multiparts (i.e. PGP/MIME). This order was chosen to facilitate backward compatibility with MUAs that were not RFC 1847 compliant. These MUAs can process the first body part and at least present the message to the user. They cannot verify their signatures.

The multipart/encrypted object has the cryptographic control information in the first body part with the protected message data in the second part. This order is specified to allow

RFC 1847 compliant MUAs to decrypt incoming messages more efficiently. By first providing the MUA with the required cryptographic control information for the protocol used, the protocol implementation can decrypt the second body part “on-line” or as it is parsed by the MUA. Conversely, the MUA would have to wait until the entire message is received and parsed before having the control information required to unprotect the message.

The significance of these multiparts is that it provides a standard framework within which security services may be applied to MIME body parts in a complimentary fashion. This framework, together with a mechanism to properly interact with the required security protocol implementation, enables MUAs to provide security services to single-part and multi-part textual and non-textual messages.

2.4 Cryptosystem/Protocol Interoperability

As previously mentioned, secure communications protocols, with their underlying cryptographic technologies, are not as flexible as or tolerant as most other technologies. This poses significant complications with respect to interoperability amongst secure technologies. Cryptosystems utilise algorithms that restrict the interoperability of encryption and decryption transformations to specific mathematical functions operating over certain mathematical groups or fields. This is further restricted by the use of a unique cryptographic variable. It is precisely this lack of interoperability, or the fact that it is so difficult to participate in this exchange,

without the specific algorithm and cryptographic variable, that provides users of secure services with the confidence that these communications are legitimate.

Consequently, interoperability and security are often seen as mutually exclusive, opposing objectives. One is striving for an open and implicitly trusting communications environment designed to facilitate connectivity, while the other creates a very closely controlled communications environment designed to formally and explicitly enforce trust mechanisms. We shall see that it is possible to utilise cryptographic technologies to provide the required security services while simultaneously providing interoperability - *at the user level*.

2.4.1 Commercial Technologies and the Standards Development Cycle

Cryptography and e-mail technologies have each existed for a long time. Proprietary, non-interoperable solutions previously sufficed as they were used to provide secure e-mail capability to relatively small and manageable groups. For example, a company or consortium could use these products and achieve “inner-operability” through company policy and procedures. It is only recently that commercial markets have required secure e-mail technologies for the general public. Unfortunately, the development of commercial cryptography has been no different than other technologies exploited by industry. Prior to realising a threshold market size, there is little incentive to develop standards or to plan and prepare the technical infrastructure for a potential technology. There are never any guarantees that these markets will materialise as forecasted. Instead, the effort is postponed until after the

market is established. This mitigates the industry's risk of investing in options that do not become commercially viable. Once the requirement is evident and the market identified, industry members are well motivated towards developing the required technical infrastructure. One must keep in mind that these companies are first individual and competing commercial organisations; second, they are members of an industry. What is good for an industry is not necessarily good for any given company. Companies that are able to establish their products early in the market development cycle typically gain substantial advantage within an industry. Thus, the market is initially flooded with products. Each product is developed independently and is usually proprietary. The proprietary nature of these products reflects both the lack of any previously established standards as well as the drive for early market share. It is little surprise that these products are incompatible with each other.

The need for ubiquitous secure e-mail is now as fundamental as the need for basic security services themselves. Once users adopt secure e-mail, the immediate issue will be to ensure secure e-mail services are consistent and compatible across the Internet. One could even argue that this Internet-wide compatibility is an implicit prerequisite of secure e-mail technologies and that mass user acceptance of any secure e-mail will be contingent upon this compatibility. The confidence afforded by secure e-mail will be of little value to users if they cannot use secure services due to incompatibilities between the enveloping protocols and/or the cryptosystems providing these security services to different users.

It is now unanimous within the industry that ubiquitous secure e-mail is the goal. Note that this requirement does not necessarily demand true interoperability across different protocols and cryptosystems. It could also be met by standardising on a single security mechanism (protocol and cryptosystem). There are at least three alternative approaches to achieve this goal. Mapping these alternatives on a single line they can be characterised as left (De Facto Standards), right (Industry Standard) and middle (IETF) in their approach to the interoperability problem. Although the interoperable architecture proposed in this thesis is based on the third or middle approach; IETF, the other two alternatives are also mentioned as they have a significant influence on the market and hence the development of these commercial cryptographic technologies. These market forces can make or break the future of a technology, especially prior to any given technology being accepted as a de facto standard. The point here is that any given technology is not accepted based on technical merit alone: a number of business factors are relevant. Borrowing an example from the VCR industry, it is a well-known fact that the Video Home System (VHS) format was accepted over the Betamax format. What is not so well known is that the Betamax format was felt by many to be technically superior [21]. It was these market forces that helped VHS become the dominant technology. The same is true with interoperable secure e-mail technologies. In the author's opinion, the IETF approach is the "obvious" solution. It has the most merit with respect to solving the secure e-mail interoperability problem while also providing the flexibility required by changing security technologies. However, there is a feeling that the industry will adopt a single enveloping protocol as an industry standard [5, 22, 23].

2.4.2 De Facto Standards

One way to solve the interoperability problem is to make your own “standards” and base your architecture on these decisions. As previously mentioned, early in the development of a technology there is little in the way of industry standards or even the organisations and forums to establish these standards. Regardless of the state of the industry’s standards committee(s), the process of establishing any standards is much too slow for the industry; once it recognises a market. Companies have little choice: wait until standards are established while their competition gets a head start exploiting these new technologies or establish their own “company standards” and join the race. The risk of establishing your own standards is that interoperability with external or non-company networks may not be possible depending on the external company standards. In addition, once true standards are established they may be incompatible necessitating costly modifications to the company’s existing architecture. On the other hand, if enough of the market adopts your company’s technology, it often becomes a de facto standard.

De facto standards usually begin as short term solutions to relatively immediate requirements. They do not, in general, address interoperability at the same level as formal or de jure standards. De facto standards may initially address interoperability at the company or organisational level (i.e. providing secure e-mail within the company or between home-based employees and the company - a type of inner-operability). De jure standards have the benefit of hind sight, implementation and user experience with these technologies as well as any options or technological advances introduced in the mean time. De jure standards address

interoperability from an industry or world-wide perspective. One cannot easily compare de facto with de jure standards in terms of how they address interoperability issues - they do so at different levels and at different times in the technology's lifecycle. One is early in addressing "inner-operability" and the other is global in scope addressing true interoperability much later in time. Yet de facto standards have a significant influence on any efforts towards establishing formal, de jure standards because organisations have usually already invested in one or more de facto standards and are predisposed to these particular standards over any other competing standards.

2.4.3 Adopting A Single Industry Standard

If de facto standards are born at one end of the interoperability spectrum, where everyone makes their own standards; then the adoption of a single standard in the interest of interoperability must be seen at the opposite end of the spectrum; where the industry somehow dictates which protocol each company must use to interoperate.

In the case of secure e-mail interoperability, the industry could avoid many of the problems described earlier by simply adopting a single enveloping protocol as the dominant industry accepted protocol for providing secure e-mail. While this is a very easy technical solution, it is an extremely complicated management or business problem as it equates to telling the world to migrate towards a given cryptographic protocol that by default will likely also stipulate the ciphersuites or cryptographic algorithms to be used. Furthermore, the nature

and application of these technologies has a unique character. American high-tech industries dominate public cryptography technologies. The US government is so concerned about the export of cryptographic technologies that, until September 1999⁵ they were legally classified as munitions for export purposes. How is another country going to react when they are “told”, by an “American industry”, that they have to use a certain cryptographic protocol? What kind of trap doors does the protocol have that the US is not telling anyone about? The American people themselves did not accept this strategy from their own government when they tried to institute the Escrowed Encryption Standard (EES) using the Clipper and Capstone chips! [14]

Undocumented “features” and political troubles aside, there are other disadvantages to adopting a single industry standard enveloping protocol. As with any design decision there are tradeoffs and a balance is normally effected. This is difficult enough given a particular system. The problem in trying to achieve this balance across an entire industry is that no single solution or protocol will be able to balance all the requirements of all systems. A given protocol could be the best choice to one application and the worst for another application or requirement. For example, MSP is rich in features but significantly more complex than the other three protocols. The US military has stated it requires and will only use MSP ⁶. Should

⁵ For the past twenty years, since cryptography entered the public forum, the US government has had very stringent export regulations in an effort to control the release of these technologies and products outside its own borders. On 19 September 1999, the US government announced that the industry would be allowed to freely export cryptographic software. The only stipulation is that companies are required to submit these products for a one-time review and certification before being exported.

⁶ Of late the US military is expressing an interest in S/MIME Version 3 Extended Security Services (ESS). Although it is a different protocol, the ESS functionality is modelled on the MSP security services. The main difference, from an industry perspective, is the S/MIME V3 ESS has the potential to become a widely available commercial protocol whereas MSP has and will not.

the industry adopt MSP to be compatible with DOD and get all these features? Or would it be a serious mistake to further complicate an already complex and often misunderstood area in the industry?

Another problem with using a single protocol is the increased negative impact that a cryptographic or protocol failure would have across the entire world. A protocol failure in any of the contenders under discussion would be unusual given the amount and extent of the open, international scrutiny undertaken to date. This does not, unfortunately, rule out the possibility entirely and again highlights the relatively unique character of cryptographic technologies. Unlike other technologies where specialists are encouraged to publish their discoveries, weaknesses in these protocols or cryptosystems may have already been discovered, but *not* made public. This errant behaviour is motivated by the desire to break these security services in ways unknown to others, thereby enabling those aware of the vulnerabilities to survey the unsuspecting.

A brief example illustrating this withholding of information related to cryptographic vulnerabilities can be found in reviewing the history of DES. DES, or Lucifer as it was first known in IBM where it was developed, was submitted for consideration as the US government's encryption standard in 1974. The National Security Agency (NSA) was asked to evaluate the security of the algorithm. NSA modified Lucifer slightly and a smaller key length (128 to 56 bits) was stipulated; but no one was told why. The obvious conclusion was that the algorithm had been weakened and some kind of back door had been introduced. In 1995 one

of the IBM developers that had created Lucifer was permitted to publish a paper [24]. In the paper Don Coppersmith explains that Lucifer was modified before becoming DES to make it more resilient to differential cryptanalysis – a complex technique that narrows the key space based on observing specially selected differences between ciphertext pairs. Probabilities are assigned to different keys based on the differences in ciphertext pairs that they would produce. Given enough ciphertext pairs a small number of keys will emerge as the most likely candidates to have produced the given ciphertext. Differential cryptanalysis remains one of the strongest cryptanalytic techniques known today. It is more efficient than a brute force attack. There are two noteworthy points here. First, differential cryptanalysis was “discovered” in 1990 by Eli Biham and Adi Shamir [25]. That is approximately 15 years *after* the NSA modified Lucifer to counter this very type of attack. Secondly, the NSA actually strengthened DES. Subsequent analysis has shown that had Lucifer not been modified, it would have been substantially weaker to differential cryptanalysis – even with the longer 128 bit key length [14].

Cryptography and cryptanalysis is a constant cat and mouse chase. Once any cryptosystem is in widespread usage (and the motivation strong enough), it is only a matter of time until the cryptanalysis techniques catch up and break or seriously weaken a cryptosystem. Thus, a single world-wide protocol also provides a single target to attack and greatly increases the incentive to do so. Similar to developing a financial investment portfolio, in which investments are diversified across different markets, developing security standards with the flexibility to utilise alternative cryptosystems and protocols would balance these risks.

The benefits of multiple protocols are not unique to security. Analysts are beginning to support the need for diverse operating systems in an effort to immunise networks. Steven Telleen points out that his company suffered from a computer worm attack in June 1999 [26]. The worm only attacked MS Windows and Exchange-based mail servers. Nevertheless, damage around the globe was substantial, as large portions of the world's mail servers are Windows and Exchange-based. The author makes a direct analogy to the US corn crop failure in the 1970s. The genetic base of the corn had narrowed so much that a single virus almost wiped it out. It was only after that economic disaster that the agricultural industry began diversifying the genetic base of corn and other crops. He postulates that in considering the increasing threats of virus management, as well as the costs of recovery when damages do occur, it may be time to re-evaluate the savings and benefits originally attributed to homogeneous systems against this cost. There may be solid business reasons that now support a diversity of operating systems.

Regardless of the benefits of diversity, there is a belief in the industry that the S/MIME protocol will be adopted as the dominant enveloping protocol. S/MIME originally lacked some of the features found in other protocols; such as support for signed receipts and the PKCS upon which it is based is owned and controlled by a single vendor. Nonetheless the vendor (RSA), the US government and the IETF are working towards correcting these shortcomings in version 3 and ESS with the aim of achieving IETF standard status.

Proponents point out that all S/MIME implementations are “guaranteed” to be able to communicate securely with other S/MIME implementations (i.e. inner-operability) and it is widely used ⁷. Once again keeping in mind the commercial environment in which these technologies are competing, S/MIME has a significant advantage in that it was developed, supported and continues to be actively promoted by probably the most publicly well-known and highly regarded commercial cryptographic development company - RSA Data Security, Inc ⁸.

2.4.4 Internet Engineering Task Force (IETF)

The IETF has approached the interoperability problem with a compromise between the previous two schemes. More specifically the Common Authentication Technology (CAT) Working Group within the IETF has been committed to designing and specifying distributed security services in a way that shields users from the need to understand the specifics of any underlying security mechanisms. This enables a degree of uniformity and modularity in security solutions that can be used to facilitate secure interoperability. Instead of eliminating alternatives (i.e. adopting a single protocol) to resolve the interoperability problem, the IETF tried to make it almost as easy for the industry to employ *all* the protocols as it was to adopt a single protocol. Since we cannot modify cryptographic protocols or their underlying

⁷ More correctly, RSA is widely distributed and available as it has been bundled with the two main web browser products. However, most users do not yet use this functionality. [5]

⁸ The RSA company logo is: “The most trusted name in cryptography.”

cryptosystems to be interoperable with other protocols and/or cryptosystems, the IETF is trying to facilitate the use of all these protocols. The assumption being if it was easy to implement these protocols the developers would employ as many protocols as they needed. True interoperability would then be achieved while retaining the flexibility associated with being able to choose from among these protocols and select the one whose strengths are best suited to the particular requirements at hand.

The reader is reminded that these technologies are being developed commercially. It is crucial that the interoperability problem be seen from this perspective - it is not simply a technical matter. The heart of the secure interoperability problem is not with the secure technologies themselves. It is the difficulty and overhead involved in the developers utilising these protocols within applications that will determine the favoured solution. Stated in a different way, the best long-term technical solution will not be adopted by commercial developers if the associated overhead is felt to be "unreasonable", especially if S/MIME satisfies all their requirements and appears to be a better near-term business decision.

The core of the IETF solution is specified in RFC 2078 entitled, "Generic Security Services Application Program Interface, Version 2" (GSS-API). The first version of GSS-API was RFC 1508. Version 2 is dated January 1997 and represents a significant step in the acceptance of this solution as it is now on the IETF "Standards Track" as a "Proposed Standard". GSS-API specifies an interface that can be leveraged to create an interoperable security architecture. The following excerpt is from RFC 2078 [27]:

"The Generic Security Service Application Program Interface (GSS-API), as defined in RFC-1508, provides security services to callers in a generic fashion, supportable with a range of underlying mechanisms and technologies and hence allowing source-level portability of applications to different environments. This specification defines GSS-API services and primitives at a level independent of underlying mechanism and programming language environment, and is to be complemented by other, related specifications:

- a. documents defining specific parameter bindings for particular language environments*
- b. documents defining token formats, protocols, and procedures to be implemented in order to realise GSS-API services atop particular security mechanisms*

This memo revises RFC-1508, making specific, incremental changes in response to implementation experience and liaison requests. It is intended, therefore, that this memo or a successor version thereto will become the basis for subsequent progression of the GSS-API specification on the standards track."

A significant amount of work has been done to extend the concepts of the GSS-API for secure e-mail or store-and-forward secure communications. The effort to develop a generic security services interface for e-mail is detailed in RFC 2479, Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API) [28]. IDUP-GSS-API is the current state of truly interoperable secure e-mail and a central topic of the Chapter Four.

Chapter 3 CURRENT STATE OF SECURE E-MAIL

This chapter justifies the need for and describes the evolution of secure e-mail to date, identifies the two primary security enveloping protocols in use today and why there is a requirement for secure *interoperability* between protocols.

"We're in an information economy in which information is the product and messaging is the work."

A study, titled "Messaging Practices in the Knowledge Economy"[29], found that a company's competitiveness is determined by the ability of workers to gather, filter, and communicate relevant information to and with ones co-workers. This function is so critical that this year's study points out a complete reversal in thinking. Studies from the previous two years each emphasised how messages *interrupted* work. This year's study highlighted how integral messaging is *to* the work and included the above quote stated by Meredith Fischer, the vice-president of corporate marketing for Pitney Bowes.

The more important this messaging becomes to business competitiveness the more companies will need and want to protect this function. William Murray, an executive consultant for Deloitte & Touche, likens this inevitable transition to the airlines industry [30]. Airline safety is to the airline industry what information security is to electronic messaging and

the Internet. The airline industry realised the importance of airline safety to the success of the industry. Airline safety eventually became an enabler for the industry. If aviation safety had been maintained at 1937 levels, yet the number of users increased to today's traffic volumes; there would be two 747s crashing everyday. Needless to say, the airline industry would not be as successful as it is today if it had not recognised and acted to improve aviation safety.

In the case of e-mail this enabler takes the form of security services that provide basic integrity, authentication and confidentiality as well as non-repudiation, proof-of-origin and others. E-mail vendors and product manufacturers are trying to provide these secure e-mail services through various secure e-mail product offerings. An e-mail vendor has four possible paths to provide these services. These are:

- a. Develop their own software to provide these services
- b. Integrate third party security software into the application through custom software development
- c. Integrate third party security software into the application through proprietary APIs
- d. Integrate third party security software into the application through open APIs

These options can also be viewed as an evolution in the integration of security services and e-mail applications. The first two options have significant disadvantages yet they were common practice in the recent past. Only of late is this slowly changing in favour of the

latter two alternatives. Given the increasing cost and complexity of developing security software, development in this area is being left to those vendors whose core business involves the development of these specialised modules. Using operating system software as an analogy, e-mail application developers developing their own security software would be equivalent to having application developers code low level, privileged interrupt routines vice simply making an operating system call for the required service; clearly a more complex, less desirable and problematic situation. Thus, for most e-mail application developers the option to develop their own security software is no longer feasible. Similarly, it is not practical for either the e-mail vendors or a user organisation to customise existing organisational messaging systems to accommodate and maintain unique tailor-made software solutions. Fortunately, there are now other options.

The third option is predominant today. Security vendors sell cryptographic toolkits so that application developers can integrate third party security software into their applications. A toolkit is composed of all the source code, DLLs, developer manuals, and procedure calls required to integrate the proprietary security software into the target application. In other words, cryptographic toolkits are simply what is needed to help the application developer tailor the target application to the security vendor's proprietary API. A significant advantage over the first two options is that the toolkit (the proprietary API) isolates the application developer from the complexities of cryptography and the cryptographic software/libraries. The application developer need only know how to use the toolkit correctly and not the intricacies of the security code, protocols and algorithms behind the toolkit. Unfortunately, this leaves

the developer with a proprietary interface to the required security services, which may or may not be proprietary themselves.

The last step in this evolution is to have these target applications tailored to a standard, open API. The process could be very similar in that a toolkit could be used to modify the target application to conform to the standard API. The advantage of an open API is that this modification would only have to be done once for any future security services (unless the API itself was to change). The target application is now isolated from the complexities of security as well as any proprietary concerns. Application developers are now free from having to understand all but a few standard function calls. A standard API would enable various components of the secure software architecture to be replaceable with a minimum impact on the rest of the architecture. The software design is now modular and gains with respect to flexibility. This has far reaching effects throughout the lifecycle of the application. The initial application software development is faster and cheaper as there is no need to understand complex cryptographic algorithms and software. The integrity of the security services is maintained as none of the specialised security code needed to be modified and there is no impact should the need arise to change or add different supporting cryptosystems after the application is deployed. In this case the supporting cryptographic DLLs are simply replaced, without the need to modify the application software. This has substantial business benefits and facilitates interoperability between secure applications. Thus, from vendor and user perspectives, a standard API provides maximum flexibility in selecting and using different security services.

3.1 Custom Development and/or Integration

Custom development and integration are no longer practical in the effort to commercially develop secure and interoperable e-mail. They are included here for completeness and historical relevance. With respect to securing e-mail services, organisations (mainly government or large corporations) were at one time on their own to develop their own secure e-mail applications as there was no other alternative. Eventually generic security applications became available but these were typically developed to a specific requirement and made publicly available afterwards. Implementing these generic modules required a significant effort to integrate with an organisation's other applications.

As previously stated, there are a number of notable disadvantages from a commercial perspective. These disadvantages include the cost of developing the software itself and/or the customisation required to integrate unique modules into existing applications. This approach forced each new combination of application and cryptography to be treated as a distinct development effort; including different versions of the same application or security software. In addition, any change to either the security or the application could require modification of the other and perhaps its interfacing code. This situation does not provide the modularity and maintainability expected and required of commercial products. Other relevant factors are the specialised skill sets required to correctly implement security software, lack of available security specialist personnel, the rapidly changing technological environment and the pressure to compete in "Internet Time".

Need For APIs

There were, however, some lessons learned that have benefited today's technology. For example, the need for security APIs became evident. APIs provided a means to isolate the security software from the application software while simultaneously enabling the application to utilise the required the security services without being hard coded into the application. This facilitated the development of security software by security specialists and freed the application developers from the complexities of cryptography and implementing these secure services.

The impact of this separation and the use of APIs is enormous. As previously stated, it permitted the establishment of a secure software sub-industry within the global software industry. Without the opportunity to focus specifically on security, it is doubtful the industry could have advanced as far as we have. Bruce Schneier [31] points out that security is different from other software in that functionality does not determine good or bad security software: the problem is that 99.9 per cent of the users cannot tell the difference. Just because secure software seems to execute correctly does not mean that it is actually secure. Beta testing and most other software development tools are functionality centric. Unfortunately, no amount of beta testing will find security flaws. A good security product must resist every possible attack, even those from the future and not yet seen. To make matters worse, attackers only need to find a single weakness, they will cheat and they can wait for new technologies to give them tools they may not currently have.

In his work evaluating security products and protocols, Schneier gives several examples of the kinds of implementation mistakes often made by developers that lack a thorough understanding of security principles. In many cases, developers did not ensure plaintext, keys or passwords were inaccessible from memory or disk after encryption. Sometimes the same data was encrypted with two different keys. The problem was that one key offered strong protection whereas the other key was very weak. In another case, the S/MIME version 2 protocol was analysed. The protocol itself is well designed and capable of providing good security to e-mail. Unfortunately, the specification [32] mandated a very weak cryptographic algorithm [33] as the default. This was not a mistake per se but the results of American export control laws on cryptography. Nonetheless, this may be an important factor for a user who might otherwise assume that the optional and stronger TripleDES [34] algorithm was being used.

Similarly, Roger Needham and Martin Abadi justify the need and value of using an API by stressing the complexity and difficulty of developing good security protocols [18].

"It has been evident for a number of years that cryptographic protocols, as used in distributed systems for authentication and related purposes, are prone to design errors of every kind. A considerable body of literature has come into being in which various formalisms are proposed for investigating and analyzing protocols to see whether they contain various kinds of blunders."

The message to e-mail user agent developers and the information technology industry in general is do not develop your own protocols or implementations. Instead, use tried and true third party modules. Understanding the need and importance of using these

third party components gives rise to the requirement for an API. The role of the API is to integrate these components, provide a demarcation boundary separating secure and non-secure functionality (so development can be properly allocated to the appropriate development specialists) and to provide the flexibility and modularity required in a COTS development environment.

The use of APIs is so fundamental to the integration of security into applications that they are becoming critical enablers to companies within the software industry. Tom Casar and Richard Estabrooks [35] explain that this is no longer a purely technical issue. The fact that information technologies are complex and diverse is well known. APIs are being used as catalysts in a reaction where commercial and competitive pressures have complicated matters.

From a developer or manufacturer's point of view it is no longer possible to offer a solution that will meet all a customer's requirements without teaming or otherwise integrating products and solutions from other vendors. This teaming process is dynamic; creating alliances whose membership is expected to eventually change due to business trends and influences. An organisation's ability to team effectively within this environment is directly related to its ability to integrate its products into many different platforms, languages and applications. The most efficient way to do this is using an API; be it proprietary or an open standard.

Thus, utilising an API can no longer be considered simply as a user-friendly feature provided to an end customer by the vendor. It is also a business necessity required by the

vendor to facilitate the integration of their products into the applications and products of their alliance partners.

Like the applications that APIs were written to support, APIs themselves vary considerably. They range from very specific hardware interfaces to a high-level generic software interface. APIs may also be proprietary or open. Some were developed and supported by individual companies and/or consortiums whereas others are the results of formal standards bodies. In an effort to compare and evaluate these various APIs, the NSA formed a Cross-Organization CAPI Team in 1995 to assess the current state of security APIs and determine which API would meet the needs of the US Government [36]. Their recommendation was not a single API but rather a suite of APIs and an ongoing study to track the progress of commercially developed security APIs. In the course of this work, the NSA developed a list of criteria to assess these APIs. A listing of these criteria is presented here, before discussing APIs further, as it provides a framework to assist the reader in understanding the nature of these cryptographic APIs, their strengths and weaknesses. Detailed explanations of these criteria are provided in Appendix C. The criteria are:

- a. Algorithm Independence. The CAPI must provide access to a large number of choices of current and future cryptographic algorithms.
- b. Application Independence. The CAPI must provide cryptographic service to a variety of applications being written today and to many new ones in the future.

A well-designed CAPI should be able to support coding of a store-and-forward application, like e-mail, as well as a connection-oriented application, like file transfer.

- c. Cryptomodule Independence. The property that the CAPI, in providing its cryptographic service, can use any current or future cryptomodule as easily as any other.
- d. Cryptographic Awareness. Cryptographic awareness refers to the amount of cryptographic knowledge the application programmer is required to have to use the API correctly. It is also used to characterise how tightly integrated an application is to its security software and how much control the application has over these security services. CAPI specifications can fall anywhere in the spectrum from requiring little knowledge to requiring a great degree of expertise in cryptography. See Figure 12 - Cryptographic Awareness Property below.
- e. Design and Auxiliary Services. Modular Design is the division of the CAPI into units which work together to provide the complete security service, each of which has a focused purpose. Auxiliary Services are support services used by the goal cryptographic service. For example: key management, authentication and certificate management.

- f. MISSI Support. MISSI stands for Multilevel Information System Security Initiative: a NSA program for developing and delivering data and network security products for the U.S. government. MISSI support is the ability of the API to be extended to support current and anticipated MISSI cryptography. In other words, it must support legacy security systems used by the U.S. government. This criterion is immaterial for non-US government systems. For the purposes of this thesis this criterion is more appropriately labelled Backward Compatibility and uses the same concepts applied in the general case (i.e. S/MIME and/or PGP backward compatibility).
- g. Safe Programming. The term used for describing steps taken to guard against inadvertent security mishaps by the programmer. Three aspects of safe programming are consistent naming conventions, information hiding, and ease of use.
- h. Security Perimeter. The boundary that prevents sensitive security-related information from leaking out of the trusted computing base into untrusted applications. The CAPI provides the access between the untrusted components and the trusted components. The CAPI must not be used as a portal to violate the security perimeter; it must restrict access to sensitive cryptographic data (e.g., unprotected keys). It must not propagate such information beyond the CAPI interface.

All of these criteria are important. Their relative importance varies as a function of the perspective or role of the evaluator. For those that implement cryptographic algorithms, the security perimeter and the independent characteristics (algorithm, application and cryptomodule) would be most important. For an application programmer the most critical is cryptographic awareness. Indeed, the whole point of the API is to isolate the application developer from the complexity; yet still provide the required security services. Especially given the current expectations for the future and experience to date within the industry. This characteristic, labelled cryptographic awareness, is what distinguishes open generic security service APIs from more granular and specific cryptographic APIs. The following figure illustrates this concept graphically:

Figure 12 - Cryptographic Awareness Property

3.2 Proprietary APIs (Toolkits) For A Given Protocol

Proprietary APIs or toolkits are the state of the art with respect to commercially available secure e-mail. Appendix F lists a number of proprietary toolkits and their attributes. Note the diversity amongst these products. The only common characteristics between them is that they are all available in the C language and can execute on the Windows and Sun Solaris platforms. These toolkits are API and protocol specific and do not resolve secure interoperability across these protocols. The e-mail application developer must first decide which cryptographic algorithms to use, the security enveloping protocol to use, then choose a cryptographic toolkit that supports these algorithms and protocol. Finally, the developer must consider the nature of the associated proprietary API and toolkit that must then be integrated into the application. The purpose of a cryptographic toolkit is to define and describe the security vendor's API through which the security services of the underlying cryptomodule and algorithms can be obtained.

Whereas the goal of the cryptographic toolkit is to ease the integration of the API itself, the goal of the API is to ease the integration of the third party cryptographic software and services into the application. A good API will integrate as many security services as possible while simultaneously maintaining the modularity and flexibility required to integrate various types of cryptographic software modules or libraries. One of the disadvantages of using a proprietary API is that the security of the application becomes dependent on the will and ability of the security vendor to integrate these various types of cryptographic services.

Another is that many toolkits are designed to facilitate their integration with certain applications and data. This is effective for those particular applications but problematic for others. Three of the four common APIs listed in Appendix F (GSS-API, JCE and PKCS# 11) are for on-line or session based communications, the Java language and cryptographic hardware tokens respectively. The fourth API is the vendor's proprietary API, which unfortunately covers store-and-forward applications such as e-mail. These are just some of the reasons that choosing a toolkit (i.e. proprietary API) is an important strategic decision for the application developer.

According to Casar and Estabrooks [35], selecting cryptographic algorithms is one of the most important decisions an application developer can make with respect to security as it could have profound effects on the utility of the application. Without a standard API, the developer must choose the proprietary API that best meets his needs as previously explained. Should something change and algorithms that are not supported by the previous toolkit be required, the application must be recompiled and reconfigured by the developer to incorporate the required changes under a new toolkit that does support the required algorithms.

3.3 De Facto Secure Enveloping Protocol Standard

Only two years ago there were four incompatible secure enveloping protocols from which secure e-mail vendors would have to choose from and support. In 1999, there remains only two. The MOSS and MSP protocols are effectively dead from a commercial perspective.

For various reasons the commercial sector never did endorse MOSS. MSP, originating from DoD's Defence Messaging System was offered to the public as it offered significant security services lacking in all other protocols. Unfortunately, MSP was also a complicated protocol and subject to the prejudice aimed at government sponsored security solutions. Furthermore, the market was still relatively immature. Most companies were still trying to sell security in its basic form much less trying to convince users of the need for the more advanced features of MSP. Although MSP itself is no longer a viable option commercially, it did have a lasting impact on the development of S/MIME as explained below.

The choice of secure enveloping protocol has narrowed to two contenders: PGP and S/MIME. Marketing and emotional claims aside; it is not clear which one will become the dominant protocol used on the Internet. What has become clear is that there are two very distinct markets both wanting the same security services but from alternate enveloping protocols. This division is not so much the result of technical differences but rather historical and/or cultural. PGP is the popular choice for protecting individual privacy. This is a reflection of PGP's history. It was originally developed as a stand alone file encryption utility to provide confidentiality for e-mail. It was soon associated with the political fight against the United States government for individual privacy in which PGP and Phil Zimmerman (the original developer) became the figurehead. The US government tried to use Phil Zimmerman as an example by attempting to imprison him for contravening US legislation that prohibited the exportation of cryptographic technologies. The following is an excerpt from the PGP User's Manual, version 6.5, written by Phil Zimmerman:

"One more point about my commitment to cryptographic quality in PGP: Since I first developed and released PGP for free in 1991, I spent three years under criminal investigation by U.S. Customs for PGP's spread overseas, with risk of criminal prosecution and years of imprisonment. By the way, you didn't see the government getting upset about other cryptographic software— it's PGP that really set them off. What does that tell you about the strength of PGP? I have earned my reputation on the cryptographic integrity of my products. I will not betray my commitment to our right to privacy, for which I have risked my freedom. I'm not about to allow a product with my name on it to have any secret back doors."

Phil Zimmerman eventually won the case (rather the US government decided not to pursue it further) but not without wide spread support from multiple privacy and individual rights groups. The long-term side effect of this action was a large e-mail user base, with many having a deep distrust of the US Government and any cryptographic products they supported. PGP has become the preferred secure enveloping protocol for these users.

S/MIME is the other protocol that is widely supported, this time by the corporate community. As with PGP, this is primarily due to its heritage. S/MIME has always targeted the business sector and up until recently was marketed more commercially than any other protocol. In the earlier stages of S/MIME development there were advantages more attractive to corporations. For example, the use of X.509 certificates was important to organisations. It provided a centralised means to control and more easily manage individual certificates as well as cross certification relationships. It also scaled very well across large corporate staffs. PGP on the other hand catered towards a "web of trust" model where individuals were expected to make their own decisions about who to trust and not expect (or want) an organisational entity to do this for them. In any case, the two protocols have converged towards each other in

terms of their ability to address both of these markets. A comparison of PGP and S/MIME is provided at Appendix D.

From a developer's perspective, a single winner should not be assumed. Thus, the need for a flexible product architecture that will allow both or either protocol, as well as different supporting cryptosystems, is critical to the survival of a secure product.

3.3.1 S/MIME

Most S/MIME compliant MUAs currently comply with S/MIME version 2. This is not an IETF standard and will not be adopted as such. Version 2 specifies proprietary RSA algorithms. In an effort to standardise the S/MIME protocol, the IETF has commissioned a working group to establish S/MIME version 3 that will become an IETF standard. Version 3 replaces the proprietary RSA algorithms with open unencumbered algorithms. The work to date is described in the following RFCs:

- a. RFC 2630 Cryptographic Message Syntax (CMS) - An enhanced PKCS #7 syntax, backward-compatible with S/MIME Version 2
- b. RFC 2632 S/MIME Version 3 Certificate Handling - Mandates that S/MIME Version 3 MUAs must support version 3 X.509 certificates
- c. RFC 2633 S/MIME Version 3 Message Specification - Defines the MIME encoding for CMS messages

- d. RFC 2634 Enhanced Security Services for S/MIME - Specifies the optional security services used with CMS

The MSP protocol had a significant influence on S/MIME. The military is a very experienced and long-time user of its own military grade cryptosystems and networks. It is not surprising then, that their requirements for a commercial grade security protocol would reflect this. Back in 1997, it was the author's opinion, that the commercial security community would see a slow but steady increase in the demand for MSP, or its equivalent richness, as non-military security users became accustomed to secure applications and eventually developed the need for these advanced security services. Once this happened, MSP would no longer be considered in a class of its own. The non-military security community would take a renewed interest in this protocol, even if for no other reason than to emulate its functionality in other, more commonly used protocols. This emulation is exactly what happened. The result is S/MIME v3 ESS. ESS provides the same advanced security services that MSP offered but as a set of enhancements to the S/MIME protocol. The advanced services include such things as signed receipts, security labels and secure mailing lists.

3.3.2 Open PGP

The Open PGP effort is focused on retaining compatibility with deployed PGP. This is a highly desirable design goal given its large user base and historic focus on ease of use, which includes maintaining background compatibility. Unfortunately, this and the group's apparent lack of concern over other interoperability issues will also make full

integration with Internet Mail unlikely. For example, the group is choosing unique algorithms, such as its own variant of Triple-DES. These actions will ensure complete non-interoperability with S/MIME. Other changes have been more conventional. For instance, PGP has moved to the primary use of non-RSA technologies, in time for expiration of the Diffie-Hellman patent as well as the incorporation of PGP & MIME. However, even with the specification (and use) of PGP/MIME, the approach to PGP packaging continues to emphasize independence, so that the “armor” around the control information is retained inside the MIME body-part

Work is progressing with respect to interoperability and certificates. PGP has always had a different trust model from S/MIME. This is explained further in the next section. Having different trust models resulted in PGP having a different certificate than the X.509 standard. PGP products now support the X.509 certificate format as well. These PGP applications can issue X.509 certificate requests to X.509 Certificate Authorities, using the data from PGP keys to make the requests. In addition, they can import X.509 certificates, which may or may not be associated with existing PGP keys. In all cases, care is taken to map the X.509 trust model pertaining to the imported certificate into the PGP trust model concept in the PGP certificate. The PGP format itself in fact, is being added to several standards such as the IKE and TLS. Eventually, all PGP enabled applications will be able to use X.509 or PGP certificates interchangeably. All the systems that currently rely upon X.509 as a certificate data format will then be able to retain the PGP format advantages of increased reliability, multiple

signatures in a certificate, improved scalability and improved cross-certification through these PGP applications.

The work to date is described in the following RFCs:

- a. RFC 2015 MIME Security with Pretty Good Privacy (PGP) - This is the PGP/MIME specification predating OpenPGP but it is still applicable
- b. RFC 2440 OpenPGP Message Format - The OpenPGP specification

3.4 Trust Models and Digital Certificates

Once a secure system is properly implemented, providing the proper security services is dependent on users using the correct public keys. Digital certificates are the means commonly used to bind public keys to individual's identities. A trusted third party binds these two objects by cryptographically signing them (with the third party private key). The signature of the authority issuing the certificate attests to the fact that the public key enclosed is correctly associated with the entity identified in the certificate. The validation process is as follows: the originator gets the certificate for the receiver's identity then extracts the receiver's public key from the certificate. To ensure it is the correct public key for the receiver, the originator can validate the signature of the trusted third party. The public key for the trusted third party is well known or widely available. It is used to validate all the certificates issued by this certificate authority. Assuming users trust this

third party, this trust is now transferred to the new public key. That is, the originator trusts that it is the correct public key for the recipient identity based on the certificate authority's trusted signature. The security infrastructure that is required to support this process is called a Public Key Infrastructure or PKI. This is a very simplistic description of the key management and verification process. It is mentioned here to introduce the concept of trust models.

Trust models are simply the rules used to determine if a certificate is valid. They are also called certification topologies and are used to explain trust relationships between multiple certificate authorities. Trust models detail how certificates issued from one authority can be verified by users certified under a different certificate authority. Electronic commerce, in its many forms, depends on secure communications between multiple organizations. Secure communications depend on correctly validating public keys which, in turn, depend on trust models. It is important to recognize and understand differences in trust models. Where one application or user may correctly verify an entity using one trust model, another user may prove the same entity invalid or be unable to verify the entity using another trust model. The concept of certificates and key validation is not unique to e-mail. There are different types of digital certificates. Session based web browsing and the Secure Sockets Layer (SSL) protocol has it's own version of certificates. For our purposes, we will focus on PGP and X.509 certificates.

X.509 certificates are the standard certificate format that most applications try to support. These are the certificates used by S/MIME. The name, X.509, reflects the fact that the International Telecommunication Union (ITU) has drafted a recommendation (standard) for this certificate format and implicitly the underlying trust model. The standard is much broader in scope than simply certificates including authentication techniques, key distribution, cross-certification and more. It provides a common way of managing certificates for use in securing applications such as secure e-mail. The X.509 standard is also known as ISO 9594-8 from the International Standards Organization (ISO). The X.500 series of recommendations deals with issues related to Directory Services. Certificate repositories or servers are used by certificate authorities to store certificates and make them available to the public. These certificate repositories are directory servers. Often other directory information for the domain is also stored on these same servers. PGP certificates contain essentially the same content as the X.509 certificates but they have a different format and use a distinct trust model.

The major difference between the two certificate standards is that PGP was designed to allow many authorising identities and signatures per certificate whereas X.509 permits only one authorising signature per certificate. In other words, PGP has done the same thing with its certificate as it did with PGP raw keys – it uses the Web of Trust (model). In this model, a PGP certificate is valid if it is signed by a trusted individual whose certificate is valid, or if it is signed by a set of cumulatively trusted individuals whose certificates are all valid. In the X.509 model, a certificate is valid if there is a hierarchical path of valid certificates that lead to a

certificate from a trusted authority. This path, called a certificate path, is an ordered list of certificates. The path starts at the certificate whose issuing authority is already trusted and ends at the authority that has signed the certificate of the public key that we want to verify: it is a chain of trust from a trusted entity leading to the unknown entity.

3.4.1 S/MIME Trust Model

The S/MIME trust model is called the Hierarchical Model. There are actually numerous types of hierarchical trust models ranging from strict hierarchies to multiple rooted trees to distributed or digraph trust models. The difference between these models is the relationships between the certificate authorities. For explanatory purposes the strict hierarchy will be illustrated and is shown in Figure 13 - Strict Hierarchical Trust Model

Having a single authorising signature per certificate necessitates a hierarchical relationship between certificates. Unless you are the certificate authority for an entity, you cannot have a direct relationship with this entity as there is only one authorising signature or relationship. Instead, it must be indirect through a certificate chain (i.e. a series of single relationships). Verifying the certificate path requires that each certificate from the root authority to the certificate in question be valid. A brief example will illustrate this concept. Bob will trust that Sylvia's certificate contains the correct public key for Sylvia if every certificate in the chain from the root to Sylvia is valid. The validation sequence would flow as follows: Bob gets and verifies the root certificate then extracts the CA public key. Using the

CA's public key Bob verifies CA 2's certificate, then extracts CA 2's public key. CA 4's public key is then verified by validating CA 4's certificate using CA 2's public key. CA4 will be trusted if we believe that CA2 signed CA4's certificate: stating CA4 was trusted as a CA. Finally the public key for CA 4 is extracted and used to verify Sylvia's certificate.

Figure 13 - Strict Hierarchical Trust Model

3.4.2 PGP Trust Model

PGP uses a Cumulative Trust Model, often called the Web of Trust. Essentially the same relationships between certificate authorities in the hierarchical model may exist between the nodes in the cumulative model. The distinction is that PGP allows individuals to act as certificate authorities and individual certificates may be signed (issued) by more than one authority. These may not appear to be substantial differences but they allow for much more flexible trust models and more direct trust relationships. For instance, the hierarchical model does not scale well when implementing peer relationships. A group of 20 individuals would need 400 (20^2 - one for each individual relationship amongst the group members) X.509 certificates whereas only 20 PGP certificates could do the same. There is, however, a cost for this increased flexibility – certificate revocation is much more difficult under the cumulative model. This is very important where centralized certificate management is necessary.

Figure 14 - A Simplistic PGP Certificate Path illustrates a trusted chain where Sylvia signs Jeff's public key and Jeff signs Bob's key. This path can be used by Sylvia to validate Bob's certificate and therefore his public key. In the Hierarchical model a direct path like this would be found between certificate authorities and not individuals. Individuals in the cumulative model are acting as certificate authorities would in the hierarchical model. Note

this is a simplistic case even for the Cumulative model, which would likely be more convoluted with many more intermediate introducers⁹.

Figure 14 - A Simplistic PGP Certificate Path

3.5 Summary of Secure E-Mail

Secure e-mail began as a collection of established cryptosystems and enveloping protocols. Two of the four enveloping protocols have been deprecated leaving S/MIME and PGP as the contenders. These protocols are not interoperable with each other. Although they are merging in terms of offering very similar security services, there remain differences such as the trust models they are founded on. While the hierarchical model offers strict centralised control over trust, it is less flexible when it comes to practical, day-to-day business requirements. What remains to be seen is whether one of these two protocols will dominate the other or if each will endure to provide the same secure services but to two markedly different user groups. If both protocols endure, true secure interoperability will be a

⁹ Introducer is a PGP term used to describe individuals that act as certifiers by signing keys of individuals they know and trust. Others, who already trust the introducer, may then decide to trust newcomers based on the signature of the introducer (i.e. certificate authority role).

requirement at the user level. Although the following quote is over a year old, it continues to reflect the current state of secure e-mail over the Internet [5]:

"The continuing uncertainty between S/MIME and PGP suggests that an implementation should avoid painting itself into a corner by assuming a single winner. This means that a flexible, extensible software framework is vital ... This might be more general than strictly necessary but the uncertain future is more than a simple binary choice. Both PGP and S/MIME are being revised and each provides mechanisms for selecting among a variety of sub-choices. In effect an implementation should assume that it must support an arbitrary number of different security algorithms and multiple packaging schemes.

Developers faced with choosing a single standard should ask whether very near-term, general interoperability with deployed software from other vendors is desired or whether long-term interoperability is required.

The debate boils down to one simple question: Which method does the user trust more? Does he/she entrust an S/MIME third-party CA (certification authority) with the job of assigning and managing their digital certificates for them, or would they rather perform these operations in house with PGP? The bottom line is that both forms of trust are viable.

It is the author's conviction that there is a requirement for both models. Not only for the aforementioned reasons but also to support two large existing user groups - large corporations and small businesses/individuals. Note also that even with a diverse protocol base things are improving for e-mail software developers as the same encryption algorithms and MIME encapsulation formats are being used for many protocols

Chapter 4 CURRENT STATE OF SECURE E-MAIL INTEROPERABILITY

There are several issues associated with current secure technologies that complicate and frustrate the adoption of interoperable secure e-mail. Many of these have already been mentioned. From a business or developer's perspective there are currently significant disadvantages involving the overhead, management, practicality and competitive (dis)advantage of integrating secure technologies into commercial applications. From a user's point of view, it is generally unacceptable to use secure technology that is not interoperable.

Interoperability is important where secure services are required between systems that implement these services using different protocols, underlying security mechanisms and cryptographic algorithms. Four secure and incompatible enveloping protocols have been described in the previous chapter – two of which remain as serious contenders today. Two different market segments requiring the same security services but different trust models exist. The corporate or business market supports S/MIME with its centralised control and the hierarchical trust model. PGP has a loyal following of users interested in securing individual privacy rights and the simplicity of the cumulative or "web of trust" model.

Public Key Infrastructures (PKIs) are facing the same interoperability problem. Warwick Ford, the co-chair of the IETF PKIX¹⁰ group and a prominent, long standing member in information security standards and technologies, as recently as November 1999 stated [37]:

"The greatest remaining interoperability challenge lies not within the PKI, but at the interface between applications and infrastructure – the PKI enablement problem. We still lack a single standard interface whereby any application vendor can leverage PKI provided by any PKI vendor. However, such an interface and accompanying toolkits will soon emerge through natural selection from competitive offerings in the marketplace and/or vendor collaboration."

4.1 Security Service and Cryptographic APIs

An SSAPI together with a CAPI-based security infrastructure are the enabling technologies for practical, global, secure e-mail interoperability. For most applications the key is the high-level SSAPI characterised as cryptographically unaware and not one of the other highly aware or specific CAPI such as that used to interface a hardware cryptographic device. Multiple CAPIs are needed for different requirements within a given secure communications system. The role of the high-level SSAPI is to interface the general application to the security mechanisms used to provide secure services. The goals and advantages of this SSAPI are to:

¹⁰ PKIX is the IETF Public Key Infrastructure working group chartered to develop the standards required to support an X.509-based PKI. Their work is expanding to include protocols and other integrally related issues such as secure interoperability.

- a. provide a simple, easy-to-use way to make security services available to application developers without their needing to understand underlying complexities of cryptographic algorithms and protocols
- b. create a framework to identify and specify primitives required to support a comprehensive set of security services. This could be extended across product lines and vendors assisting in collaborative efforts between vendors and across industries
- c. facilitate interoperability and a means for parties to negotiate a mutually agreed upon secure communications environment (i.e. establish cryptographic algorithms, security services and mechanisms utilised)
- d. foster standardised, portable and reliable security by negating the need to have application developers recreate their own security software.

Few companies have adopted this approach stating that they have put much work into integrating proprietary toolkits into their applications, they are happy with the results and see no reason to change things, especially if an SSAPI may introduce a decrease in performance. These are good arguments; from a short-term perspective. However, the long-term supports SSAPIs/CAPIs as professed by the EC Company, a company in Silicon Valley, California that develops secure applications for electronic commerce. In what amounts to a company testimonial, the virtues of an open SSAPI and supporting CAPI-based security infrastructure are supported from first hand experience [38]. Even more telling is that fact that

the company initially resisted the idea in favour of a proprietary third party toolkit. The following quote is from their vice president:

"Having gained the perspective of having more time pass since we completed our initial implementation (the RSA toolkit), and having invested additional consideration into our overall security architecture, we now appreciate the benefits that an API approach offers. For instance, we have come to realize that a purely software-based PKI (as opposed to utilising smart card technologies) has some disadvantages... Before the availability of an API-based security capability, we would have had to write largely distinct functions to generate, retrieve, and manage keys stored in the file system vs. those stored on smart cards... With the advent of an API and the availability of products that adhere to it, we only need to write these functions once, regardless of the underlying storage technology. We can even let customers plug in the desired technology when they install our products, based on their own security concerns... Other opportunities that have become increasingly important to us are international markets (i.e. export control laws on cryptographic products) and business opportunities that want to deploy our solutions on non-Windows platforms (i.e. APIs also facilitate cross platform functionality)."

Utilising their experience as a case study the following points are clear from this first hand success story:

- a. The EC Company had initially selected and integrated the RSA BSafe toolkit into their secure applications. They selected the RSA toolkit because of the consumer confidence associated with the RSA name. The toolkit also provided a single interface (albeit proprietary) to the wide range of security services offered by RSA and, should the EC Company decide to do so, it was easy to select and use any of the cryptographic algorithms supported by the toolkit.
- b. An SSAPI went well beyond the RSA toolkit in providing the extensibility and modularity required for long-term competitive advantage in securing

applications yet it permitted the continued use of the RSA products themselves, should the company wish to do so.

There have been three organised efforts targeting the development of an Open SSAPI and/or the security infrastructure required to support interoperable secure e-mail. A description of each is given in the following sub-sections.

4.1.1 Common Data Security Architecture (CDSA)

CDSA, as its name suggests, is an entire architecture for secure services. It was developed in the Intel labs as a distinct and comprehensive security architecture from which a wide range of security services could be managed and provided to generic applications. Both session (connection) and store-and-forward (connectionless) communication services are accommodated. The API solution referred to by the EC Company was CDSA. However, the concepts and benefits mentioned in the EC Company example apply generally to any SSAPI/CAPI-based architecture. From a developer's perspective, it is important that products are standards-based. This maximises the potential to implement the product once and apply it to multiple solutions, such as extending it across multiple platforms. The EC Company was very conscious of this and highlighted the fact that CDSA is now an Open Group standard and publicly supported by a significant roster of key vendors within the industry. The message is that CDSA is a viable, well supported API-based architecture designed to support a higher-level SSAPI. Third party products already exist that both use and support CDSA with others

being developed. The following is a summary description of the CDSA based on version 2 of the specification [39].

CDSA is an open set of layered security services and associated APIs designed to meet the current and any future needs the industry may have. These services may be used to support a broad range of applications including electronic commerce, content distribution (software, information, entertainment) and personal applications such as e-mail and home banking. As an architecture, it also provides a framework to encourage the development of security standards and interoperability.

Figure 15 - The Common Data Security Architecture

Figure 15 - The Common Data Security Architecture illustrates this architecture.

The figure depicts four horizontal layers, the top layer being the actual applications that employ the CDSA provided security services; an MUA providing secure e-mail for instance. The second layer is called the Layered Services layer and is essentially an optional intermediary interface layer between the application and the Common Security Services Manager (CSSM) security API or third layer. Layered Services may be used to translate between different languages should the application be written in a language other than C, which is what the CSSM API is written in. Other functions that may be accomplished within the second layer are transparent security services, security management tools and the provision of high-level security services. Transparent security services are implemented by emulating a existing, insecure API to the application yet utilising the desired CDSA security services in the lower layers. Legacy applications can be secured in this manner without having to rewrite any legacy code. For instance, a legacy application that uses HTTP can be enhanced to use SSL to provide confidentiality services while using HTTP. Secure enhancements exploiting this option may be limited by previous logic and user interfaces, unless the legacy application itself is modified. Security management tools are applications specifically written to manipulate and configure the CDSA environment. From the perspective of this thesis, the most noteworthy role of the Layered Services layer is the provision of high-level security services: this layer hosts the SSAPI.

Applications may call the CSSM API directly or go through another high-level API in the second layer. Applications that call the CSSM API directly must be cryptographically

aware. In this case, application programmers must be aware of CDSA and cryptography basics to understand and avoid implementation-based security problems. Alternatively, applications that call another high-level API need not be cryptographically aware. They are not subject to the same potential security problems, yet gain all the benefits of CDSA. The CDSA specification does not actually provide this high-level API. It does, however, have a place for it in the architecture and it does provide a detailed specification into the third layer (the CSSM API) to which this high-level API must interface. Fortunately, there is an ideal high-level API that has already been developed for this role. This will be explained in the next section after the CDSA architecture is explained.

As already stated the third layer is well defined in the CDSA specification. CSSM is the heart of the CDSA. The CSSM API interfaces the higher layers into the security architecture. A series of lower layer APIs are also defined for the service providers (e.g. Service Provider Interface - SPI) or security modules to interface up into the CSSM layer. CSSM acts as the manager and integrator of application requests (CSSM API Calls) and security services (Service Provider responses). Security services are managed and may be dynamically extended by the CSSM which also contains the Trusted Computing Base (TCB) used to watch over and maintain the integrity of the environment. Independent software and hardware vendors may provide add-in security modules as competitive products. Applications use CSSM to direct their requests to modules from specific vendors or to any module that performs the required services. Applications can use multiple service providers of all types concurrently. Add-in modules augment the set of available security services.

CSSM functionality is itself divided into categories. Each category employs a module manager to co-ordinate relevant CSSM API calls from above and routes them to the appropriate service provider (add-in module) to generate the required service. There are currently eight categories within CSSM. New categories and module managers will be added as the need arises. The current functional subgroups or module managers are:

- a. Cryptographic Services Provider (CSP) Manager – controls the CSPs that may be installed on the local system, defines a common low-level or cryptographically aware API for accessing all of the CSPs that may be used by an application.
- b. Trust Policy (TP) Services/Module Manager – manages the trust policy modules that may be installed on the local system and defines a common API for access to these provider modules. Trust policies are the rules that define the credentials required for authorisation to perform an action on another object. For instance, being able to digitally sign a message on behalf of someone else in an organisation. Much of the activity in this area is based on using digital certificates.
- c. Authorisation Computation (AC) Services Manager – administers the Authorisation Computation Service Providers that may be installed on the local system and defines a common API for accessing Authorisation Computation Service Providers. The API defines a general authorisation

evaluation service that computes whether a set of credentials and samples are authorised to perform a specific operation on a specific object. Trust policies define the type of credentials required. Once it is established that the correct types of credentials are available, the AC libraries perform the authentication and authorisation services, using these credentials. These services combined provide access control functionality.

- d. Certificate Library (CL) Services/Module Manager – oversees the Certificate Libraries (modules) that may be installed on the local system and defines a common API for these libraries. Using this API, applications may create, sign, verify, and extract field values from certificates and certificate revocation lists.
- e. Data Storage Library (DL) Services Manager – defines an API for secure, persistent storage of certificates, certificate revocation lists (CRLs) and other security objects. The API allows applications to search and select stored data objects and to query meta-information about each data store (such as its name, date of last modification, size)
- f. Elective Module Manager – These module managers define additional APIs for new categories of service. This is the mechanism CSSM uses to provide the extensibility required to meet future security requirements. If an

application chooses to use an elective service API, CSSM extends the services available to that application by dynamically attaching the appropriate module manager and an add-in service module to the running CSSM.

- g. Integrity Services – these are the core services used by CSSM to provide a set of integrity services for the module managers, add-in modules, and applications to verify the integrity of themselves, and the integrity, identity, and use authorisations of other components in the CSSM environment. When developers provide CSSM add-in modules they must also provide the signature or authentication files for these modules. Integrity Services use these authentication files to verify the add-in modules as they are attached to CDSA. This prevents unauthorised or Trojan horse modules from impairing the security of the CDSA. Because application and add-in security service modules are dynamic components from a CDSA perspective, CSSM requires the use of a strong verification mechanism to screen all these components as they are added to the CSSM environment.
- h. Security Contexts – Managing and tracking the numerous parameters required for security services can quickly become an onerous task requiring a detailed knowledge of cryptographic algorithms and their supporting data structures, especially if there are multiple technologies and or algorithms available. CSSM provides security context services to assist applications in specifying and

managing the numerous parameters required for cryptographic operations.

CSSM assists by managing the data structures used to hold these parameters.

This enables applications to remain cryptographically unaware.

The fourth layer houses the Security Add-In Modules. This is where the security services are actually generated. The CSSM does not provide security services itself. CSSM is a security management layer whereas the add-in modules are the actual cryptographic modules or cryptographic software or hardware implementations. Similar to the categories in the third layer, the fourth layer is subdivided into a series of low-level APIs. There is an API for each of the functional categories defined in CSSM. Vendors are free to implement CDSA add-in modules however they wish with minimal restrictions. If a single module provides services across CDSA functional boundaries, the single module must be registered within each of the CDSA managers and comply with each of the SPIs that are relevant to the services provided by this add-in module.

The following is a summary of CDSA benefits subdivided into four categories:

General benefits:

- a. Standards-based
- b. Wide support by key players
- c. Modularity & Extensibility - can change algorithms or security mechanisms without major implementation issues.

Benefits to applications developers

- a. Reduces time to market for security-enabled applications
- b. Promotes interoperability among applications
- c. New product opportunities for platform toolkit developers: security modules and middleware for the second layer of the CDSA model
- d. Enables applications to be cryptographically unaware (if using IDUP or Layered Services). As a minimum it provides a single unchanging security interface (CSSM API) regardless of the underlying mechanisms and algorithms used.
- e. Provides access to variety of security technologies
- f. Accommodates specialisation, developers can focus on application vice security concerns and they do not have to “reinvent the security wheel”
- g. Greater return-on-investment as does not need security expertise, not liable for correct implementation of basic security services, reduced maintenance and development costs
- h. Helps solve export control restrictions and customer unique requirements as application can be developed without cryptographic modules. Customers can then provide their own CDSA compliant cryptographic service modules.

Benefits to Security Service Providers. In addition to those above:

- a. SPIs allow developers to implement to a standard that will be used by a large market (portable infrastructure-based solution for multiple platforms)

- b. Accommodates specialisation, developers can focus on security vice application interfaces and concerns
- c. Will encourage hardware integration of security mechanisms that will reduce software-based vulnerabilities, improve performance, reduce costs and further increase the use and presence of security technologies.

Benefits to End Users:

- a. Customers get new applications quicker with higher assurance that security is correctly implemented
- b. Consistent security across different environments
- c. IT Managers can deploy a consistent level of security in applications across heterogeneous environments
- d. Can manage security modules (add-in modules) separately
- e. Applications that interoperate and work well
- f. Architecture/infrastructure that will support future security technologies (extensibility)
- g. Trust policies can be established and enforced locally for business-specific (or company-specific) security policies

4.1.2 International Cryptography Experiment (ICE)

The International Cryptography Experiment is a U.S. government sponsored program with two primary goals. The first is to support a broad, informal alliance of international groups and governments to help resolve issues related to the balancing of internationally interoperable cryptographic security solutions with government imposed controls and restrictions on the use of cryptography. Activities in this area cover both technical and non-technical issues. The second goal of the ICE program is a technical effort to develop and demonstrate cryptographic components for the program. This aspect of the program is called the Defence Advanced Research Projects Agency (DARPA) ICE project. DARPA is the research and development agency for the US Department of Defence. The specific objectives of the DARPA ICE project are:

"... to develop and demonstrate modular, removable, replaceable cryptographically-based security components that are commercially available ... to test the notion that secure software applications can be separated from specific cryptographic security components such that users can employ a number of alternatives with minimal impact on the applications; and, conversely, that specific cryptographic implementations (such as hardware or software modules) can support a number of different applications. Such an approach should enable easier, widespread development and use of secure applications utilising cryptography, allow combinations of cryptographic security that support a range of protection levels (e.g., military, government, commercial, and multinational), and, hence, provide flexible, lowcost methods of cryptographically protecting sensitive information." [40]

Although DARPA is co-ordinating the research, the NSA is the central agency involved in the technical development and oversight of the cryptographic technologies. To assist the NSA, Network Associates Incorporated (NAI) has been contracted. NAI is the

commercial security conglomerate that bought out the original contractor, Trusted Information Systems (TIS). TIS had a history of NSA contracts. TIS was also the contractor that had previously developed the reference implementation of the MOSS protocol. This implementation was called TIS/MOSS.

Beginning in 1994, TIS developed a high-level security architecture based on layered security services integrated through a series of APIs. Figure 16 - DARPA ICE Project Architecture shows the architectural model. Demonstrations of a secure e-mail application were developed to validate the architecture and its feasibility. Over the next few years the architecture and demonstrations evolved with the commercial cryptographic technologies. Each year NSA produced a report summarising the current state of commercial cryptographic technologies and how they could best be used to advance the DARPA ICE project. TIS would then be tasked to enhance the demonstrations appropriately. The last publicly available report is from July 1997. Although the NSA reports appear to have ceased since TIS was acquired by NAI, work in this area continues in the NAI labs ¹¹.

¹¹ Personal conversations with Dr. Dave Balenson, NAI Labs

Figure 16 - DARPA ICE Project Architecture

The latest available results of this work are summarised in Figure 17 - ICE Project Recommended CAPI Suite. The NSA team found it was not possible to select a *single* API standard to achieve the goals of the ICE project. The multitude of applications and range of cryptographic service modules available could only be supported using a suite of SSAPI/CAPIs. Each of the APIs was chosen as the best of breed in its category based on the NSA selection criteria previously introduced and detailed in Appendix C. The following suite of APIs was recommended as a means to enable applications to be cryptographically unaware, give the user maximum flexibility in selecting cryptographic service modules and simplify the addition and modification of security services used by an application:

- a. IDUP-GSS-API. IDUP-GSS-API was chosen as the SSAPI or high-level security service API that will provide security services to most applications,

without the need for these applications to be cryptographically aware.

These applications will need a one-time modification to call the correct IDUP-GSS-API routines. Changing cryptographic modules or algorithms or any other form of security infrastructure enhancements will be accommodated on the other side of the IDUP-GSS-API negating the need to change the application for security reasons. This is the safest interface in that applications are restricted from manipulating cryptographic elements. Conversely, it does not allow enough control over the cryptography to administer the supporting cryptographic modules; that is role of CSSM API and lower level APIs.

- b. CSSM API. As previously explained, this is the core of the CDSA. Interfacing into this infrastructure provides a wide range of standards-based security services and vendor security products. The CSSM API requires developers to have some cryptographic knowledge. Fortunately, most application developers will write to the IDUP-GSS-API having no need to interface directly with the CSSM API. Only specialised security applications or tools will need to interface directly with the CSSM API or lower APIs. Developers writing these applications must have cryptographic expertise to correctly implement these tools and avoid implementation-based security vulnerabilities. The CSSM API was intended for these types of developers. On the other side of the CSSM API is the CSSM Service Provider Interface (SPI) which is the API that

developers writing cryptographic service modules will use to interface up into the CDSA and the applications it is supporting.

- c. CAPI. The Microsoft CAPI (MS CAPI) is an API that allows cryptographically aware applications to call on both hardware and software CSPs or cryptographic modules. The MS CAPI, as well as an RSA-based CSP, is shipped as part of the Windows 95 and Windows NT operating systems. The advantage of using this CAPI/CSP pair is that they are virtually ubiquitous. From a developer's perspective, this is particularly attractive as a means to quickly provide security services to clients. It is, however, a proprietary API and applications must be specifically tailored to this API in much the same way as toolkits are used for other proprietary APIs. Furthermore, the US government would not permit Microsoft to export this CAPI without a set of rigid controls. The operating system will not load a CSP unless it is cryptographically signed by Microsoft. Both the US government and Microsoft must first approve CSPs before they can be used (i.e. signed) via the MS CAPI. This will not pose a substantial problem for CSP developers targeting North American markets but it may have a significant impact on their use/re-use anywhere else in the world [41]. This is anticipated to be much easier now that the US government is relaxing its controls on cryptographic software. Note: The diagram shows a Fortezza CSP, as this is what TIS developed for the US government.

- d. Cryptoki. PKCS # 11, developed by RSA, has become the de facto hardware token interface. It is also known as the Crypto Token API or Cryptoki. This API interfaces directly with cryptographic tokens making it the logical place for user authentication functionality. Management of the token itself is also achieved via this API and the applicable software tools. A major component of the MISSI program is the Fortezza-based hardware token. The Cryptoki interface is required wherever hardware tokens will be used. In the case of the ICE project, it is required for MISSI compatibility.

Figure 17 - ICE Project Recommended CAPI Suite

Note that all three lower-layer APIs are not required as any one of these APIs may be used to service an application. These three were chosen to provide the full range and flexibility needed to support all types of MISSI applications. Cryptographic developers are free to implement to the API best suited for their particular functionality or target application.

This CAPI-based architecture has many benefits identical to those already mentioned for CDSA. This is partially due to the fact that CDSA is incorporated into this architecture but also because many of these benefits are related to API-based architectures in general. The following are additional general benefits of this architecture:

- a. Many current secure standards efforts incorporate functionality that would be better provided at another "security layer". With established APIs these standards efforts would be simplified and allow for specialisation at each layer. This is the same fundamental reasoning behind the OSI model and will become increasingly important with the sophistication of secure standards and services. Thus, GSSM-API builds the security support functions for IDUP-GSS-API and likewise; Cryptoki builds the token functions for GSSM-API.
- b. It helps reduce licensing fees by giving both developers and users the flexibility to use non-proprietary or freely available algorithms where and when they are available. This, in turn, encourages the use of secure applications

- c. It helps most applications to be cryptographically unaware
- d. Applications that utilise a standard SSAPI and CAPI can access multiple cryptographic implementations. This decreases implementation efforts.
- e. It provides an alternative supporting security infrastructure to CDSA, yet remains SSAPI/CAPI-based should this be required.

IDUP-GSS-API provides the safest interface between applications and security services. With respect to the other CAPIs, IDUP-GSS-API is also the most limited in its capability to manipulate the cryptography. Since the majority of applications will be cryptographic unaware, the recommendation is for applications to use IDUP-GSS-API. The two lower level CAPIs are used to support applications that need to be cryptographically aware or to build the underlying IDUP-GSS-API mechanisms.

Complimenting these theoretical results is the latest publicly available demonstration. Evolving from previous demonstrations in which secure e-mail applications were implemented using hard coded, protocol specific, UNIX implementations; this demonstration is an API-based, modular, cross platform (i.e. UNIX and Windows) secure e-mail application. Figure 18 - DARPA ICE Project Architecture, Conceptual Components and Figure 19 - DARPA ICE Project Secure E-Mail depict the architecture of this demonstration. It is a Windows Eudora

"plug-in" that provides integrity, confidentiality and authentication services for the Eudora e-mail application. The MOSS protocol is used to provide these security services. The plug-in is based on the TIS/MOSS implementation, the MS CAPI and three CSPs. Two of the CSPs provide hardware-based cryptographic services: the Fischer International Crypto SmartDisk and the NSA Fortezza Crypto card (for MISSI compatibility). The other CSP is a software token. In fact, it is the RSA BSAFE product simulating a hardware token by implementing the Cryptoki hardware API. The fourth CSP, the Microsoft CSP, has not yet been implemented. It is shown to illustrate that CSPs are not required to implement the Cryptoki API. They may interface directly through the MS CAPI but Microsoft must cryptographically sign them before the operating system will load the CSP. Future ICE demonstrations are expected to include an MS CAPI, non token-based CSP as well as the implementation of other secure enveloping protocols, the incorporation of SSAPIs and the CDSA.

Figure 18 - DARPA ICE Project Architecture, Conceptual Components

Figure 19 - DARPA ICE Project Secure E-Mail Demonstration Architecture

4.1.3 Internet Engineering Task Force (IETF)

The first two research efforts were broadly focused on developing or defining the API-based security infrastructure required to support multiple cryptographic technologies. The Common Authentication Technology Working Group of the IETF is credited with recognising that multiple secure enveloping protocols existed but a common generic high-level SSAPI to access these protocols was lacking. The problem was not the availability of a single protocol but rather the myriad of interfaces required to use these different security protocols and cryptographic modules. As stated by Carlisle Adams, the author of the IDUP specification [42]:

"The Independent Data Unit Protection (IDUP) is currently the only existing generic store-and-forward API stipulating the required security services. It was created specifically to fill this void."

The name given to this generic security services interface for e-mail is the Independent Data Unit Protocol Generic Security Service Application Program Interface (IDUP-GSS-API or just IDUP). The name is meant to reflect its close relationship with the GSS-API as well as the differences attributed to the store-and-forward communications for which it applies. "Independent Data Unit" is a reference to the general idea of individual data packets that characterise store-and-forward communications. An e-mail message can be thought of as an independent unit of data where the security services provided to it must be independent of any communications with the message destinations. They must also be independent of any other data units meaning that each e-mail message must be autonomous

with respect to its security services. All the required cryptographic control and data information must be incorporated into the message as a stand-alone unit of data.

The following text is taken from RFC 2479, the IDUP-GSS-API specification [28] to provide a short description of the goals, services and nature of this SSAPI:

"The IDUP-GSS-API extends the GSS-API [RFC-2078] for applications requiring protection of a generic data unit (such as a file or message) in a way which is independent of the protection of any other data unit and independent of any concurrent contact with designated "receivers" of the data unit. Thus, it is suitable for applications such as secure electronic mail where data needs to be protected without any on-line connection with the intended recipient(s) of that data. The protection offered by IDUP includes services such as data origin authentication with data integrity, data confidentiality with data integrity, and support for non-repudiation services. Subsequent to being protected, the data unit can be transferred to the recipient(s) - or to an archive - perhaps to be processed ("unprotected") only days or years later..."

As with RFC-2078, this IDUP-GSS-API definition provides security services to callers in a generic fashion, supportable with a range of underlying mechanisms and technologies and hence allowing source-level portability of applications to different environments. This specification defines IDUP-GSS-API services and primitives at a level independent of underlying mechanism and programming language environment, and is to be complemented by other, related specifications:

- *documents defining specific parameter bindings for particular language environments;*
- *documents defining token formats, protocols, and procedures to be implemented in order to realise IDUP-GSS-API services atop particular security mechanisms."*

The aim of the IDUP-GSS-API is to facilitate source level portability (of the MUA in our case) across security environments that use different mechanisms to provide security

services. The IDUP-GSS-API is a portable set of functions that allow application programmers to create secure applications without learning the particular elements of any of the various underlying security mechanisms available to them. Since it provides a generic interface through which security services are provided by the different cryptosystem implementations, the MUA programmer need only learn this single API and avoids having to invest the time, code and energy into keeping up with multiple cryptosystem implementations. From the developer's point of view, once the IDUP API is mastered and integrated into their products, it is essentially as easy to employ one secure enveloping protocol as it is multiple protocols.

For example, MD5 is commonly used between commercial organisations but is not endorsed by the US or Canadian governments. Eventually, most commercial organisations will also have a requirement to communicate securely and electronically with the government. Given the adoption of IDUP as the standard SSAPI, the commercial organisation would simply have to have a CSP module implementing the government approved SHA-1 algorithm available to the MUA to enable interoperable secure communications with the government.

Carlisle Adams first recognised the value of IDUP in separating MUA functionality from secure enveloping protocols as early as 1995 when he proposed the idea that IDUP could be used to enable MUAs to "understand" a number of secure enveloping protocols and therefore create "Universal Decoders" for Secure e-mail [43]. This thesis represents a step toward this realisation.

The IDUP API itself can be viewed as three interfaces: the General Purpose (GP), Signing and Encryption (SE) and the Evidence (EV) interface. The GP interface is the original, full service IDUP interface providing access to a powerful and sophisticated set of security services. The disadvantage of the GP interface is the complexity of the interface itself - especially if only a basic subset of security services is required. The SE and EV interfaces were developed to fill this niche. These calls provide a simpler interface to the most common security services for encryption, integrity and evidence generation. Using both the SE and EV interfaces, it is possible to use multiple SE/EV calls to provide the same functionality as the GP interface. The cost of this simplicity is performance and overhead, as the GP interface would only require one call.

To claim conformance with the IDUP specification, implementations must support the credential management calls, the environment-level calls, the support calls and some subset of the GP, SE or EV calls. The subset supported will depend upon the underlying mechanisms and will typically be the SE calls, the EV calls, or both. Support for the GP calls are encouraged but this is not required for conformance.

Vendors and clients that value formal security evaluation ratings will benefit greatly from having IDUP compliant applications. IDUP serves these applications particularly well by de-coupling the security evaluation and application lifecycles. When a developer decides to integrate third party security into a product it is risky in that both product reputations are dependent on the third party security software. Independent testing and certification of the

security product (i.e. product evaluation) assures both users and application developers that the underlying security on which they are depending, has indeed been implemented correctly. There is currently a synchronisation problem between these two lifecycles. When the non-secure functionality of an evaluated application is updated the overall assurance or evaluation rating of the application is invalidated, even if the underlying security had not changed. This is due to the fact that the application and security functionality are so tightly integrated. The evaluation was based specifically on both the old application and how it worked with the security code. Unfortunately, the subsequent security evaluation of the new updated application takes another year to catch up (if it does). If the evaluation rating of the security code could be decoupled from the rest of the application (i.e. user non-secure functionality), this evolution would be much smoother. The validated cryptographic modules could be then be re-used, regardless of the version of the application. Should the cryptographic modules themselves require changing, they could be replaced once they are re-validated.

Having a standard API not only provides the modularity to replace untested modules with those that have been certified; it also provides a clean and well defined boundary between the application and the security product. This helps to identify and correct any deficiencies that may be found in the certification process.

A team of security evaluators working in the Canadian Security Establishment emphasised the same benefits from firsthand experience [35]. A security flaw could originate from the security software itself or from the application in its use of security related parameters

or processing. They state that two fundamental principles of certification are clearly defining exactly what is being certified (the security target) and keeping this target as small and simple as possible. The goal of both these principles is to keep the certification within feasible and manageable bounds. In the case of a monolithic entity, where security and application software are so tightly integrated; it would be very difficult to identify clearly where secure functionality stopped and the application functionality began. Thus, a large portion, if not the entire application, would need to be certified. Clearly, this scenario is less desirable than having a well-defined boundary separating the secure domain from the application domain. Furthermore, once the security flaw is corrected and the security module is certified this certification can be transferred not only to the next version of the same application but to a completely different application utilising the same validated cryptographic modules. The security domain remains intact, as there is no need to modify it for integration with the application domain.

Developers originally criticised IDUP as being overly complicated for their needs. Circumstances have changed in both the business and technical arenas. The IDUP specification has since been modified to include the SE and EV, high-level set of procedure calls aimed at simplifying the interface for applications requiring only basic security services. Secondly, the strategic and business advantages resulting from the improved flexibility and modularity of using IDUP (an SSAPI) may well swing the pendulum the other way. To remain competitive developers need extensible, modular security architectures to provide the flexibility required in the current market place. IDUP has been developed over the last five

years to provide exactly such architectures. It is a solution to a number of advanced security issues, including secure interoperability. Perhaps IDUP debuted prematurely, in an industry that was just beginning to use basic security services and competing in a market that was unaware of the need and value of the advantages of an SSAPI.

It is ironic that today, in a multiple protocol environment, IDUP could be accepted to resolve many of the same issues for which it was originally rejected; back at a time when vendors simply had to pick a protocol to claim secure functionality of any kind. For instance, IDUP can decrease the time and cost of integrating (multiple) security protocols and it is relatively less complicated than understanding and implementing (multiple) unique toolkits and interfaces for each protocol implementation.

The benefits of using the IDUP SSAPI are:

- a. True interoperability which, in turn, supports a diversity of secure technologies and the previously mentioned benefits of this diversity without the current problems associated with supporting a diverse technology base
- b. It greatly facilitates the maintenance and configuration management of secure implementations for network administrators or security managers

- c. The consolidation and separation of existing MUA software into two broad domains: the secure domain responsible for the security of the system and the application or e-mail domain responsible for all other functionality. This advocates the specialisation of skill sets for each of these domains. IDUP provides the demarcation of these domains with the secure domain on one side of the API and the application on the other
- d. Isolates application developers from the variety of cryptographic libraries, hardware tokens, and software tokens, giving the user maximum flexibility in selecting cryptographic services for inclusion in an application.
- e. IDUP-GSS-API facilitates co-existence of both PGP and S/MIME (Private and corporate) protocols
- f. Frees software vendors from government export restrictions without restricting markets yet prevents cryptographic algorithms from being exported by the vendor.
- g. Decouples application and security evaluation lifecycles
- h. Together with an underlying infrastructure, such as that recommended by the ICE project, conciliates secure interoperability at the user level by providing a practical means to develop and maintain the required supporting infrastructure.

- i. Alleviates need for application developers to limit themselves to a specific proprietary toolkit and vendor as well as any strategic or business impact this decision may have *without preventing the developer from using these vendor products* at lower levels should it be commercially advantageous to do so. (i.e. using the RSA name)

4.2 Summary of Interoperable Secure E-Mail

When multiple security protocols are implemented without a SSAPI, the MUA must be customised for each of the individual mechanisms or protocol implementations. Different protocols will, of course, not communicate or interoperate. A fully interoperable secure MUA is currently required to handle multiple security enveloping protocols with S/MIME and PGP as a minimum. An API-based security infrastructure is also needed to support these protocols with the flexibility and modularity necessary for the commercial market place.

The IDUP-GSS-API is an SSAPI that was designed specifically to provide a high-level generic API between applications and security mechanisms. This SSAPI, combined with an API-based supporting security infrastructure, can provide secure interoperable e-mail over the Internet.

Chapter 5 THE CRYPTOGRAPHIC INTEROPERABILITY PROBLEM

To reach the next quantum level in its evolution, the Internet must gain the trust of the general public. Secure communications are needed now to achieve this trust and the acceptance of the general public. This would provide not only a substantial user base in terms of casual or non-business related activities but more importantly, it would provide businesses of almost every nature with the justification to fully endorse the Internet. Once this momentum is achieved, Internet use will become as common as the telephone.

The problem is how to provide this security across different cryptographic protocols. This is not a new problem. In 1996, EDIINT stated [19]:

"The predominant problem with the use of the Internet for EDI is inter-operability between Vendor products, specifically in the areas of integrity, confidentiality, digital signatures and non-repudiation."

This is no longer a technology problem as this thesis illustrates. It is a business problem. These technologies are now, for the first time in their history, being developed in the commercial sector. As such they are subject to commercial stresses. Until recently, commercial application developers wishing to secure their products were forced to strategically

commit to proprietary or specific security solutions excluding other options and any future strategic options. Companies are understandably reluctant to take such risks, especially in light of the many incompatible security technologies. Moreover, security technologies are constantly changing to remain secure against increasingly sophisticated and powerful cryptanalysis techniques. Furthermore, US government legislation aimed at controlling the distribution of cryptographic technologies continues to represent a significant unpredictable business risk. As previously noted, this legislation was relaxed in September 1999. The long-term nature of these changes has yet to be seen. It has only been since cryptographic technology has become commercially viable that these long-standing restrictions conflicted with the use (sale) of these technologies globally. The influence of the US government in the commercial cryptographic industry is changing. It should not, however, be ignored and can change again. This thesis cannot solve political or legislative conflicts. It can, however, show how the industry can move forward in a way that will help resolve many of these business concerns and even provide a competitive advantage using existing technologies.

Problem Statement

It is possible to provide secure e-mail over the Internet that is both interoperable and practical using existing technologies.

Interoperable secure e-mail in this context is defined as an e-mail service that will give users the ability to communicate securely to other users regardless of the secure protocol

employed by the other user. We have established that secure protocols by their nature are not interoperable. Once a message is protected using a given protocol, it must be unprotected using the same protocol. We have also shown that, contrary to commercial pressures, it is desirable to have more than a single protocol and the ability to communicate across these protocols. Interoperability across these protocols will require that users have the ability to use these different security protocols. This must be accomplished in a way that is transparent to the user. Hence, secure interoperability is meant to mean the apparent interoperability across protocols *from the user's perspective*. That is to say that within a given set of standard secure protocols the user will simply request secure services and process the e-mail message in a routine fashion regardless of the secure protocol required.

To be practical, security solutions must resolve problems for both the industry and the user. Users require highly technical solutions that will allow them to communicate securely to users without restrictions on either party. These solutions must minimise the need for user involvement and training. They must be flexible enough to be easily changed or modified when the need arises without a negative impact. They must be widely available and inexpensive.

From an industry perspective, the same solutions must also be reusable. Unlike past practices where MUAs were custom tailored for each protocol, developers must be able to code to a single interface that will provide all the required security services they need. This will dramatically decrease development and maintenance costs due to security. It will also greatly

reduce the risk to the company and return a significant element of strategic control to the application developer's business managers by eliminating the need to choose a specific security technology. Similarly, developers cannot be expected to become security experts and vice versa. Application and security functionality must be isolated and developed by experts within each speciality. Yet the resulting products must interface closely to correctly serve the user. Solutions must thus enable this separation and specialisation of development skills. As with most technical developments the solutions must also be standardised. Standardisation has many benefits, such as interoperability, and it enables the growth of the commercial infrastructure necessary for support of the technology. For example and as previously stated, survival in the commercial marketplace is dependent on the ability of vendors to partner with other developers to provide innovative products based on the combined competencies of both. The ability to successfully integrate products and organisational infrastructures is much easier and efficient in a standards-based environment.

Open APIs solve many of these requirements. The IDUP-GSS-API was specifically designed as an SSAPI to provide a common high-level interface for different security protocols and cryptographic mechanisms. Lower-level CAPIs also exist providing the modularity and flexibility necessary for the practical requirements of commercially developed security. These CAPIs have helped reduce development costs, enabled "plug and play" cryptographic modules and resolved some aspects of the conflict between government cryptographic export control and commercial interests in foreign markets. Standardisation

work and support for these CAPIs is also progressing quickly to establish an API-based, standardised security infrastructure as in the case of CDSA.

IDUP is the only existing SSAPI and from that perspective is the current state of interoperable secure e-mail from an application developer's perspective and within the context of this thesis. IDUP provides security services such as data origin authentication, data integrity, confidentiality and non-repudiation in a generic fashion that can be supported with a range of underlying secure enveloping protocols and cryptographic technologies.

Surprisingly, the IETF efforts and IDUP have not yet been met with much enthusiasm from the industry. This is likely due to the popularity of a single industry-wide protocol solution and the marketing efforts of RSA in favour of the S/MIME protocol.

This thesis supports the notion that multiple protocols are not only desirable but also essential in the long term. The unique nature and role of e-mail security distinguishes this issue from most others before it. S/MIME and PGP may be virtually identical in the technical security services they offer. They remain equal yet opposing contenders because they each have a large disparate user base. For the most part, these user bases themselves are also incompatible in that the human opinion, belief and trust invested into these protocols over the years are not transferable to the other camp. In many cases there seems to be an active distrust of the technology and motivations within the other camp. The capability to generically utilise security services supported with a range of underlying secure enveloping protocols and

cryptographic technologies would allow both camps to communicate with each other as well as others that may mandate another protocol such as the US Government and the Fortezza Cryptographic PC Card. It also reduces the negative impact should an error or vulnerability be found in any given protocol and thus need to be replaced.

Chapter 6 AN INTEROPERABLE SECURE E-MAIL PROTOTYPE

This chapter describes a generic MUA architecture that portrays the design currently used in secure MUAs. The model is then modified to illustrate an architecture that will provide users with application level secure interoperable e-mail over the Internet as well as the benefits of an API-based security architecture. This represents a step beyond the published literature and the current state of secure e-mail described in the previous chapters. A working prototype, based on this design, is then presented.

6.1 Architecture Model

Figure 20 - Current Architecture Model depicts the model of an interoperable MUA using current architectural designs. As there is no standard SSAPI, proprietary or custom coded interfaces are used for each protocol. It is assumed that all insecure e-mail utilises the MIME protocol and is labelled protocol "A". Secure mail, however, will need to be processed using one of the three possible protocols B, C or D corresponding to PGP, S/MIME and a third unnamed protocol. The point is that there could be a need for some form of interoperability beyond PGP and S/MIME and the model must reflect this functionality. Each of the architectural components shown in the model are described below:

Non-secure e-mail using MIME (A)
 Secure e-mail using different
 enveloping protocols (B, C and D)

Figure 20 - Current Architecture Model

- a. MUA Code. The Mail User Agent (MUA) is the e-mail program the user employs to provide e-mail services. At the front end of the MUA is the human or user interface. At the back end, the MUA interfaces with transport protocols to transmit and receive e-mail messages over the Internet. The MUA code represents all the code providing mailer functionality that is not directly related to security.

- b. MUA Security Interface Code (MSIC). The MSIC is the code in the MUA required to employ the security services of a particular secure enveloping protocol. These are proprietary third party interfaces or hard coded programming logic that integrate the mailer application with each particular secure enveloping protocol. Each one is illustrated as having a different size to highlight their individual characteristics.

- c. Enveloping Protocol Code. Each of the secure enveloping protocols: PGP, S/MIME and one other have their own implementations to provide secure services. It is quite likely that multiple enveloping protocols would require one or more common underlying cryptosystems. For example, all the protocols could support DES as an encryption algorithm. Unfortunately, it is also highly likely that each protocol implementation would integrate this DES implementation within the protocol code, especially if the protocols were appended to the MUA individually as opposed to being part of a cryptographic "tool kit". In the worse case, identical functionality (i.e. DES code) would be implemented in triplicate - a copy integrated within each of the protocol implementations. Each implementation is illustrated as having a different size and colour to highlight their independence.

6.2 Interoperable Architecture Model

Using the components described below, the architecture in Figure 21 - IDUP-based Architecture can be constructed:

Figure 21 - IDUP-based Architecture

- a. MUA IDUP Interface Code. Each MUA must be modified to make the correct IDUP calls. Once these MUAs are IDUP-based, there is no more need to modify the MUA for security reasons, regardless of the protocol used. This contrasts sharply with the current situation where the MUA and its

custom coded; dedicated protocol interfaces must be modified every time any change is made to any protocol.

- b. MPIM. This is the actual code that provides the IDUP functionality and, in effect, integrates all the secure enveloping protocol implementations below it. The MPIM responds to the IDUP API calls by performing any required pre-processing and then passing the object to the correct secure enveloping protocol to protect or unprotect the message. Once this secure processing is finished, the object is handed back to the MUA, through the IDUP API, to be handled in the same manner as any other MIME object. Note that much of the overhead in using these protocols has not magically disappeared. It has, however, moved from the MUA side of the IDUP API into the secure side of the API. This security related code can now be treated as an entity distinct from the MUA itself; in the same way as the operating system and its functionality is thought to be distinct from the MUA.
- c. Shared Cryptographic Engine. This represents a suite of cryptographic algorithm implementations to support the secure enveloping protocols. Prior to CAPI-based infrastructures, each developer would integrate the cryptographic algorithms into their particular protocol implementation.

Once implemented, this architecture will allow e-mail users to communicate securely with all others, regardless of their particular security implementations, with little or no impact at the users' level. Users will not have to be concerned with security matters related to the different protocols. Furthermore, users will not be restricted to communicating securely only with those utilising a single security implementation and/or protocol, as is currently the case. The proposed architecture simply requires users to request secure or non-secure communications and the message will be delivered, regardless of the particular security implementations utilised at either sending or receiving environments. In other words, we will have achieved transparency across secure enveloping protocols or secure e-mail interoperability at the application level.

6.3 IDUP Prototype

The purpose of creating a secure interoperable MUA was to demonstrate the following:

- a. IDUP was ideally suited for this purpose
- b. It is possible to provide secure interoperable e-mail without a significant, chronic development effort for the MUA vendors
- c. Users and MUA developers need not be knowledgeable in cryptographic principles

regardless of the secure protocol and technologies deployed behind the IDUP API

- d. How IDUP facilitates secure interoperability
- e. Once an MUA is IDUP compliant, major changes in security services (i.e. adding an entirely new security enveloping protocol) may be achieved with only slight modifications to the MUA. The original prototype code used in this thesis was written specifically for the S/MIME protocol. A more generic MUA could be written to minimise even further any required modifications to the MUA.
- f. How IDUP can segregate the application and security functions. Code for MUA functionality is distinct from that within the IDUP mechanism. This fosters specialisation where MUA or application developer skills may be focused on the application and, perhaps more importantly, security specialists can design, analyse and implement reusable cryptographic modules without application dependant constraints or distractions.
- g. A single interface can support multiple services and protocols

The first initiative to develop such a prototype proved to be infeasible and was abandoned after significant effort. The plan was first to build a multi-protocol IDUP

mechanism and then integrate this mechanism into an MUA through the IDUP API. At the time, PGP and the TIS/MOSS security protocol reference implementation were the only two implementations available from source code. TIS/MOSS was written for and within a UNIX environment. The development and target platforms for the IDUP prototype were DOS and/or Windows. The TIS/MOSS implementation was almost completely ported over to DOS when this prototype was abandoned.

Similarly, the PINE MUA was selected, as the source code was freely available from the University of Washington. The plan was to customise PINE to provide secure e-mail via the IDUP API and the PGP/MOSS IDUP mechanism constructed in phase one.

Recall the mandate of the ICE project, described in Chapter Four, is to develop and demonstrate cryptographic solutions that can be utilised in a wide range of secure applications. The ICE project has recently developed a single protocol IDUP-based mechanism designed in and for the Windows environment. This demonstration has not yet been made publicly available. Fortunately, the objectives of this thesis were recognised to be closely related to those of the ICE project and the source code for the IDUP-based demonstration was released to the author for educational and research purposes. The author would like to thank and recognise Dr. David Balenson and the Cryptographic Technologies Group at Network Associates Incorporated (NAI) Labs for releasing this source code and permission to extend this demonstration.

At this stage of the ICE project, the goal of this implementation was to use IDUP and demonstrate its strength in terms of modularity, reusability, application and communications protocol independence as well as the ability to generically interface to the underlying cryptographic mechanisms and service modules. All of these benefits offer direct support to the objectives of the ICE project. Application level secure interoperability or secure protocol transparency is not a direct goal of the ICE project and was not an objective of this original demonstration.

For the purposes of this thesis however, this same ICE demonstration has been extended and a prototype implementation created to validate the use of IDUP to provide practical multiple secure enveloping protocol support to applications. IDUP enables secure interoperability at the application level through the use of IDUP mechanisms that support multiple secure enveloping protocols. All of the advantages previously mentioned and for which this demonstration was originally implemented remain valid but in the broader context of a multi-protocol environment.

Excerpts from the original ICE demonstration User's Guide are used to briefly describe this implementation. Only those aspects germane to the interoperable prototype will be summarised. For example, the NAI demonstration includes a Fortezza CSP that is interfaced through the MS CAPI. This CSP is not described. Following this outline is a description of the modifications carried out to extend this demonstration and implement the secure interoperable prototype.

6.3.1 Overview of IDUP-S/MIME Plug-in

The IDUP-S/MIME Plug-in provides security services based on the S/MIME protocol. The plug-in is written specifically for the MUA or electronic mail program entitled Eudora, versions 3.x and 4.x. These services are accessed through the IDUP API and provided through an IDUP mechanism. The IDUP mechanism utilises the MS CAPI to gain access to the RSA CSP provided with MS Windows. The services provided include signature generation, message encryption and evidence generation, request, response, and verification. After the plug-in is installed, IDUP services will be available in Eudora, using button clicks and pull-down menus. The plug-in enables Eudora to both send/encode and receive/decode messages secured with the S/MIME protocol.

The IDUP mechanism provides six different secure message types to choose from:

- a. Clear Sign
- b. Encrypt
- c. Sign and Encrypt
- d. Encrypt and Sign
- e. Sign with Evidence (Generation or Request)
- f. Sign with Evidence and Encrypt (Generation or Request)

Using the plug-in with the Microsoft supplied RSA CSP allows the user to choose either MD2 or MD5 for a message hash algorithm, and either RC2 or DES for a message

encryption algorithm. Using the plug-in with the Fortezza CSP allows the user to use SHA-1 as the message hash algorithm and Skipjack as the message encryption algorithm. Depending on the type of certificate database the user has, the plug-in automatically selects the signature and key exchange algorithms as illustrated in Table 6 - CSP Algorithms.

	MS Base	NAI Labs Fortezza
Message Encryption	RC2 or DES	Skipjack
Message Hash	MD2 or MD5	SHA-1
Signature	RSA	DSA
Key Exchange	RSA	KEA

Table 6 - CSP Algorithms

6.3.2 IDUP-S/MIME Plug-in Architecture

As can be seen from Figure 22 - Plug-in Architecture, the plug-in consists of a body that implements the Extended Message Services (EMS)-API for Eudora. The body calls the IDUP mechanism via the IDUP SE/EV or GP interfaces. The SE/EV interface is simpler but may require multiple calls to provide all the requested services. Using the GP interface, only one call is required but it is a more complicated interface. The GP interface must be used for the Sign with Evidence and Encrypt service, as the EMS-API prevents the IDUP mechanism from being able to make the multiple calls it needs using the SE/EV interface. The IDUP mechanism uses S/MIME to format the messages, which in turn calls the RSA CSP for supporting cryptographic services. Once the message is protected it is returned to Eudora. Eudora then sends the messages via standard SMTP services.

Figure 22 - Plug-in Architecture

The EMS-API was developed by Qualcomm to provide a means for third party developers to interface with Eudora. Plug-ins (i.e. DLLs) can then be developed to extend the functionality of Eudora and the user may then install them independently. By exploiting the EMS-API, in lieu of releasing the source code, Qualcomm benefits from having third party products and support without exposing or threatening the integrity of the MUA itself. The company also preserves any commercial or proprietary interests in the MUA itself.

As security mechanisms and CSPs are often implemented as DLLs, it is important to note that the IDUP-S/MIME DLL is part MUA and part security mechanism. The IDUP-S/MIME DLL described above has been physically implemented as a file named Psuedosn.dll. In building this DLL, it was linked with a library called idup.lib. The source code is segregated

into Psuedosn or MUA source code files making up the plug-in body as depicted in Figure 22 and the IDUP mechanism files making up the IDUP library. The IDUP API is what separates the MUA from the IDUP mechanism programmatically. Thus, modifying the Psuedosn source code is the equivalent to modifying the MUA (in this case an extension of Eudora functionality) while changes to the IDUP library affect the IDUP mechanism itself. Both the RSA CSP as well as the MS CAPI are yet other DLLs entitled rsabase.dll and crypt32.dll respectively.

6.3.3 IDUP-S/MIME Plug-in Use

The Eudora application (MUA) is coded to check certain directories for the presence of plug-ins or DLLs written to the EMS-API. Finding Psuedosn.dll, Eudora displays two icons in the composition window (i.e. whenever the user is creating or editing a message). Should the user wish to protect this message, there is a choice of two icons. One of the icons is selected for the simple signing, encryption and evidence IDUP calls and the other is selected should the general-purpose IDUP interface be required. With one of these icons selected and the message ready for transmission, the user then selects the "send" button. In processing the message for transmission, Eudora calls the Psuedosn.dll; which presents the user with another Graphical User Interface (GUI) panel or "Service Dialogue" box as shown in Figure 23 - Plug-in User Interface. At this point the user must select specific security services and any related options.

Only those services applicable are offered. For example, if the sign service was selected, then the only choices made available would be which signature algorithm to use: MD2 or MD5. All other, non-applicable choices would be dimmed as the SHA-1, DES and Skipjack choices are shown in Figure 23. As previously noted, these services are unavailable as the FORTEZZA CSP, which was not used in this prototype, provides them.

Figure 23 - Plug-in User Interface

Once the user makes this selection and clicks on the "OK" button, the plug-in continues to secure the message as requested by the user. If the plug-in detects an error in

securing the message, an error message is displayed for the user and further processing of the message is suspended. The user is then required to acknowledge this message by clicking the OK button on the alert message box. Eudora will then abort the transmission of the message. Otherwise, the protected message is sent.

6.3.4 IDUP-S/MIME Plug-in Mechanism

The IDUP S/MIME mechanism is code within the idup.lib files that assembles various components into an S/MIME formatted message. Example functions are those that encode ciphertext output from the RSA CSP into a base64 mime body part. Other functions add the required MIME format lines such as "Content-Type: application/pkcs7-signature" and "Content-Transfer-Encoding: base64".

6.3.5 Overview of Prototype Plug-in

The multi-protocol prototype is not a commercially releasable IDUP mechanism. It is a demonstration implementation to illustrate the goals stated at the start of this chapter. The PGP CSP used by this prototype is not integrated programmatically with the plug-in. The PGP Software Development Kit (SDK) or Tool Kit that enables an application to interface with the PGP CSP through its proprietary API was not available. Note that with respect to the objectives of this prototype, there is a significant difference in utilising a proprietary API between the IDUP mechanism and its CSP as opposed to between the MUA and its security

module or provider. Similarly, for the purpose of this prototype, there would have been no advantage to incorporating the PGP CSP code into the IDUP mechanism itself. Alternatively, the command line interface for PGP version 6.5.1 was used. Thus, where the MS CAPI is used to integrate the RSA cryptographic services for the original S/MIME mechanism, the PGP command line is used to integrate PGP cryptographic services.

Other liberties were also taken where they could minimise the programming effort without detracting from the goal of the prototype. Again, the idea was not to develop a commercial product but simply to demonstrate the feasibility and advantages of doing so using the IDUP API. For example, in the prototype, the icon originally used to select the IDUP GP services is now used to select PGP-based cryptographic services. The disadvantage of this approach is that the Sign with Evidence and Encrypt (Generation or Request) S/MIME service is no longer available in the prototype. This service requires the full IDUP GP calls. All the other services described were available through either the SE/EV or GP IDUP interface and therefore remain accessible through SE/EV interface calls. The code implementing the GP calls has been disabled and replaced with the code implementing the PGP-based functionality. Similarly, the PGP functionality only provides clear signing of e-mail messages, as this was sufficient to demonstrate how these PGP-based services can be provided through the same IDUP API and mechanism.

6.3.6 Prototype Architecture

Figure 24 - Modified Plug-in Architecture illustrates the prototype architecture. It is essentially the same architecture as the S/MIME Plug-in with the GP IDUP interface removed, some additional code in both the plug-in body and IDUP mechanism to include the PGP protocol, and the addition of the PGP CSP. The plug-in body is still using the IDUP interface to call for security services from either protocol.

Figure 24 - Modified Plug-in Architecture

6.3.7 Prototype IDUP PGP Mechanism/CSP

PGP Freeware version 6.5.1 was used as a CSP within this prototype. In actual fact, this freeware implementation provides the services of a complete PGP mechanism. In other words, this "CSP" not only provides the cryptographic services used in PGP, it also provides PGP/MIME formatting or enveloping services. The IDUP mechanism and not the RSA CSP provide the equivalent S/MIME formatting services. This functional deviation highlights the ability of IDUP to isolate the application (MUA) from the complexities of its underlying security services provider. Because the command line interface is used to integrate the PGP CSP, PGP Freeware version 6.5.1 and its command line interface option must be installed with Eudora and the S/MIME plug-in for this prototype to function. Moreover, a PGP key pair must be created to be used in signing messages with the PGP/MIME protocol and the PGP configuration file must have the following parameters set to ensure the output of PGP is a clear signed message:

- a. `ARMOUR = on` # Use ASCII armour (for 8 bit signature values)
- b. `TEXTMODE = on` # Tells PGP to convert message text to canonical text
- c. `CLEARSIG = on` # Suppress armour and compression of message text
- d. `PGP_MIME = on` # Use to specify compatibility with PGP-MIME.
- e. `PGP_MIMEPARSE = on` # Use to instruct PGP to try to parse MIME body parts.

6.3.8 Prototype Plug-in Use

Use of the prototype is identical to that of the original IDUP S/MIME Plug-in with the following exceptions:

- a. the GP icon is now labelled as "PGP"
- b. the only PGP-based security service implemented is clear signing, regardless of the fact that other options on the "Service Dialogue" box show as if they were available
- c. when PGP signing is selected, an additional window is presented to the user very soon after the "OK" button is selected in the "Service Dialogue" box. The user is required to enter the PGP passphrase to gain access to the PGP private key for signing the message. This is standard PGP functionality, now incorporated into the plug-in. The user simply enters the passphrase and PGP will complete the signing process.

6.3.9 Source Code Modifications

The changes made to the original IDUP S/MIME Plug-in source code are listed in Appendix G. The next few paragraphs give a textual summary of these changes.

Plug-in Body

Files `SignTran.cpp` and `smime.cpp` were modified to incorporate PGP into the TIS (NAI) IDUP S/MIME Plug-in. A new function, entitled `pgp_protect_message`, is created paralleling the original `smime_protect_message` function. The NAI plug-in was intended to be a dedicated (single protocol) IDUP mechanism. This is reflected in its design and implementation. The purpose of the `pgp_protect_message` function is to replace S/MIME specific code with the equivalent PGP specific logic.

IDUP Mechanism

Five source code files were modified to incorporate clear signing using the PGP/MIME protocol. These five files were:

- a. `idup_abolish_env_chafe.c`
- b. `idup_establish_env_chafe.c`
- c. `idup_se_singlebuffer_protect_chafe.c`
- d. `smime_sign_chafe.c`, and
- e. the header file `mech.h`

idup_abolish_env_chafe.c

The MUA calls `idup_abolish_env` to abolish an environment; regardless of the protocol used. Working beneath the IDUP API, this function checks which protocol is being used and then calls the appropriate function to abolish the environment. The file also contains definitions for CSP (protocol) specific abolish functions. A new function, `pgpapi_abolish_env`, was created and is included here.

idup_establish_env_chafe.c

The MUA calls `idup_establish_env` to establish an environment; regardless of the protocol used. This function checks minimal requirements (e.g. must have credential handle or user name), sets the requested policy (policies are the rules used in protecting messages - i.e. CSP specific) to the default value unless otherwise specified, then checks this value and sets a number of CSP specific variables within the environment structure. A policy identifier and the appropriate variables were created for the PGP CSP. The environmental structure was also modified to accommodate PGP specific function pointers. The pointers are set here but called later on in the message protection processing.

idup_se_singlebuffer_protect_chafe.c

The `idup_se_singlebuffer_protect` function is called to apply the SE (Sign and Encrypt) IDUP services to a message; regardless of the protocol used. These services are formally titled confidentiality with integrity (CONF) and data origin authentication with integrity (DOA).

This function checks minimal requirements (e.g. must have an environment handle), extracts the signature and encryption algorithm identifiers from the environment structure and then calls the appropriate function according to the service requested. In calling these functions, the correct signature and/or encryption algorithm is specified by passing the identifiers previously assigned to the environment (extracted from the environment structure). Note also that this function depends upon another calling structure within the environment structure that must be correctly initialised before this function is called. The structure entry (function pointer) corresponding to message signing will be called whenever these services are required. The exact function that is executed is unknown to `idup_se_singlebuffer_protect`. The initialisation of this calling structure, including algorithm identifiers and function pointers, is done while establishing the IDUP environment.

To enable the PGP-based clear signature service, the calling structure was augmented to include new entries for the PGP routines and a check was added to determine which protocol to use before calling the PGP or S/MIME routines.

smime_sign_chafe.c

The `smime_sign_chafe.c` file contains the definitions for the IDUP functions that call the CSP functions to begin cryptographic operations. The original mechanism used MSCAPI to access both RSA and Fortezza libraries (CSPs). Thus, there was only one MSCAPI call in this function. The multi-protocol prototype uses two CSPs: MS CAPI and the PGP command line (i.e. DOS command line). A second function was created to work within the IDUP API paradigm, extracting any required parameters from the IDUP structures and then invoking the PGP clear signing service on the file containing the message that Eudora is processing.

mech.h

The IDUP mechanism header file, `mech.h`, contains declarations for many of the functions, variables and structures used by the mechanism. One of the key structures declared here is the environment structure (`smime_cred_id_t`). This structure was modified to include two additional function pointers for the PGP routines.

6.3.10 Summary of Prototype Plug-in

This prototype demonstrates how multiple protocols and cryptosystems can be available to MUAs using a single IDUP mechanism. Furthermore, the application developer does not need to be aware of the underlying cryptographic technologies or principles on the

other side of the IDUP API. This has significant advantages from a secure interoperability perspective:

- a. the MUA, Eudora in this case, need only use the IDUP interface to access all the services offered by the IDUP mechanism. In this prototype Eudora may use either the PGP or S/MIME secure enveloping protocol implemented by the IDUP mechanism through the same IDUP API. Other protocols or CSPs could also be added. Thus, by utilising a multi-protocol IDUP mechanism, an application can be multi-protocol enabled using a single interface. The IDUP API calls were implemented once and from that point on the security services offered to the application are a function of the DLL acting as the IDUP mechanism for the application.
- b. it allows application developers to offer secure products without the concerns of cryptographic export restrictions
- c. clients can now develop or choose and "implement" their own cryptographic services by deploying the desired IDUP mechanisms (i.e. DLLs) together with IDUP compliant applications.

Other advantages are:

- a. Very little change was required to the MUA. An entirely different security protocol was added to the mechanism with very minor changes to the MUA (i.e. code within the plug-in body). Had the mechanism been written more generically and not specifically for the S/MIME protocol, these changes would have been further minimised and integration made easier still.
- b. Modularity. The architecture of this secure e-mail application is composed of integrated, yet separately manageable components. For example, the main body of the MUA, the body of plug-in, the IDUP mechanism, the MSCAPI API and the CSPs are all individual modules. These components are maintained by separate organisations having different expertise. Respectively they are Qualcomm, NAI, the author, Microsoft, RSA and PGP.
- c. Flexibility. To replace or upgrade the security of this prototype, the user can simply replace the appropriate DLL without affecting the remainder of the application. This is a marked contrast from current architectures. Illustrating this, the original IDUP-S/MIME DLL was replaced with the prototype DLL and Eudora changed from an S/MIME based secure mail agent to a mail agent capable of secure messaging using either the S/MIME or PGP protocols.

- d. Preservation of Security Evaluation Ratings. The reverse is also true and especially valuable to evaluated products. In this scenario it may be possible for a product to maintain its security evaluation, by keeping the previously evaluated security modules and replacing or migrating the application on top of the IDUP API. Thus, the application or non-secure functionality may be upgraded without negatively impacting the evaluation rating of the whole.

Chapter 7 CONCLUSIONS AND RECOMMENDATIONS

7.1 Conclusions

The Internet is here to stay. But will it evolve to its full potential to become as common as the telephone? At this point in time, one of the most important factors influencing the Internet's widespread acceptance by the general public is communications security.

Secure communications must provide confidentiality, data integrity, authentication, and non-repudiation services. User requirements demand that these secure services have a minimum impact on the user. They must be as transparent as possible and interoperable in that they must not limit users to a single protocol; as there currently are different protocols, cryptosystems and implementations in use throughout the Internet. Providing secure services comes at the cost of interoperability *at the cryptosystem level*. User requirements can still be met by providing what appear to be interoperable security services - *at the application level*. To gain the necessary support in a commercial development environment, the solution must also be practical from both the user and industry perspectives.

E-mail is probably the “lowest common denominator” or one of the most common Internet applications used by the general public. For this reason it holds a key role in achieving widespread public acceptance of the Internet. Acting as a de facto ambassador for the future of the Internet, it is important that e-mail meets these user requirements. Furthermore, e-mail can also be used as the “communications subsystem” for larger systems. For example, a client-server database application can query remote databases through the use of e-mail messaging.

This thesis has presented an API-based Mail User Agent architecture based on existing specifications and standards. Secure interoperable e-mail over the Internet is possible today using this architecture. A secure interoperable prototype implementation was developed to verify this claim.

Having stated this, secure interoperability is more than technology. The reality is that industry or business concerns have a significant, even overriding, influence on realising secure interoperable e-mail over the Internet. Surprisingly, the IETF efforts and IDUP have not been met with much enthusiasm from the industry probably due to the popularity of a single protocol solution using the S/MIME protocol as well as the significant influence and marketing efforts of S/MIME proponents.

In the author’s opinion multiple security protocols supported through the IDUP-GSS-API and an API-based architecture is the best option. It is also the only existing option

providing a truly transparent, multi-protocol secure e-mail solution. Furthermore, it offers a number of additional benefits such as fostering a diversity of secure technologies and the previously mentioned benefits of this diversity. On the pragmatic side, it provides the flexibility and modularity required in a commercial product avoiding many of the current problems associated with supporting a diverse technology base. It facilitates the demarcation of the secure and insecure domains which, in turn, advocates the specialisation of skill sets for each of these domains and drastically improves secure application evaluation and certification including the re-use of certified modules. Users also benefit in that the modularity and flexibility greatly facilitate the maintenance and configuration management of secure implementations for network administrators or security managers as well as allowing users to select their own security service modules to secure e-mail communications.

7.2 Summary of Contributions

In summary, this thesis has made the following contributions:

- a. Justified the need for application level interoperability across multiple secure enveloping protocols for secure e-mail

- b. Identified a distinct lack of literature addressing this application level secure interoperability in general and specifically with respect to secure enveloping protocols for e-mail
- c. Provided a detailed explanation of cryptographic principles and e-mail fundamentals
- d. Presented the current state of research and development efforts in the area of SSAPIs, CAPIs and API-based security architectures
- e. Advanced the state of the art by proposing a solution to secure application level interoperable e-mail for the Internet
- f. Validated this solution with a prototype implementation

7.3 Future Work

The following recommendations are made with respect to future work in this area:

- a. The prototype could be completed using the full PGP API (PGP Tool Kit) and then used to demonstrate and/or prove selected properties of the IDUP interface in a multi-protocol environment. For example, demonstrating that

IDUP does not invalidate or introduce any potential weaknesses in securing an application. This would be an example of proving the property of "safe programming" described in Appendix C. Similarly, work could also be done to verify the IDUP security perimeter.

- b. Using the Common Criteria Secure Evaluation Methodology and an application secured using either GSS-API or the IDUP-GSS-API, verify the ability (and associated advantages) to de-couple the secure and non-secure functionality of the application towards the goal of maintaining a given evaluation rating despite changes to the non-secure functionality of the application.
- c. Another although similar demonstration could be developed using a different MUA that was also modified to use IDUP. Application level interoperability could then be shown across both MUAs and protocols - this being a more realistic simulation of an Internet scenario. Utilising different IDUP and/or GSS mechanisms would also emphasise the capability to support "plug and play cryptography" products.

In presenting this work it is hoped that the existing void in the literature is narrowed, the problem and its supporting technologies will be better understood and that the state of secure application level interoperable e-mail will be advanced closer to the goal of comprehensive, transparent security services across the Internet.

REFERENCES

1. Jim Carroll and Rick Broadhead, *Canadian Internet Handbook, Revised 1995 Edition*. Scarborough, Ontario: Prentice Hall Canada Inc., 1995.
2. D. Crocker, "Standard for the Format of ARPA Internet Text Messages," RFC 822, University of Delaware, Aug. 1982.
3. J. Postel, "Simple Mail Transport Protocol," RFC 821, Aug. 1982.
4. M.T.Rose, *The Internet Message: Closing the Book with Electronic Mail*. New Jersey: Prentice-Hall Inc., 1993.
5. D. Crocker and G. Klyne, "Internet Data Object Security", <http://www.g5forum.org/G5RPTv2-04.pdf>, Mar. 1998.
6. "Internet Messaging Primer: Accessing Your Mail When And Where You Want On The Internet", QUALCOMM White Paper, San Diego, California, April 24, 1997.
7. Nathaniel Borenstein and Ned Freed, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," RFC 2045, First Virtual and Innosoft, Nov. 1996.

8. Nathaniel Borenstein and Ned Freed, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," RFC 2046, First Virtual and Innosoft, Nov. 1996.
9. K. Moore, "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text," RFC 2047, University of Tennessee, Nov. 1996.
10. N. Fred, J. Klensin, and J. Postel, "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures," RFC 2048, Innosoft, MCI and ISI, Nov. 1996.
11. Nathaniel Borenstein and Ned Freed, "Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples," RFC 2049, First Virtual and Innosoft, Nov. 1996.
12. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, *Handbook of Applied Cryptography*, New York: CRC Press, 1997.
13. W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol IT-22, pp. 664-654, Nov. 1976.
14. B. Schneier, *Applied Cryptography Second Edition*. New York: John Wiley & Sons, 1996.

15. W.S. Jevons, *The Principles of Science* (1st ed. 1874, 2nd ed. 1877). London: Macmillan and Co. Ltd., 1907.
16. R.L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM*, vol 21, No. 2, Feb 1978.
17. R.C. Merkle, "Secure Communication Over an Insecure Channel", *Communications of the ACM*, 21 (1978), 294-299.
18. R. Needham and M. Abadi, "Prudent Engineering Practice for Cryptographic Protocols", *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 122-136, May 1994.
19. C. Shih, M. Jansson, R. Drummond and L. Yarbrough, "Requirements for Interoperable Internet EDI". Internet Draft draft-ietf-ediint-req-01.txt (expired), Nov. 1996.
20. Jim Galvin, Sandy Murphy, Steve Crocker, and Ned Freed, "Secure Multiparts for MIME: Multipart/Signed and Multipart/Encrypted," RFC 1847, Trusted Information Systems, Innosoft International and CyberCash Inc., Oct. 1995.

21. Rosenbloom and M. Cusumano, "Technological Pioneering and Competitive XXIX, No. 4, pp. 51-76, Summer 1987.
22. Personal communications with Carlisle Adams and various Internet news groups.
23. William Stallings, "S/MIME: E-Mail Gets Secure", Byte, pp. 41-42, July 1998.
24. D. Coppersmith, "The Data Encryption Standard (DES) and its strength against attacks," IBM Research, TJ Watson Research Center, Yorktown Heights, NY, 1995.
25. E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Advances in Cryptology - CRYPTO '90 Proceedings*, Springer-Verlag, pp.2-21, 1991.
26. Steven Telleen, "Virus Attacks Point Up Value of Diverse operating Systems," Internet World, pg. 17, 21 June 1999, www.internetworld.com/print/1999/06/21/intranet/19990621-advisor.html.
27. John Linn, "Generic Security Service Application Interface, Version 2," RFC 2078, OpenVision Technologies, Jan. 1997.
28. C. Adams, "Independent Data Unit Protection Generic Security Service Application Interface (IDUP-GSS-API)," RFC 2479, Entrust Technologies, Dec. 1998.

29. The Institute for the Future (ITF - Menlo Park, CA), "Messaging Practices in the Knowledge Economy", Study contracted by Pitney Bowes Inc, Jun 1999.
30. A. Briney, "Infosecurity: A View From the Frontlines", *Information Security*, vol 2, No. 2, Feb. 1999.
31. B Schneier, "Security Pitfalls in Cryptography", Essay from Counterpane Systems, www.counterpane.com, 1998.
32. S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, L. Repka, "S/MIME Version 2 Message Specification," RFC 2311, Mar. 1998.
33. RC2-40, Description of the RC2 Encryption Algorithm, <ftp://ftp.ietf.org/internet-drafts/draft-rivest-rc2desc>.
34. FIPS 46-3, "Triple Data Encryption Standard (DES)", Federal Information Processing Standards Publication 46, U.S. Department of Commerce/National Institute of Standards and Technology, Information Technology Laboratory (ITL), Gaithersburg, Maryland, 1999. Note: Also known as ANSI Standard x9.52, "Triple Data Encryption Algorithm Modes of Operation", 1998.

35. T Casar, R Eastabrooks, "A Recent Trend in Security - Cryptographic Toolkits", *Proceedings of the 11th Annual Canadian Information Technology Security Symposium*, May 1999.
36. NSA Cross Organisation CAPI Team, "Security Service API: Cryptographic API Recommendation," The National Security Agency, Ft. Meade, Maryland, June 12, 1995.
37. Warwick Ford, "PKI Grows Up," *Information Security*, vol 2, No. 11, pg. 54, Nov. 1999.
38. T. Spitzer, "Getting That Secure Feeling," *DBMS Online*, www.dbmsmag.com/9806d18.html, Jun 1998.
39. The Open Group, "Common Security: CDSA and CSSM, Version 2", Document C902, Nov, 1999, <http://www.opengroup.org/pubs/catalog/c902.htm>.
40. Network Associates Inc., "International Cryptography Experiment (ICE)", Mar, 1998, http://www.nai.com/nai_labs/asp_set/crypto/crypto_ice.asp.
41. B. Gladman, "US Government Controls on the Microsoft Cryptographic Application Programming Interface", Paper for the Third International ICE/CAPI Workshop, Mar 96, www.nai.com/nai_labs/asp_set/crypto/gladman5.asp.

42. C. Adams, "IDUP and SPKM: Developing Public-Key-Based APIs and Mechanisms for Communications Security Services," *Proceedings of the Symposium on Network and Distributed System Security*, vol XI+, pp. 128-35, 1996.
43. C. Adams, "Toward a 'Universal Decoder' for Secure Electronic Mail", unpublished work.
44. Whitfield Diffie, "The First Tens Years of Public Key Cryptography", G.J. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, 135-175, IEEE Press, 1992.
45. John Linn, "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421, Feb. 1993.
46. Steve Kent, "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management," RFC 1422, BBN Communications, Feb. 1993.
47. David Balenson, "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers," RFC 1423, Trusted Information Systems, Feb. 1993.
48. Burton Kaliski, "Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services," RFC 1424, RSA Laboratories, Feb. 1993.

49. Jim Galvin, Sandy Murphy, Steve Crocker, and Ned Freed, " MIME Object Security Services," RFC 1848, Trusted Information Systems, Innosoft International and CyberCash Inc., Oct. 1995.
50. Burton Kaliski, "An Overview of the PKCS Standards," RSA Laboratories Technical Note, Nov. 1993.
51. P. Zimmerman, "PGP User's Guide Volume I: Essential Topics," vol 1, Oct. 1994.
52. K. Yamamoto, "An Integration of PGP and MIME", *Proceedings of the Internet Society 1996 Symposium on Network and Distributed System Security*, Feb.1996.
53. M. Elkins, "MIME Security with Pretty Good Privacy (PGP)", RFC 2015, The Aerospace Corporation, Oct. 1996.
54. Secure Data Network System, "Message Security Protocol", Specification SDN.701, Rev 3.0, Mar. 1994.
55. Secure Data Network System, "Multipurpose Internet Mail Extensions with the Message Security Protocol", Specification SDN.704, Rev 1.3, Jan. 1996.

APPENDICES

Appendix A - A Complex Multipart MIME Message

This example is from RFC 2049 [11]. The message contains five parts that are to be displayed serially: two introductory plain text objects, an embedded multipart message, a text/enriched object, and a closing encapsulated text message in a non-ASCII character set. The embedded multipart message itself contains two objects to be displayed in parallel, a picture and an audio fragment.

```
MIME-Version: 1.0
From: Nathaniel Borenstein <nsb@nsb.fv.com>
To: Ned Freed <ned@innosoft.com>
Date: Fri, 07 Oct 1994 16:15:05 -0700 (PDT)
Subject: A multipart example
Content-Type: multipart/mixed; boundary=unique-boundary-1
```

This is the preamble area of a multipart message.
Mail readers that understand multipart format
should ignore this preamble.

If you are reading this text, you might want to
consider changing to a mail reader that understands
how to properly display multipart messages.

--unique-boundary-1

... Some text appears here ...

[Note that the blank between the boundary and the start
of the text in this part means no header fields were
given and this is text in the US-ASCII character set.
It could have been done with explicit typing as in the
next part.]

--unique-boundary-1
Content-type: text/plain; charset=US-ASCII

This could have been part of the previous part, but
illustrates explicit versus implicit typing of body

parts.

--unique-boundary-1

Content-Type: multipart/parallel; boundary=unique-boundary-2

--unique-boundary-2

Content-Type: audio/basic

Content-Transfer-Encoding: base64

... base64-encoded 8000 Hz single-channel
mu-law-format audio data goes here ...

--unique-boundary-2

Content-Type: image/jpeg

Content-Transfer-Encoding: base64

... base64-encoded image data goes here ...

--unique-boundary-2--

--unique-boundary-1

Content-type: text/enriched

This is <bold><italic>enriched.</italic></bold>
<smaller>as defined in RFC 1896</smaller>

Isn't it

<bigger><bigger>cool?</bigger></bigger>

--unique-boundary-1

Content-Type: message/rfc822

From: (mailbox in US-ASCII)

To: (address in US-ASCII)

Subject: (subject in US-ASCII)

Content-Type: Text/plain; charset=ISO-8859-1

Content-Transfer-Encoding: Quoted-printable

... Additional text in ISO-8859-1 goes here ...

--unique-boundary-1--

Appendix B – The RSA Public Key Cryptosystem

Diffie and Hellman laid the foundation for public key cryptography in 1976 [13]. Whitfield Diffie later wrote a personal and first hand account of the events leading up to this discovery in a work entitled, “The First Tens Years of Public Key Cryptography”[44]. Unfortunately, they had not yet identified a mathematical algorithm that was practical enough to be implemented as a public key cipher. In Feb 1978, three authors named R.L. Rivest, A. Shamir and L. Adleman provided such a mechanism; the RSA public key cryptosystem. The remainder of this subsection is spent explaining the mathematical background and concepts behind the RSA cipher.

Assume that Alice¹² wants to send Bob an encrypted message. Alice encrypts the message with Bob’s public key and sends it to Bob. It is not important how Alice got this key, as it is available publicly. It is only important that it is the correct key for Bob. Assuming the key is correct for Bob, and Bob received the encrypted message, he will now use his private or secret key to decrypt this message. Note that anyone else could have used Bob’s public key to encrypt this message and anyone else could also have intercepted the encrypted message from Alice to Bob. To correct for these possibilities, digital signatures are used to prove the originator of the message and the cipher itself (i.e. encryption) prevents everyone except Bob

¹² In lieu of the classic A and B variables used in mathematical syntax, Alice and Bob were used in RSA’s landmark paper [A Method for Obtaining Digital Signatures and Public-Key Cryptosystems] and are now the convention in public key cryptography. Whitfield Diffie [The First Ten Years of Public Key Cryptography] humorously refers to them as, “the first couple of cryptography”.

(or anyone else having the private key) from successfully decrypting the message. The following is a simple example to illustrate the basic operation of the RSA cipher. An actual implementation would utilise much larger integers to complicate cryptanalysis. This example expands upon the same cipher used in Chapter Two to demonstrate a trapdoor one-way cipher.

Bob's encryption cipher is $f(x) = x^3 \bmod 2773$ or $f(x) = x^e \bmod n$

His public key is $(3, 2773)$ or (e, n)

Bob's decryption cipher is $x = f(x)^{1779} \bmod 2773$ or $x = y^d \bmod n$

His private key is $(47, 59, 1779)$ or (p, q, d) .

P and q are required to calculate d as shown towards the end of this appendix. Once d is calculated, p and q will no longer be needed but they must never be revealed, as the value of n is publicly available. Knowledge of either p or q will allow anyone else to calculate d in the same manner as Bob did and will void this key pair (i.e. Bob no longer is the only one that knows the private key). Note that if this did occur, the entire cipher is not forfeited but only that particular key pair (i.e. that one bijection of the many available). The same cipher could continue to be used but a new key pair would need to be generated.

Assuming we are confident that the key pair has not been compromised, Alice must first convert her plaintext message into a series of integers. This is not to conceal the

information but merely to convert text into integers. The exact conversion is immaterial as long as there is no ambiguity as to the letter-integer substitutions as the reverse conversion must be done during the decryption of the message. For our example we will convert using $\mathbb{Z}_{26}$, the set of integers modulus 26. The integers are assigned as follows: a blank = 00, A = 01, B = 02 C = 03 ... Z = 26. Let's assume Alice's message was:

THE ANSWER IS TEN.

Converting this message using $\mathbb{Z}_{26}$ produces:

T	H	E		A	N	S	W	E	R		I	S		T	E	N
20	08	05	00	01	14	19	23	05	18	00	09	19	00	20	05	14

Table 7 - Plaintext converted to $\mathbb{Z}_{26}$

The plaintext integer value is then split into groupings having the same or less than the number of digits in the modulus n . Padding may be required to extend the plaintext integer value to the required group size boundary. Here we simply pad with blank values. In Bob's case, $n = 2773$, a value having four digits. The plaintext integer value is thus grouped into four digit pairs:

2008 0500 0114 1923 0518 0009 1900 2005 1400.

Each plaintext integer value is then encrypted for Bob by transforming the values from the plaintext (domain of the trapdoor one-way function) into the ciphertext (range or codomain of Bob's function). This is accomplished by applying the algorithm to the plaintext integer values using Bob's public key values:

$$f(x) = x^3 \bmod 2773 \quad \text{For the first value, 2008, the ciphertext value is:}$$

$$f(2008) = 2008^3 \bmod 2773 = 8096384512 \bmod 2773 = 0952$$

Similarly,

X	2008	0500	0114	1923	0518	0009	1900	2005	1400
f(x)	952	1479	1821	218	155	729	2019	1205	305

Table 8 - Encrypting plaintext into ciphertext using the RSA algorithm

The plaintext is thus encrypted using Bob's public key. Alice then sends the following ciphertext message to Bob:

095214791821021801550729201912050305

Bob receives this message and must then decrypt the ciphertext. Using his private key (i.e. trapdoor information) Bob inverts the encryption function to reveal the original plaintext integer values as follows:

$x = f(x)^{1779} \bmod 2773$ or $x = y^d \bmod n$ where d is the private key value (the multiplicative inverse of the encryption exponent over the modulus n). For the first ciphertext value, the plaintext value is:

$$x = f(0952)^{1779} \bmod 2773 = 9.8885 * 10^{5298} \bmod 2773 = 2008$$

The second is:

$$x = f(1479)^{1779} \bmod 2773 = 2.36255 * 10^{5639} \bmod 2773 = 0500$$

Once the entire ciphertext is decrypted, the resulting plaintext is actually the plaintext integer value of the original message. This integer value must then be converted from $\mathbb{Z}_{26}$ back into text.

20	08	05	00	01	14	19	23	05	18	00	09	19	00	20	05	14
T	H	E		A	N	S	W	E	R		I	S		T	E	N

Table 9 - $\mathbb{Z}_{26}$ converted back to plaintext

The security of the RSA cryptosystem hinges upon the difficulty of anyone else being able to determine Bob's private key. There are other approaches to breaking the cipher but none have been proven to be any easier than either a brute force attack on the key space or attempting to factor the modulus n to find the values of p and q ; assuming of course that Bob doesn't write it on a piece of paper and keep it beside his computer! Factoring a large bicomposite into its prime factors has always been a difficult mathematical problem. RSA was designed to take advantage of this difficulty. Note that no one has been able to prove that the security of the RSA cipher is actually equivalent to the integer factorisation problem. While it is widely accepted to be equivalent and there is little debate this is a difficult problem; there may in fact be a little known algorithm or technique to solve this problem or perhaps some other technique to break RSA that does not need to factor the modulus. The only certain fact in this regard is that it is not possible to prove mathematically that RSA is computationally equivalent to the integer factorisation problem.

In an actual implementation p and q are chosen in such a way as to maximise the difficulty of factoring n . For example, they should be sufficiently large to challenge even the most powerful computing machines (usually 100 digits each creating a 200 digit modulus as their product), similar in bitlength to avoid elliptic curve factoring algorithms and the difference large enough to avoid factoring through trial and error knowing that if $p \cong q$, then $n \cong \sqrt{n}$.

Illustrating the computational security of the RSA cipher, the RSA company has a standing challenge on the Internet to anyone that can break the cipher (i.e. factor the modulus)¹³. On 22 August 1999 the record was again advanced. A 512-bit modulus was factored into the correct prime factors after a calendar time period of approximately 5 months. The computational effort was enormous, although well within the capability of many large organisations. This new record took 292 computers at 11 different sites a total of 35.7 CPU years to find the values of p and q using a factoring technique called the General Number Field Sieve¹⁴. The General Number Field Sieve is considered the best general-purpose factoring algorithm. A 512-bit modulus is equivalent to a 155-digit value of n . Previous to this the largest modulus ever known to be factored was 140 digits in Jan 99 after approximately 9 weeks of calendar time. RSA's present recommendation is to use a 768-bit modulus which is 230 digits or greater.

Calculating the Private Key

Given the values of p and q , d may be calculated in exactly the same way as Bob found d . The mathematical background used to calculate d is based on a few facts and theorems from number theory. Without rigorous proofs, these are:

¹³ See <http://www.rsasecurity.com/rsalabs/challenges> for details

¹⁴ See <http://www.rsasecurity.com/rsalabs/challenges/factoring/rsa155.html> for this specific event

RSA is based on fields over bicomposite (non-prime) numbers, n , which are the product of two large primes. The one-way characteristic of RSA is based on the fact that it is easy to find large prime numbers having the correct characteristics and multiple them to produce the modulus n . On the other hand, factoring a large number into its prime factors is very difficult. This is why n is a large bicomposite number and not simply a large prime itself. Both the plaintext and ciphertext values are integers between 0 and $n-1$.

Euler's Totient Function (or the Euler Phi Function) $\phi(n)$ calculates the number of integers between 0 and n that are relatively prime to n . Relatively prime to n means that the only common factors between n and the number is 1. Another way of stating this is that the greatest common divisor (gcd) between n and the number equals 1. This is written $\text{gcd}(x,n) = 1$. For the case where p and q are prime and $n = pq$, then $\phi(n) = (p-1)(q-1)$. This is why the factors p and q must never be revealed: the value $\phi(n)$ cannot be determined without knowing them, which prevents the subsequent calculation of d . Note also that almost all of the integers between 0 and n are relatively prime to n , that is the value $\phi(n) = (p-1)(q-1)$ is almost the same value as $n = pq$.

Euler is also credited for another theorem called Euler's Theorem which states that if a number x is invertible mod n , then raising it to the value of $\phi(n)$ will equal unity mod n . This is written as:

$$\text{if } p \in \mathbb{Z}_n^*, \text{ then } p^{\phi(n)} = 1 \bmod n.$$

Where Z_n^* denotes the reduced set of residues. This is also known as a multiplicative group of Z_n . In other words, Z_n is the set of integers modulo n . Not every element in this set has a multiplicative inverse such that $px = 1 \pmod n$, where $p \in Z_n$ and x is its multiplicative inverse. Elements of Z_n are invertible if and only if $\gcd(p, n) = 1$ or they are relatively prime to the modulus n . The set of all the elements in Z_n that are invertible form a multiplicative group of Z_n . This subset or multiplicative group is denoted as Z_n^* . $Z_n^* = \{ p \in Z_n \mid \gcd(p, n) = 1 \}$. Where n itself is prime, then all the elements will be relatively prime to n and therefore $Z_n^* = \{ p \mid 1 \leq p \leq n-1 \}$. Therefore raising any one of these elements to $\phi(n)$ will produce $1 \pmod n$.

Another theorem that can be generalised to state essentially the same thing is Fermat's Little Theorem:

if $\gcd(x, n) = 1$, then $x^{n-1} = 1 \pmod n$ (or $x^{\phi(n)} = 1 \pmod n$ when n is prime).

It can now be shown how the private and public keys are related. Once this relationship is explained, two of Euclid's algorithms will be used to show how the private key is determined.

Recall from the basic characteristics of a cryptosystem; for each key $E_K: P \rightarrow C$ as well as $D_K: C \rightarrow P$ such that $D_K = E_K^{-1}$. This can also be written as:

$$D_K(E_K(p)) = P$$

Euler's theorem gives us $p^{\phi(n)} = 1 \pmod n$. Multiplying both sides by $p \pmod n$, is the same as saying $p^{\phi(n)+1} = p \pmod n$. If $p \pmod n$ were the plaintext, then this is very similar to $D_K(E_K(p)) = P$ where $D_K(E_K(p)) = p^{\phi(n)+1}$. Since we already have $E_K(p) = p^e \pmod n$, we must solve for D_K so that $p^{ed} \pmod n = p \pmod n$. In other words, we want $ed = 1 \pmod{\phi(n)}$ which is the same as $ed = k\phi(n) + 1$. If this were true then $p^{ed} = p^{k\phi(n)+1} = p * p^{k\phi(n)} = p(p^{\phi(n)})^k = p(1)^k = p \pmod n$.

We now need Euclid's algorithm as well as Euclid's Extended algorithm to calculate the value of d (i.e. private key) to make this true. Euclid's algorithm is used to find the gcd of two numbers. Euclid's Extended algorithm uses these results (the gcd) to formulate an equation of a form that identifies the modular inverse (d) of an invertible number (e). This explanation is best served by example.

Calculate d

$$P = 47, q = 59, \phi(n) = 2668, e = 3$$

Euclid's Theorem is based on the fact that if a and b are positive integers where $b < a$, then the $\gcd(a, b) = \gcd(b, a \bmod b)$. Continuing with the example previously started:

Find $\gcd(\phi(n), e)$ or $\gcd(2668, 3)$ using Euclid's algorithm

The algorithm uses a , b and r in the form $a = (x) b + r \bmod b$. It uses a repetitive cycle which left shifts the coefficients a , b and r until the last $r = 0$. The $\gcd$ is the value of r immediately prior to its final value of 0. In this case, a is initialised as $\phi(n)$ and b as 3. Therefore:

$a = (x) b + r$ becomes

$2668 = (889) 3 + 1$ where x = the necessary value to determine $r \bmod b$. For the next repetition, we left shift $b \rightarrow a$ and $r \rightarrow b$. The new equation becomes:

$3 = (3) 1 + 0$ Note that $r = 0$ indicating we have reached the last iteration and the $\gcd$ is the previous value for r , which is the value 1.

$2668 = (889) 3 + 1 \leftarrow$ This indicates that $\gcd(2668, 3) = 1$

Now use Euclid's extended algorithm to find the value of d . We know that $\gcd(2668, 3) = 1$ or $\gcd(\phi(n), e) = 1$. This is equivalent to saying $1 = de + k\phi(n)$ or $1 = d(3) +$

$k(2668)$ where k is some multiple of the modulus ($\phi(n)$) and d is the multiplicative inverse of $e \bmod \phi(n)$. Recall above where $2668 = (889)3 + 1$, therefore,

$$1 = -889(3) + 1(2668) \text{ now compare this format with}$$

$$1 = de + t\phi(n) \pmod{\phi(n)} \text{ which tells us that } d = -889 \bmod_{2668}$$

$$\text{But } -889 \bmod_{2668} = 1779 \bmod_{2668} \text{ and thus } d = 1779 \bmod_{2668}$$

We can check this by verifying that $(de) \pmod{2668} = 1 \bmod_{2668}$.

$$\text{So } (1779)(3) \bmod_{2668} = 5337 \bmod_{2668} = 2(2668) + 1 \bmod_{2668} = 1 \bmod_{2668}. \quad \checkmark$$

Appendix C – Cryptographic API Criteria

Algorithm Independence: *Algorithm Independence* is defined as the property that the CAPI does not specify use of a particular algorithm to provide cryptographic service. The CAPI must accommodate a broad range of choices of current and future cryptographic algorithms. This property gives an application access to any cryptographic algorithm supported by the underlying cryptomodule, enhancing interoperability. An example of algorithm independence is a reference to an “encryption algorithm” rather than the “DES algorithm.”

Application Independence: *Application Independence* is defined as the property that the CAPI is equally suitable for designing any application. The CAPI must provide cryptographic services to the variety of applications being written today as well as future applications. A CAPI that has the property of application independence will enable the programmer to write a variety of applications. This will give the CAPI widespread use and longevity. A well-designed CAPI should be able to support all applications such as store-and-forward applications, like e-mail, as well as connection-oriented applications, like file transfer.

Cryptomodule Independence: *Cryptomodule Independence* is the property that the CAPI, in providing its cryptographic service, can use a particular cryptomodule

as easily as any other. The application should not need to know the specifics of the underlying cryptographic implementation. For example, the application need not know whether the cryptography is provided in hardware or software. Cryptomodule independence provides a solid foundation for the use of multiple cryptomodule implementations.

Degree of Cryptographic Awareness: *Cryptographic Awareness* refers to the amount of cryptographic knowledge required of the application developer. It is also used to characterise how tightly integrated an application is to its security software and how much control the application has over these security services. The goal for the majority of applications is to require only a minimal degree of cryptographic knowledge from the developer. Some applications, for example those in key management, require a higher degree of cryptographic knowledge from the developer. CAPI specifications can fall anywhere in this spectrum from requiring little knowledge to requiring a great degree of expertise in cryptography. Therefore a CAPI suite must be used in order to support both cryptographically aware and cryptographically unaware applications.

Modular Design and Auxiliary Services: *Modular Design* is the division of the CAPI into units that work together to provide the complete security service, each of which has a focused purpose. *Auxiliary Services* are support services used by the goal cryptographic service. These can include: key life-cycle management,

cryptomodule verification, user authentication, certificate management, query capability, and (user-to-CAPI) session set-up/tear-down capability. The CAPI architecture must also be functionally complete. In simpler architectures, there may be only two modules: cryptographic service and key management. It is expected that auxiliary services that are not fully developed and modularised today will become separate modules. In many current standards, the lack of APIs providing access to these security support services has caused the inclusion of that functionality within the CAPI. While this meets our needs for now, continuing development will most likely result in numerous other auxiliary service APIs. As a separate auxiliary service, the CAPI must support, preferably include, a function to verify its underlying cryptosubsystem. It is envisioned, for example, that cryptographic token verification functionality could filter up through the CAPI suite from a lower level CAPI.

MISSI Support: MISSI provides evolutionary, interoperable security solutions for the Defence Information Infrastructure constituent programs via an integrated, cohesive security architecture composed of compatible building block products and a common Security Management Infrastructure. *MISSI Support* is defined as the ability to be extended to support current and anticipated MISSI cryptography. It is vital that no CAPIs preclude utilising current cryptographic products supported by MISSI.

Safe Programming: *Safe Programming* is the term used for describing steps taken to guard against inadvertent security mishaps by the programmer. Three aspects that contribute to safe programming include consistent naming conventions, information hiding, and ease of use. By having consistent naming conventions, the possibility of a programmer misunderstanding the purpose and use of each procedure and its parameters decreases. Information hiding is provided by the use of opaque handles and pointers, to prevent the inadvertent exposure of sensitive information. Ease of use results in the CAPI mechanism doing many of the detailed tasks of parameterising and sequencing of cryptographic operations, and is likely to decrease the probability of programmer error. This is especially important for CAPI specifications aimed at the cryptographically unaware programmer.

Security Perimeter: *Security Perimeter* is defined as the boundary that prevents sensitive security-related information from leaking out of the trusted computing base into untrusted applications. In trusted systems, architectures with a security perimeter will contain the cryptography within the perimeter while the application is outside. The CAPI provides the access between the untrusted components and the trusted components. The CAPI must not be used as a portal to violate the security perimeter. It must restrict access to sensitive cryptographic data (e.g., unprotected keys) and must not propagate such information beyond the CAPI interface.

Appendix D – Secure Enveloping Protocols

MIME Object Security Services (MOSS)

MOSS is currently a proposed IETF standard for secure e-mail over the Internet. The first IETF secure e-mail work was Privacy Enhanced Mail (PEM) [45, 46, 47, 48] which also remains on the Internet Proposed Standards List¹⁵. Much of the MOSS specification [49] is based on PEM. Unfortunately, PEM is limited in that it specifies security services for text messages only. With the introduction of MIME and the multitude of non-textual content-types, there was a need for a secure enveloping protocol that could serve all message types. The key development enabling MOSS to provide secure services to all MIME types was the creation of the Secure Multiparts for MIME described in Section 2.3.1.

MOSS assumes that there exists a public key for each recipient. Although the MOSS specification acknowledges the importance of validating the public keys used and evaluating the degree to which private keys are protected from disclosure; the issues are considered independent topics and not addressed in the protocol. In addition to supporting X.509 certificates, MOSS supports bare public/private key pairs permitting the “web of trust” key distribution model (see PGP). This makes the protocol immediately usable by individuals and small communities of co-operating users. Given this capability it is critical that security administrators are aware that MOSS presumes the validity of the keys it uses. The MOSS

¹⁵ RFC 2600, Internet Official Protocol Standards, March 2000.

protocol does not provide key validation services. The importance of this cannot be understated.

MOSS itself is centred on the MIME protocol. The basic operation of MOSS is to accept a MIME object as input, apply the required security service and then output a different secured MIME object. The output object is a reincarnation of the original input object together with the control information required to remove the applied security services. The MOSS protocol is independent of the cryptographic algorithm used. This gives MOSS the flexibility to be used for a variety of applications and frees it from deprecation should any specific algorithm is deemed insecure. The disadvantage is that since it doesn't mandate the encryption algorithm or key size, a MOSS implementation can use a key that's too small (i.e. deemed insecure) or fail to interoperate with other MOSS applications that have implemented different cryptographic algorithms. From this perspective MOSS has been criticised as being more of a security framework than a secure enveloping protocol.

MOSS provides digital signatures (data integrity, data origin authentication and non-repudiation with proof of origin) and encryption (confidentiality) services. MOSS uses a digital signature with appendix scheme and provides confidentiality using the algorithms listed later in this appendix in the Protocol Comparison - Part One, criterion 17. The digital signature service outputs a MIME object having a multipart/signed content-type. A multipart/encrypted MIME object is output for encryption services. In both cases the appropriate headers are created, formatted as per RFC 822 and inserted into the control

information of the secure multipart object. Because MOSS uses the Security Multiparts encapsulation scheme, which is based on MIME, the MOSS protocol takes advantage of the recursive nature of MIME. A MIME object can have either a digital signature or encryption applied to it; or, if it has already had a digital signature applied to it; can have another by simply using the output object of the previous application as the input object to the next application of the MOSS protocol. Figure 25 - Signed Message Using MOSS Protocol is an example of a MOSS output object. The "*" in the figure indicates the actual lines that were signed.

RFC-822 Header

From: Kevin Chafe<chafe@sce.carleton.ca>
 To: 513 Class<95_513@scs.carleton.ca>
 Date: 20 Mar 97 01:10:57 CST
 Subject: Crypto Presentation Slides
 MIME-Version: 1.0
 Content-Type: multipart/signed;
 protocol="application/moss-signature"; micalg="rsa-d5";
 boundary="unique-boundary"

RFC-1521 (MIME) Body

```

--unique-boundary
* Content-Type: multipart/mixed;
   boundary="inner-unique-boundary"
*
  --inner-unique-boundary
* Content-Type: text/plain
*
  Hello everyone, take a look at these attached slides will
  you? They are in PowerPoint 7 format. Let me know
  what you think.
  Thanks
*
  Kevin
*
  --inner-unique-boundary
  Content-Type: application/x-powerpoint7
  Content-Transfer-Encoding: Base64
*


Attached PowerPoint7 File


*
  --inner-unique-boundary--

  --unique-boundary
  Content-Type: application/moss-signature;

  Version: 5
  Originator-ID: Public Key goes here
  MIC-Info: RSA-MD5,RSA,Signature goes here

  --unique-boundary--
  
```

Figure 25 - Signed Message Using MOSS Protocol

Secure MIME (S/MIME)

Early in the quest to provide commercial security services based on public-key cryptography, a consortium of computer industry vendors; lead by RSA Data Security, created their own set of specifications detailing how to implement the RSA public-key algorithms into other applications. These specifications became known as the Public-Key Cryptography Standards (PKCS). PKCS is actually a suite of specifications. The number of PKCS standards has slowly increased since their debut in June, 1991. They currently number from 1 to 15, although numbers 2 and 4 have been deprecated and others have since been updated. S/MIME is a way to incorporate the pertinent PKCS into Internet e-mail messages using MIME. A listing of the 15 PKCS with a brief description of each is enclosed at Appendix E.

Of all the secure enveloping protocols, S/MIME is the most commercial in nature. Its "owners" are a distinct group of vendors with a direct commercial interest - to maximise RSA market penetration. It is the only one developed and promoted using commercial marketing practices, vice using the common, co-operative, committee-based standards model. For example, the PKCS are not standards at all, although the RSA consortium would dearly like them to become the industry standard (de facto). The following quotes are taken from PKCS documentation [50].

"The role of RSA Laboratories in the Standards-making process isretain sole decision-making authority on what each standard is. This includes arbitrary object identifier choices, etc....Thus the standards-making process is not the usual committee-oriented method."

"For simplicity, we refer to the various works involving public-key cryptography as 'standards', without regard to their formal approval by a standards-making body."

This is not meant to imply that the PKCS is technically or cryptographically inferior in any way to the other protocols, only that it has a relatively strong commercial character. As with any documentation that is solely controlled by the vendor, it exudes an obviously favourable bias which is in sharp contrast to the other "committee-oriented" protocol specifications. S/MIME is also marketed more actively than the other protocols, which are not "owned" the same way as PKCS. It is this focused marketing effort which is in part, no doubt, the reason for S/MIME popularity and widespread acceptance throughout the industry.

S/MIME provides digital signatures (data integrity, data origin authentication and non-repudiation with proof of origin) and encryption (confidentiality) services [32]. S/MIME uses a digital signature with appendix scheme and provides confidentiality using the algorithms listed later in this appendix in the Protocol Comparison - Part One, criterion 17. The protocol provides these services by extending the MIME specification and defining three new content-types: application/x-pkcs7-mime, application/x-pkcs10 and application/x-pkcs7-signature. Application/x-pkcs7-mime stipulates that a MIME body part has been protected as defined in PKCS #7, the Cryptographic Message Syntax Standard. Application/x-pkcs10 is used to submit certification request messages as defined in PKCS #10, the Certification Request Syntax Standard. Application/x-pkcs7-signature is used for PKCS #7 detached signatures.

PKCS #7 is similar to MSP in that it defines its own internal format that is also ASN.1, BER encoded. PKCS #7 defines a general structure labelled ContentInfo and several sub-structures or content types such as SignedData (signature data), EnvelopedData (encrypted data) and SignedAndEnvelopedData. These content types are PKCS specific definitions and not MIME content-types. Therefore, when the MIME content-type “application/x-pkcs7-mime” is used, the body part to which it refers will contain a PKCS #7 ContentInfo structure that is BER encoded (i.e. binary data). Because BER produces 8 bit data, it will also be either quote-printable or base64 encoded and have the applicable content-transfer-encoding MIME header. There are no restrictions on the format of the data that is to be signed or encrypted; however, it must be a MIME object. This is why the MIME sub-type is labelled x-pkcs7-mime and is required; so that once the PKCS processing is completed, the object can be passed directly to a normal MIME MUA. Note that S/MIME employs its own MIME extensions and does *not* use the secure multipart extensions for MIME as specified in RFC 1847. Figure 26 - Signed Message using S/MIME Protocol graphically demonstrates the S/MIME format and its encapsulation scheme.

In sharp contrast to the MOSS protocol, S/MIME literature emphasises the protocol’s ability to interoperate. Unlike the MOSS specification, S/MIME does specify a number of mandatory items to ensure compatibility across S/MIME implementations. Unfortunately, the scope of this interoperability is limited to an S/MIME MUA being able to communicate with another S/MIME MUA. The focus is on ensuring product compatibility and not on supporting competing protocols. A more appropriate term would be “inner-

operability". One area in which S/MIME does support backward compatibility across protocols is clear-signed messages. Clear signed messages have the content of the message in plain text, as per RFC 822, so that MUAs that are not PKCS or MIME aware can still present the message to the user even if the signature cannot be verified. S/MIME does use the secure multipart/signed explicitly for this purpose. The format of the multipart/signed message follows that specified in RFC 1847 with the second body part having a MIME content-type of application/x-pkcs7-signature and typically a Content-Transfer-Encoding of base64. The data in this body part is a PKCS #7 detached signature. A detached signature is a PKCS #7 data object (i.e. ContentInfo) of the SignedData PKCS content type having signatures on some external data and the inner content fields empty. The external data is the fully encoded MIME entity in the first part of the multipart/signed MIME object. Simply stated the second MIME body part is just a PKCS #7 SignedData type without the actual message data encapsulated within it. Instead, this message data is placed into the first body part as per RFC 1847. Note also that the message data in the first body part is plaintext and not BER encoded as it would normally be inside the PKCS #7 envelope.

RFC-822 Header

From: Kevin Chafe<chafe@sce.carleton.ca>

To: Class<class_list@sce.carleton.ca>

Date: 20 May 99 01:10:57 CST

Subject: Crypto Presentation Slides

MIME-Version: 1.0

Content-Type: application/x-pkcs7-mime;

Content-Transfer-Encoding: base64;

S/MIME Body (Encapsulated Content – BER and base64 encoded)

PKCS Content Type: SignedData

PKCS Control and Cryptographic Information

Inner RFC 1521 MIME Object

--unique-boundary

Content-Type: text/plain

(blank line)

Hello everyone, take a look at these attached
slides will you? They are in PowerPoint 7 format.
Let me know what you think.

Thanks

Kevin

(blank line)

--unique-boundary

Content-Type: application/x-powerpoint7

Content-Transfer-Encoding: Base64

(blank line)

Attached PowerPoint7 File

(blank line)

--unique-boundary--

Figure 26 - Signed Message using S/MIME Protocol

Pretty Good Privacy (PGP)

PGP is perhaps the most well known and wide spread cryptographic application used by the general public. Generally speaking, international versions are freely available and used by many throughout the world. Although export restricted, it is also available outside of North America through alternative distribution channels (i.e. other than the official North American distributors). PGP was originally designed to provide privacy and authentication by encrypting files for distribution through bulletin board services. It has since expanded its role and is also used to encrypt e-mail messages, among other applications.

Strictly speaking PGP began as an application and not a secure enveloping protocol like the others. The fact that PGP is a stand-alone application has been both a boon and a curse. PGP's widespread acceptance can be attributed in part to its independent or stand-alone nature as well as its technical simplicity. The original PGP had one algorithm (IDEA) for encryption. Like S/MIME, this and the fact that they all used the same internal PGP syntax or enveloping protocol ensured that all PGP installations were "inner-operable" with one another. Unlike the others, PGP did not need a formal certificate authority structure. Users simply had to install and use it like any other application. They did not need to have expertise in systems integration or security to use PGP; as they would need to employ the other protocols at the time. The other protocols are designed to be a component of some application and are integrated by an application developer. On the other hand, this stand-alone characteristic limited PGP's integration with MUAs. Users literally prepared their e-mail in the usual way and saved the message to their local disk. Then they had to manually invoke

PGP and enter the commands to encrypt this saved e-mail file which produced another file: the ciphertext. The user then mailed this ciphertext file as an attachment to an e-mail message. The recipient must reverse this process to decrypt the message into a plain text e-mail message. There are numerous shells or secondary programs that try to minimise this awkward, manual MUA/PGP interface. These shells do not truly integrate the MUA and PGP but instead automate the interface just described.

PGP offers the four services: digital signatures (data integrity, data origin authentication and non-repudiation with proof of origin), encryption (confidentiality), compression and radix-64 conversion [51]. PGP uses a digital signature with appendix scheme and provides confidentiality using the algorithms listed later in this appendix in the Protocol Comparison - Part One, criterion 17. Compression is a default service applied after signature but before encryption to frustrate cryptanalysis. Radix-64 conversion is a service to convert 8 bit binary data into 7 bit ASCII characters for transmission over Internet mail systems. This is conceptually identical to the content transfer encoding services provided by MIME (base64 and quote-printable). The actual formats or mapping scheme used in this conversion is, however, different and therefore incompatible with MIME. The output of the radix-64 conversion is called ASCII Armour.

PGP was unique in its concept of key management or key distribution. Unlike the other protocols at the time, which used a centralised, hierarchical model of trust; PGP used a decentralised “web of trust”. Simply stated this means that users public keys are distributed

through a type of honour system. The concept of certificates is used the same way as the other protocols - they legitimise a public key by having a trusted third party (a Certification Authority or CA) sign the certificate and "witness" this public key. The difference with PGP is that anyone can be a CA or an "introducer" as they are called in PGP documentation. There is no hierarchy of trust. The notion is that it is left up to the PGP user to decide which public keys they will trust. It is not left up to some third party. There is even a mechanism within PGP to prioritise or grade public keys. Each public key can be designated as unknown, untrusted, marginally trusted or completely trusted to certify other keys. Users can designate whomever they wish to execute the role of the CA.

One of PGP's biggest drawbacks was that it did not support automated processing of multimedia files. A user could encrypt multimedia files: PGP would simply treat them as 8 bit binary data files. But the receiver would then need detailed instructions as to how to manually decrypt and concatenate these files so that the receiver could then submit the correct files to the appropriate application(s) for presentation to the user. A simple example is given here to portray one of the incompatibilities with respect to automated multimedia processing. If a file is large in size PGP would automatically process (encrypt, sign and/or convert to radix-64) and then split the file into a sequence of smaller files. The purpose of this was to avoid problems caused by some Internet mail gateways that prohibit relaying messages greater than 64

kilobytes ¹⁶. PGP would assist by ensuring that e-mail and text files were within 64 kilobytes size limits. The receiver would then have to manually concatenate these smaller files into a single large file before submitting the complete or aggregated file to PGP to further processing. This relatively uncommon inconvenience when working with text files; became the norm with multimedia files, as they are characteristically large in size. As multimedia became more common, PGP needed a better way to process multimedia files.

As luck would have it, MIME was on the other side of the fence looking towards PGP (and others). MIME could obviously handle multimedia files as it was designed specifically for this purpose. Using the message/partial content-type, MIME provided basically the same file splitting and concatenating services as PGP. The difference was that it was all done automatically, it was transparent to the user and any MIME-compliant MUAs already had the necessary software routines. On the other hand, what MIME needed was some way to secure the services it had to offer. PGP and MIME were integrated to form PGP/MIME – a protocol designed to secure MIME objects with PGP as well as to envelope PGP objects with the MIME wrapper [52¹⁷, 53]. Figure 27 - Signed Message Using PGP and Secure Multiparts (i.e. PGP/MIME) shows an example output message.

¹⁶ SMTP (RFC 821) does not specify or limit the maximum message length (message content). However, support for lengths of at least 64 KB is mandated in Internet Standard 3 or RFC1123 - Requirements for Internet hosts - application and support.

¹⁷ Ref 52 was an important contribution in the development of PGP/MIME but the final version is slightly different and described in ref 53. One of these differences is that ref 53 takes advantage of Secure Multiparts for MIME (RFC 1847) and

PGP can output two different formats: binary and ASCII Armour (Radix 64). For Internet e-mail, ASCII Armour is required to be compatible with the 7 bit Internet mail transport system (same reason that MIME uses content transfer encoding). ASCII Armour is a mandatory requirement in the PGP/MIME specification (RFC 2015).

As an interesting aside, at roughly the same point time in time, another group within the IETF was planning a similar marriage only their focus was on integrating the IETF secure e-mail solution – Privacy Enhanced Mail (PEM) with MIME. The result eventually became known as MOSS.

Today, PGP has evolved into a mainstream secure enveloping protocol. The first step in this evolution was the initial step towards integrating PGP and MIME as previously described. The next step was fostering the programmatic integration of PGP into MUAs. This was done with version 5.0 of PGP. PGP 5.0 was completely rewritten and this time included an API to facilitate integration with MUAs.

Further details on PGP (all versions up to 6.0) are listed later in this appendix. The description in Protocol Comparisons - Part One is based on version 2.6 of PGP. The updated

creates three new MIME content types: application/pgp-encrypted, application/pgp-signature and application/pgp-keys. This was adopted in lieu a single new "application/pgp" content-type which was suggested in ref 52.

comparison in part two includes the later versions beginning with version 5.0 (there was no version 3 or 4, versioning went from 2.6 to 5.0). Version 5.0 introduced the following significant enhancements:

- a. Options for Triple DES, SHA and other public-key algorithms
- b. Enhanced key revocation procedures
- c. Improved key management procedures
- d. A completely rewritten code base
- e. An API for integrating PGP into other programs

The latest available version, PGP 6.5, adds functionality that is not as significant with respect to enveloping protocols. The following user-friendly features have been added:

- a. PGPnet - provides VPN functionality between itself and any other machine running PGPnet.
- b. Self-Decrypting Archives - can be used by users who do not even have PGP.
- c. Integrated PGP Command Line - provides a way to integrate PGP with other Windows applications and automated processes on your desktop system.
- d. Fingerprint Word List. When verifying a PGP public key fingerprint, you can now choose to view the fingerprint as a word list instead of hexadecimal characters. The word list is made up of special authentication words that PGP uses and are carefully selected to be phonetically distinct and easy to understand without phonetic ambiguity.

- e. Plug-ins for Outlook 2000 and Outlook Express 5.0. This version of PGP adds support for sending and receiving encrypted e-mail in Microsoft Outlook 2000 and Outlook Express 5.0.
- f. HTTP Proxy Support – for accessing HTTP key servers through the proxy.

To briefly summarise, PGP now has five distinct versions:

- a. The original, monolithic work which was not integrated into Internet mail explicitly. This includes all versions up to and including PGP version 2.6.
- b. The current standard, PGP/MIME, which integrates all versions up to PGP 2.6 into Internet mail, using the MIME standard.
- c. PGP 5.0-5.5 continues with PGP/MIME but allows the use of non-RSA algorithms (e.g. Diffie-Hellman/DSS)
- d. PGP 6.0-6.5 incorporates incremental user functionality over 5.5
- e. Lastly is a new and evolutionary IETF standards effort to develop an open, PGP specification that is fully integrated with MIME, called Open PGP.

RFC-822 Header

From: Kevin Chafe<chafe@sce.carleton.ca>
 To: Class<class_list@sce.carleton.ca>
 Date: 20 May 99 01:10:57 CST
 Subject: Crypto Presentation Slides
 MIME-Version: 1.0
 Content-Type: multipart/signed;
 boundary="unique-boundary2"; micalg=pgp-md5;
 protocol="application/pgp-signature"

RFC-1521 (MIME) Body

```

--unique-boundary2
* Content-type: multipart/mixed; boundary="unique-boundary"
*
* --unique-boundary
* Content-Type: text/plain
*
* Hello everyone, take a look at these attached slides will
* you? They are in PowerPoint 7 format. Let me know what
* you think.
* Thanks
*
* --unique-boundary
* Content-Type: application/x-powerpoint7
* Content-Transfer-Encoding: Base64
*
* Attached PowerPoint7 File
*
*
* --unique-boundary--
* --unique-boundary2
* Content-Type: application/pgp-signature
* -----BEGIN PGP MESSAGE-----
* Version: 2.6.2
*
* iQCVAwUBMJrRF2N9oWBghPDJAE9UQQAti7LuRVndBj
* rk4EqYBIb3h5QXIX/LC//jJV5bNvkZIGPIcEmI5iFd9boEgvpir
* HtiREEqLQRkYNoBActFBZmh9GC3C041WGq
* -----END PGP MESSAGE-----
* --unique-boundary2--
  
```

Figure 27 - Signed Message Using PGP and Secure Multiparts (i.e. PGP/MIME)

Message Security Protocol (MSP)

NSA developed MSP for the US Military in the late 1980s. It is independent of the user's message preparation system as well as the message content being protected [54]. Although MSP was initially intended to be used within an X.400 Message Handling System; it may also be used as a secure enveloping protocol with other messaging environments such as RFC 822/SMTP (The Internet). An X.400 message consists of content and an envelope. Figure 28 - X.400 Message Using MSP is a conceptual diagram showing an X.400 message that has been secured by MSP. The MSP output object (i.e. MSP content) is encapsulated within an X.400 (i.e. P1/P3/P7) envelope. In the Internet environment the X.400 envelope is replaced with the SMTP envelope. An example of an MSP message encapsulated by MIME is given in Figure 29 - Internet Message Using MSP.

Figure 28 - X.400 Message Using MSP

Since MSP was designed for X.400 environments, it produces an ASN.1 encoded output that, in turn, produces binary data as its output for transmission. In the X.400 environment this aids interoperability. Ironically, ASN.1 is not likewise routine on the Internet with the result that this encoding hinders more than it helps interoperability in this environment. In other words, a message signed using the MSP protocol is not backward compatible with RFC 822. Only MUAs that are MSP aware will be able to read the message, much less verify the signature.

MSP provides digital signatures (data integrity, data origin authentication and non-repudiation with proof of origin), encryption (confidentiality), Access Control and Non-repudiation with proof of delivery (signed receipts) services. MSP uses a digital signature with appendix scheme and provides confidentiality using the algorithms listed later in this appendix in the Protocol Comparison - Part One, criterion 17. The protocol provides these services by encapsulating the message content and then adding a MSP header. This header is the heart of the MSP protocol and is very different from an RFC 822 formatted header. All of the required control and cryptographic information used by MSP is incorporated into this single header. It can be a very large. The header is a sequence of structures and sub-structures. A high level breakdown of the MSP header structures and some of their content is as follows:

- a. OriginatorSecurityData. The originator X.509 certificate, identifiers for confidentiality, integrity and other algorithms and user keying material.

- b. SignatureBlock. Information used for either non-repudiation with proof of origin or delivery (i.e. the receiptsIndicator, hash values over unique variables created by the originator and used by recipients), recipients X.509 certificates and signature algorithm identifiers
- c. RecipientSecurityData. A set of PerRecipientTokens, one for each recipient of the message. Each token has a clear text tag to enable the recipient to select the proper token. The rest of the token is protected and contains the symmetric encryption key used to encrypt the message content and a security label
- d. ContentDescription. Information available to a recipient prior to MSP processing. It is an unprotected string which may be used to select MSP protected messages
- e. MlControlInformation. Mail list information used to process and track deliveries of messages to multiple recipients
- f. MspSequenceSignatureAlgorithm. An identifier indicating the signature algorithm used if the entire MSP header and message content are optionally signed
- g. MspSequenceSignatureCertificate. A certification path or certificate used to verify a signature value if the entire MSP header and message content are optionally signed

- h. EncapsulatedContent. An octet string representing the actual message content in ASN.1 binary output

Two services that are unique to MSP (and S/MIME version 3 ESS) are access control and non-repudiation with proof of delivery. Access control is a determination made involving the authorisation of the originator to send the message and the recipient to receive it. Based on the security label (see `recipientSecurityData`) of the message, MSP ensures that the classification, privacy or security category of the message is dominated by both originator and recipient authorisation levels as well as verifying that it is within the range allowed on the originator's workstation.

Non-repudiation with proof of delivery is achieved by the return of a signed receipt from the recipients. Using the same mechanism as message signatures for proof of origin, these receipts allow originators to establish to a third party that a recipient received the original message. This, of course, is only possible if the receipt is bound to the original message through the signature. Thus, this service is only possible for messages that were signed by the originator.

Figure 29 - Internet Message Using MSP is an example of the message similar in content to that shown in Figure 28 - X.400 Message Using MSP but digitally signed using the MSP protocol and encapsulated by MIME for use in the Internet messaging environment [55].

Note that in incorporating the MSP Content into the Internet environment, MIME encapsulation is used and ~~not~~ the Secure Multiparts for MIME.

Figure 29 - Internet Message Using MSP

Protocol Comparison - Part One (18 July 1996)

The IETF Electronic Data Interchange - Internet Integration (EDIINT) Working Group, drafted a functional specification for interoperable secure EDI over the Internet. EDI and secure e-mail security requirements are analogous. Many EDI operations would actually use Internet secure e-mail for their store-and-forward communications. The following table was extracted from this functional specification and used here to capture the state of these secure enveloping protocols as they applied to interoperable secure e-mail in July 1996.

Factor	PGP	S/MIME	MOSS	MSP
1) Exportable Out Of The USA	<p>PGP is already outside the USA and except for countries that prohibit encrypted messages with long key lengths (instead of just restricting the import of long key length algorithms) PGP long key length messages can be read.</p> <p>No since the encryption algorithm</p>	Has the 40 and 56 bit export restrictions if RC2 or RC4 is used for encryption.	<p>Not with full key length</p> <p>Depends on the data encryption algorithm used. RFC 1423 specifies DES in CBC mode, which is not exportable. Moss however allows the use of variety of cryptographic algorithms.</p>	Depends on the key management and data encryption algorithm used. MSP allows the use of variety of cryptographic algorithms.

Factor	PGP	S/MIME	MOSS	V/SZ
	specified is IDEA.			
2) Easily Integrated Into Products In A User Transparent Manner	Maybe in V.30. Not in earlier versions. There seems to be general disagreement on this one.	Yes.	Do not know.	Yes. Support for signed receipts may require GUI enhancements.
3) Fully Compatible With Like Versions World Wide	PGP version 2.6 is compatible with any earlier versions. Version 3.0 should be also.	RSA has an active interoperability program in place Implementations to the spec should guarantee interoperability.	Moss does not require any particular security algorithm. Moss provides the means to identify which algorithms are used for each message. A suite of algorithms is defined in RFC 1423.	Implementations to the spec should guarantee interoperability when the same cryptography is used.
4) Current Implementation Status	Version 3.0 is due out 4Q96. Version 2.6 is available. Qualcomm. Preamail. PGPMIME Reference Implementation.	Two companies have implemented several others have committed. Twelve companies currently committed to implementing in a pilot sponsored by Commercenet. Products and	TIS, Innosoft and SupplyTech.	SPYRUS, Nortel, Xerox, LJI, BBN, and J. G. Van Dyke all have implementations. Product is shipping. In use for military messages.

Factor	PGP	S/MIME	MOSS	VSP
		development tool-kits currently shipping.		
5) Confidentiality	Yes	Yes	Yes	Yes
6) Signature	Yes	Yes	Yes	Yes
7) Return Receipt	No	No	No	Yes - supports non-repudiation with proof of delivery.
8) Delivery Notification	Via MIME extensions RFC1891-94.	Via MIME extensions RFC1891-94.	Via MIME extensions RFC1891-94.	Via MIME extensions RFC1891-94.
9) Authentication	Yes	Yes	Yes	Yes
10) Multimedia	Yes	Yes	Yes	Yes
11) Integrity	Yes	Yes	Yes	Yes
12) Trust Model (Key Management & Revocation)	PGP 3.0 will have hierarchical model of public-key certificates. RSA used for key management in current versions. Ad hoc key revocation.	RSA based using X.509 all versions. NT's Entrust will be usable with this product very soon.	Both RSA and DES based key management.	X.509 all versions.
13) Certificate (Information, Format, Distribution)	Yes using proprietary "Key rings". Not clear what V3 will use.	Yes using X.509 - all versions.	Yes with optional X.509.	Yes using X.509.
14)	Base 64	ASN.1 - BER	Base 64	Base64 encoding

Factor	PGP	S/MIME	MOSS	V/S/P
Infrastructure Overhead	encoding.	and DER encoding. Base 64 encoding.	encoding.	and ASN.1 encoding.
15) Envelope Type	MIME/ASCII	PKCS #7 and MIME	MIME/ ASCII	MIME /ASN.1
16) Envelope / Structure Specification (ASN1 or ASCII)	ASCII	ASN.1 and ASCII	ASCII	ASN.1
17) Algorithms Supported				
Encryption,	RSA and IDEA in pre 3.0.	RSA. RC2, RC4 and RC5.	DES in CBC.	Algorithm independent. Implementations exist using: RSA & DES.
Key Management,	Diffie Hellman and DSA in Ver. 3.0.		RSA or DES.	FORTEZZA
One-way Hash,	IDEA in CBC.	MD5 & RSA.	MD2/MD5 and RSA.	
Digital Signature,	MD5 & RSA.	SHA-1.		(DSS SHA-1, KEA, Skipjack).
Key Lengths For Encryption)	128 for casual grade, 512 commercial grade, 2048 military grade. A 128 bit IDEA key length.	Note: S/MIME like Moss is a format and allows any type of algorithm to be specified. RSA of course specifies their own algorithms. Triple-DES/RC 5.	56 bit key lengths for DES. FORTEZZA. Note: Moss like S/MIME allows a variety of cryptographic algorithms to be used. The suite of algorithms	

Factor	PGP	S/MIME	MOSS	VSP
			defined above are found in RFC 1423.	
18) Common Algorithms With EDIFACT AUTACK List Of Codes	RSA (yes). IDEA (yes). DSA (yes). MD5 (yes).	RSA (yes). RC2 and RC4 (yes). DES (yes). MD5 (yes).	RSA (yes). DES (yes). MD5 (yes).	RSA (yes). DES (yes). MD5 (yes). DSS (yes). SHA-1 (yes).
19) Coexistence With Others For Reception (signature not readable) of MIME Multipart/Signed Data	Yes V3.0.	Yes, but user selectable.	Yes.	Yes, if used with MIME encapsulation (see <draft-housley-msp-mime-01.txt>.
20) Signed Message Body Readable By RFC822/ MIME Readers	Should be in V3.0.	Yes - if one of the options multipart/signed is used.	Yes.	Yes, if used with MIME encapsulation.
21) Signature Separate From Signed Document	Yes.	No.	Yes	Yes
22) Backward Compatibility	To PGP.	To PEM.	To PEM.	No.
23) Uses Proprietary Algorithms?	Ver 3.0 will use Diffie-Hellman with expiring patents in 1997. Ver 3.0 will use DSA (Digital	Yes, but it supports different options.	YES, but it supports different.	It supports both standards (FIPS and X9) as well as proprietary algorithms.

Factor	PGP	S/MIME	XOSS	V3.0
	Signature Algorithm invented at the NSA).			
24) Adequate Security For EDI Purposes	Yes.	Yes.	Yes.	Yes.
25) Scaleable	Not enough experience to tell. The current trust model will not scale well.	Not enough experience to tell.	Not enough experience to tell.	Yes.
26) Solid MIME Integration	V3.0 - yes.	Yes - but it mixes PKCS technology with MIME.	Yes.	Yes (see <draft-housley-msp-mime-01.txt>.
27) Variable Key Sizes Supported	No.	Yes, 40- 128 bit. Symmetric 512-2048 bit RSA.	Yes.	Yes. Symmetric (DES = 56 bits; SKIPJACK = 80 bits) Signature (DSS = 512 .. 1024 bits; RSA = 512 .. 2048 bits). Key Management (KEA = 1024 bits; RSA = 512 .. 2048 bits).
28) Only X.509 Or Other Certificate Distribution Methods		X.509 any version.		X.509 any version.
29) Very Solid	V3.0 Yes -	Yes.	No.	Yes. DISA is

Factor	PGP	S/MIME	MOSS	MS/P
API And Took Kit	anticipated.			submitting an API to X/Open. At least two toolkits for sale.
30) Tool Kits Can Be Made Available To All Countries Not Under UN Embargo	Probably from alternate sources.	Export version probably to most countries.	Export version probably to most countries.	Export version probably to most countries. Toolkits exported to many countries, including France. Alternate source likely.
31) Fit With Future Direction Of EDI	Unknown.	Unknown.	Unknown.	Unknown.

Updated Comparison of Secure Enveloping Protocols

Part Two (12 March 1998)

Early in 1998, a study was done to assess the state of secure Internet e-mail [5]. The following table was extracted from this report. Brief reference was made to the PEM, MSP and MOSS protocols stating; " They have no meaningful commercial support. Only PGP and S/MIME are viable candidates for global e-mail security."

The table uses row headings with these meanings:

Signing, Encrypting	Packaging sections of data and control information into Internet Mail and distinguishing between the sections
Records	Separating internal information "records" and fields
Transport Protection	Protecting data against vagaries of transport services--especially e-mail transport--by adding a layer of data encoding, for example, so that trailing white spaces are not eliminated
Selection	Mechanism for specifying choices among algorithms, etc.
Certificate	Associating identifiers with keys and validating the association
Session	Public key mechanism for exchanging random session keys between correspondents
Digest	Algorithm(s) for performing data integrity hash calculation
Signed	Algorithm(s) for encrypting content digest, to achieve data authentication
Encrypt	Algorithm(s) supported for encrypting content data to achieve privacy

Algorithms and mechanisms cited in the table are:

ASN.1	Data encoding as records in a binary, hierarchical structure, with type and length headers for each field.
CAST5	New data encryption algorithm developed in Canada, limited by a patent claim
CMS	Cryptographic Message Syntax, an open specification derived from RSADSI's PKCS#7
DES	Data Encryption Standard promoted by NIST; Now deemed easy to compromise; Triple-DES is a much stronger enhancement
Diffie-Hellman	The first public-key algorithm, with utility limited to exchange of session keys.
DSA	Digital Signature Algorithm promoted by the U.S. National Institutes of Standards and Technology (NIST)
ElGamal	Variant of Diffie-Hellman which can be used for signatures and encryption
IDEA	International Data Encryption Algorithm, used in the original version of PGP
MD5	Popular message digest algorithm which produces a 128-bit hash
MIME	Multipart/Signed, Multipart/Encrypted, MIME Content-Transfer-Encoding, etc., as appropriate
PKCS#7	RSADSI's ASN.1-based common cryptographic syntax for "records" or sets of fields, such as for signed data, encrypted data, certificates and certificate revocation lists
PKIX	Open public key certification mechanism based on the International Telecommunication Union's (ITU) X.509v3 being developed by the IETF
RSA	Public key algorithm that can be used both for encryption and digital signatures
SHA-1	Secure hash algorithm promoted for message digest use by NIST

In the following table, **bold entries** indicate the mandatory default choice. The term "Special" refers to use of techniques developed specifically for the cited work, rather than being developed independently and/or for more general use. Note that OpenPGP and S/MIME v3 are still under development and, therefore, subject to change.

	PGP 2.6	PGP/ MIME	PGP 5.0	OpenPGP	S/MIME v2	S/MIME v3
	ASCII	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text	Text	Text
	Text	Text	Text	Text		

Appendix E – Public Key Cryptography Standards (PKCS)

PKCS #1: RSA Encryption Standard

This standard provides recommendations for the implementation of public-key cryptography based on the RSA algorithm covering the following aspects: cryptographic primitives, encryption schemes, signature schemes with appendix and ASN.1 syntax for representing keys and for identifying the schemes.

PKCS #2: The PKCS # 2 standard has been discontinued - it is now part of PKCS # 1.

PKCS #3: Diffie-Hellman Key-Agreement Standard

This standard describes a method for implementing Diffie-Hellman key agreement. The intended application of this standard is in protocols for establishing secure communications.

PKCS #4: The PKCS # 4 standard has been discontinued - it is now part of the PKCS # 1.

PKCS #5: Password-Based Cryptography Standard

This document provides recommendations for the implementation of password-based cryptography, covering the following aspects: key derivation functions, encryption schemes,

message-authentication schemes and ASN.1 syntax identifying the techniques. These recommendations are intended for the protection of sensitive information such as private keys, as in PKCS #8. A password is not directly applicable as a key to any conventional cryptosystem; however, some processing of the password is required to perform cryptographic operations with the key. Special care is required in that processing to defend against search attacks. Other cryptographic techniques based on passwords, such as password-based key entity authentication and key establishment protocols are outside the scope of this document. Guidelines for the selection of passwords are also outside the scope.

PKCS #6: Extended-Certificate Syntax Standard

This standard describes syntax for extended certificates, consisting of a certificate and a set of attributes, collectively signed by the issuer of the certificate. The intended application of this standard is to extend the certification process beyond just the public key to certify other information about the given entity.

PKCS #7: Cryptographic Message Syntax Standard

This standard describes a general syntax for data that may have cryptography applied to it, such as digital signatures and digital envelopes.

PKCS #8: Private-Key Information Syntax Standard

This standard describes syntax for private-key information, including a private key for some public-key algorithm and a set of attributes. The standard also describes syntax for encrypted private keys. The intention of including a set of attributes is to provide a simple way for a user to establish trust in information such as a distinguished name or a top-level certification authority's public key.

PKCS #9: Selected Attribute Types

This standard defines selected attribute types for use in PKCS # 6 extended certificates, PKCS # 7 digitally signed messages, PKCS # 8 private-key information, and PKCS # 10 certificate-signing requests.

PKCS #10: Certification Request Syntax Standard

This standard describes syntax for a request for certification of a public key, a name, and possibly a set of attributes.

PKCS #11: Cryptographic Token Interface Standard

This standard specifies an API, called Cryptoki, to devices that hold cryptographic information and perform cryptographic functions. Cryptoki, pronounced cryp-toe-key and short for cryptographic token interface, follows a simple object-based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a cryptographic token.

PKCS #12: Personal Information Exchange Syntax Standard

This standard specifies a portable format for storing or transporting a user's private keys, certificates and miscellaneous secrets.

PKCS #13: Elliptic Curve Cryptography Standard

This is a new document being developed by RSA Laboratories as part of its Public-Key Cryptography Standards (PKCS) series, covering public-key techniques based on elliptic curve cryptography.

PKCS #14: Pseudorandom Number Generation Standard

This document will address many aspects of pseudorandom number generation. It is still under development.

PKCS #15: Cryptographic Token Information Format Standard

Tokens (Smart Cards) offer a great potential for secure identification of users of information systems. There is a requirement for credential portability and interoperability. Interoperability demands standardisation. PKCS # 15, is intended to establish a standard which ensures that users will be able to use cryptographic tokens to identify themselves to multiple, standards-aware applications, regardless of the application's cryptoki (or other token interface) provider. This necessitates agreement on the syntax for storing digital credentials (e.g. keys, certificates) on these tokens and how this information is to be accessed.

The PKCS # 11 specification alone cannot offer this functionality since it is an API specification aimed at offering applications a uniform interface to cryptographic tokens. Different tokens require different PKCS # 11 drivers, and unless a user's desktop has the 'right' PKCS # 11 driver installed, the user will be unable to use the token on that desktop.

Appendix F – Comparison of Cryptographic Toolkits

Vendor & Toolkit	Certification & Type				API				Language				O/S Platform				Symmetric Algorithm				Public Key				Standards Supported												
	FIPS 140 - 1	CEP	Other	Hardware	Software	Proprietary	GSS-API	JCE	PKCS # 11	C	C++	JAVA	Other	Win32	Sun Solaris	HP-UX	IBM AIX	JDK 1.1/1.2	Other	DES	Triple DES	RC2/RC4/RC5	CAST	Serpent	Proprietary	RSA/SHA-1	RSA/MD5	DSA/SHA-1	Diffie-Hellman	EC Support	S/MIME	PKCS	SSL	SPKM	IPSec	Other	
Baltimore www.baltimore.com ECS Desktop C/SSL&J/SS SMT JCRYPTO CST 7.0					✓	✓	✓	✓		✓		✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
Cericom www.cericom.com Security Builder Smart Card Evaluation Toolkit	✓	✓		✓	✓	✓			✓					✓	✓	✓			✓	✓	✓							✓	✓	✓							

Vendor & Toolkit	Certification & Type				API			Language			O/S Platform				Symmetric Algorithm				Public Key					Standards Supported												
	FIPS 140 - 1	CEP	Other	Hardware	Software	Proprietary	GSS-API	JCE	PKCS #11	C	C++	JAVA	Other	Win32	Sun Solaris	HP-UX	IBM AIX	UDK 1.1/1.2	Other	DES	Triple DES	RC2/RC4/RC5	CAST	Serpent	Proprietary	RSA/SHA-1	RSA/MD5	DSA/SHA-1	Diffie-Hellman	EC Support	S/MIME	PKCS	SSL	SPKM	IPSec	Other
Information Security www.infoseccorp.com AT&T Crypto- Graphic Development Kit					✓	✓				✓				✓	✓	✓			✓	✓	✓				✓	✓	✓	✓								
Litronic Inc. www.litronic.com Profile Manager EDK Profile Manager EDK-Plus				✓		✓			✓	✓	✓	✓		✓	✓	✓				✓	✓				✓	✓	✓									
Network Associates www.nai.com PGP Software Development Kit	✓				✓	✓				✓	✓			✓	✓					✓	✓		✓		✓	✓	✓	✓								

Vendor & Toolkit	Certification & Type	API	Language	O/S Platform	Symmetric Algorithm	Public Key	Standards Supported
	FIPS 140 - 1						
	CEP						
	Other						
	Hardware						
	Software						
	Proprietary						
	GSS-API						
	JCE						
	PKCS #11						
	C						
	C++						
	JAVA						
	Other						
	Win32						
	Sun Solaris						
	HP-UX						
	IBM AIX						
	SDK 1.1/1.2						
	Other						
	DES						
	Triple DES						
	RC2/RC4/RC5						
	CAST						
	Serpent						
	Proprietary						
	RSA/SHA-1						
	RSA/MD5						
	DSA/SHA-1						
	Diffie-Hellman						
	EC Support						
	S/MIME						
	PKCS						
	SSL						
	SPKM						
	IPSec						
	Other						
RSA Data Security www.rsa.com BSAFE Crypto - C Crypto - J SSL - C SSL - J S/MIME - C							
Spyrus www.spyrus.com SecureWeb Toolkit SecureWeb Payments FORTEZZA - Software Crypto Card							

Appendix G – Source Code Modifications For Prototype

```
/*
 * Filename: SignTranChafe.cpp
 *
 * Last Edited: Jan 25, 2000
 *
 * Author: Scott Manjourides
 * Modified first by: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This file is the code to incorporate PGP into the TIS (NAI) IDUP
 * SMIME Plug-in. It is a modified version of the SignTran.cpp file
 * distributed as part of the IDUP S/MIME Plug-In demonstration from
 * Network Associates Inc.
 */

#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
#include <afxwin.h> // MFC & Windows master header

#ifdef _DEBUG
#include <ctype.h>
#endif // _DEBUG

#include "resource.h"
#include "PseudoSn.h"

#include <malloc.h> // malloc, free
#include <string.h> // strdup

#include <istream.h> // ifstream class
#include <fstream.h> // ofstream class

#include <limits.h>

// Include for dialog boxes
#include "ServiceEvidenceDlg.h"
#include "GetKeyInfoEx.h"

#include "mimetype.h"
#include "encoding.h"
#include "rfc822.h"
#include "rfc1847.h"
```



```

#include <windows.h>
#include "ems-win.h" // The EMS API
#include "file1847.h"

/*****
/* CONSTANTS */

static const int kPluginID = 2185;          // Unique ID of this plugin
static const char FAR*kPluginDescription = "TIS IDUP-S/MIME Plugin";
static const unsigned long kPluginIconID = IDI_MAIN;

static const int kNotSignEncryptMessage = 0;
static const int kSpecialTrans           = 1;
static const int kArrivalTrans           = 2; // unique ID within the
plugin
static const int kDecodeTrans            = 3;
static const int kGeneralTrans           = 4;

static const int kNumTrans                = 4;      // How many translators in
this plugin

static const char FAR *kTransFailedStr = "Translator Failed";

#define kMimeMainType  "multipart"
#define kMimeSubType   "signed"
#define kSigMainType   "application"
#define kSigSubType    "pkcs7-signature"
#define kSigSubTypePgp "pgp-signature" // Chafe mod
#define kSignProtocol  (kSigMainType "/" kSigSubType)
#define kSignProtocolPgp (kSigMainType "/" kSigSubTypePgp) //Chafe
mod

/*****
/* GLOBALS */

const char *global_pInFilename; // Chafe mod - Used to pass pInFilename
from // DoTransmission to pgp_sign_message

extern CPseudoSnApp theApp;

static struct TransInfoStruct {
    const char *description;
    long type;
    long subtype;
    unsigned long flags;
    const char *MimeType;
    const char *MimeSubtype;
    const char *MimeProtocol;
    unsigned long nIconID;
} gTransInfo[] = {
    {

```

```

    "S/MIME Special Purpose (SE/EV) Protect",
    EMST_SIGNATURE,
    1L, // unused
    EMSF_Q4_TRANSMISSION | EMSF_WHOLE_MESSAGE | EMSF_REQUIRES_MIME |
    EMSF_GENERATES_MIME,
    NULL, NULL, NULL, /* Any type/subtype/protocol */
    IDI_SP_ICON
},
{
    "S/MIME Clear Sign Verifier",
    EMST_SIGNATURE,
    2L, // unused
    EMSF_ON_ARRIVAL | EMSF_ON_DISPLAY,
    kMimeMainType, kMimeSubType, kSignProtocol,
    IDI_SIGN_ICON /* OnArrival icon never shown */
},
{
    "S/MIME Decode Message",
    EMST_CERT_MANAGEMENT,
    3L,
    EMSF_ON_ARRIVAL | EMSF_ON_DISPLAY,
    "application", "pkcs7-mime", NULL,
    IDI_ENCRYPT_ICON
},
{
    "PGP Protect",
    EMST_CERT_MANAGEMENT,
    4L,
    EMSF_Q4_TRANSMISSION | EMSF_WHOLE_MESSAGE | EMSF_REQUIRES_MIME |
    EMSF_GENERATES_MIME,
    NULL, NULL, NULL,
    IDI_ENCRYPT_ICON
}
};

/*****
/* MACROS */

#define safefree(p) { if (p) { free(p); (p) = NULL; } }
#define safeddelete_buf(p) { if (p) { delete_buf(p); (p) = NULL; } }

/*****
/* LOCAL FUNCTION PROTOTYPES */

// Generalized functions
int CheckNotNowContext(short trans_id, long context);
int CheckNowContext(short trans_id, long context);
int CheckValidMimeType(short trans_id, emsMIMeTypeP in_mime);
void DoIconInit(short trans_id, HICON FAR*FAR* trans_icon);

// The actual filters

```

```

int DoTransmission(const char *pInFilename,
    const char *pOutFilename,
    emsProgress progress,
    emsMIMEtypeP FAR* out_mime,
    int encrypt,
    struct list_t *,
    int evidence,
    int simple,
    int sig_gop,
    int enc_gop);

int DoOnDisplay(const char *pInFilename,
    const char *pOutFilename,
    emsProgress progress,
    emsMIMEtypeP FAR* out_mime,
    char FAR*FAR* out_desc,
    int decode);

// The signature functions
int CalcFileHash(const char *pFilename, unsigned long *pHashVal,
    emsProgress progress);
void WriteSig(const char *pFilename, unsigned long nSigVal);
    unsigned long ReadSig(const char *pFilename);

int smime_protect_message (const char *out, const char *in, int
    encrypt, struct list_t *recips, int evid
    ence, int simple, int sig_gop, int enc_gop);
int smime_decode_message (const char *out, const char *pidufile, const
    char *sigfile, char **msg);

extern "C" {
int gn_CspType (const char *);
char *smime_db_path (void);
};

// Chafe Mod - add new function declaration
int FileRename(const char *in_name, const char *out_name);
extern int pgp_protect_message (const char *out, /* output file */
    const char *in, /* input file */
    int encrypt, /* perform encryption
        services */
    struct list_t *recips, /* recipients
        of message to
        encrypt to */
    int evidence, /* what type of
        evidence to request */
    int simple, /* use simplified
        functions /
    int sig_gop,
    int enc_gop); /* To copy smime
        prototype for IDUP
        calls later */

```

```

/*****
/* 16-BIT INITIALIZATION (if needed) */

#ifndef WIN32
/* FUNCTION: LibMain(HANDLE, WORD, WORD, LPSTR)

    PURPOSE:    Is called by LibEntry. LibEntry is called by Windows
                when the DLL is loaded. The LibEntry routine is provided
                in the LIBENTRY.OBJ in the SDK Link Libraries disk. (The
                source LIBENTRY.ASM is also provided.)

                LibEntry initializes the DLL's heap, if a HEAPSIZ value
                is specified in the DLL's DEF file. Then LibEntry calls
                LibMain. The LibMain function below satisfies that call.
*/

int CALLBACK LibMain(HINSTANCE hPlugin, WORD wDataSeg, WORD cbHeapSize,
                    LPSTR lpszCmdLine)
{
    //g_Instance = hPlugin;
    return 1;
}

/* FUNCTION: WEP(int)

    PURPOSE:    Performs cleanup tasks when the DLL is unloaded. WEP()
                is called automatically by Windows when the DLL is
                unloaded (no remaining tasks still have the DLL loaded).
                It is strongly recommended that a DLL have a WEP()
                function, even if it does nothing but returns success
                (1), as in this example. */

int CALLBACK WEP (int bSystemExit)
{
    return(1);
}
#endif // WIN32

/*****
/* TRANSLATER API FUNCTIONS */

/*****\
*
*  ems_plugin_version: Get the version of the API used for this plugin
*
\*****/

extern "C" long WINAPI ems_plugin_version(
    short FAR* api_version    /* Place to return api version */
)
{
    *api_version = EMS_VERSION;
}

```

```

    return (EMSR_OK);
}

/* ----- */

/*****\
*
*  ems_plugin_init: Initialize plugin and get its basic info
*
\*****/

extern "C" long WINAPI ems_plugin_init(
    void FAR* FAR* globals,      /* Return for allocated instance
                                structure */
    char FAR* config_dir,        /* Optional directory for config file */
    char FAR* user_name,         /* Users full name from Eudora config */
    char FAR* user_addr,         /* Users address (RFC-822 addr spec) */
    short FAR* num_trans,        /* Place to return num of translators */
    char FAR* FAR* plugin_desc,  /* Return for string description of
                                plugin */
    short FAR* plugin_id,        /* Place to return unique plugin id */
    HICON FAR* FAR* plugin_icon /* Return for plugin icon data */
)
{
    // NOTE:
    // We no longer need the use of the globals parameter. Thus I
    // commented it out. I left it in the code for reference in case
    // future demands require the use of the parameter
    ///////////////////////////////////////////////////////////////////
    //if (globals)
    //{
    //    if ( (*globals) = (void FAR *) malloc(sizeof(int)) )
    //        *((int FAR *) (*globals)) = CSettingsDlg::te_ask;
    //    else
    //        return (EMSR_TRANS_FAILED);
    //}
    //else
    //    return (EMSR_UNKNOWN_FAIL);
    ///////////////////////////////////////////////////////////////////

    if (num_trans)
        *num_trans = kNumTrans;

    if (plugin_desc)
        *plugin_desc = strdup(kPluginDescription);

    if (plugin_icon)
        DoIconInit(-1, plugin_icon);

    if (plugin_id)
        *plugin_id = kPluginID;
}

```

```

    return (EMSR_OK);
}

/* ----- */

/*****\
 *
 *  ems_translator_info: Get details about a translator in a plugin
 *
 \*****/

extern "C" long WINAPI ems_translator_info(
    void FAR* globals,          /* Pointer to translator instance
                                structure */
    short trans_id,             /* ID of translator to get info for */
    long FAR* trans_type,       /* Return for translator type, e.g.,
                                EMST_xxx */
    long FAR* trans_sub_type,    /* Return for translator subtype */
    unsigned long FAR* trans_flags, /* Return for translator flags */
    char FAR* FAR* trans_desc,   /* Return for translator string
                                description */
    HICON FAR* FAR* trans_icon   /* Return for translator icon data */
)
{
    TransInfoStruct *InfoPtr = NULL;

    if ( (trans_id <= 0) || (trans_id > kNumTrans) )
        return (EMSR_INVALID_TRANS);

    InfoPtr = gTransInfo + (trans_id - 1);

    if (trans_type)
        *trans_type = InfoPtr->type;

    if (trans_sub_type)
        *trans_sub_type = InfoPtr->subtype;

    if (trans_flags)
        *trans_flags = InfoPtr->flags;

    if (trans_desc)
        *trans_desc = strdup(InfoPtr->description);

    if (trans_icon)
        DoIconInit(trans_id, trans_icon);

    return (EMSR_OK);
}

/* ----- */

```

```

/*****\
*
*  ems_can_translate_file:
*      Check and see if a translation can be performed (file version)
*
\*****/

extern "C" long WINAPI ems_can_translate(
    void FAR* globals,          /* Pointer to translator instance
                                structure */
    long context,               /* Context for check; e.g. EMSF_ON_XXX */
    short trans_id,             /* ID of translator to call */
    emsMIMEtypeP in_mime,       /* MIME type of data to check */
    char FAR*FAR* addresses,    /* List of addresses (sender and
                                recipients) */
    char FAR* properties,       /* Properties for queued translations */
    char FAR*FAR* out_error,     /* Place to return string with error
                                message */
    long FAR* out_code           /* Return for translator-specific result
                                code */
)
{
    if ((trans_id <= 0) || (trans_id > kNumTrans))
        return (EMSR_INVALID_TRANS);

    if (CheckValidMimeType(trans_id, in_mime))
    {
        if (CheckNotNowContext(trans_id, context))
            return (EMSR_NOT_NOW);

        if (CheckNowContext(trans_id, context))
            return (EMSR_NOW);
    }

    return (EMSR_CANT_TRANS);
}

/* - - - - - */

/*****\
*
*  ems_translate_file: Actually perform a translation on a file
*
\*****/

extern "C" long WINAPI ems_translate_file(
    void FAR* globals,          /* Pointer to translator instance
                                sructure */
    long context,               /* Context for translation; e.g.
                                EMSF_ON_XXX */
    short trans_id,             /* ID of translator to call */
    emsMIMEtypeP in_mime,       /* MIME type of input data */

```

```

char FAR* in_file,          /* The input file name */
char FAR*FAR* addresses,    /* List of addresses (sender and
                             recipients) */
char FAR* properties,       /* Properties for queued translations */
emsProgress progress,       /* Func to report progress / check for
                             abort */
emsMIMEtypeP FAR* out_mime, /* Place to return MIME type of result*/
char FAR* out_file,         /* The output file (specified by
                             Eudora) */
HICON FAR*FAR* out_icon,    /* Place to return icon representing
                             result */
char FAR*FAR* out_desc,     /* Returned string for display with the
                             result */
char FAR*FAR* out_error,    /* Place to return string with error
                             message */
long FAR* out_code          /* Return for translator-specific
                             result code */
)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState()); // push all state on
                                                // stack. This is so we
                                                // don't have a problem
                                                // when the dialog pops
                                                // up. The state is
                                                // restored at the end
                                                // of this scope

    int nStatus = FILE1847_OK;

    if ((trans_id == kGeneralTrans) || (trans_id == kSpecialTrans))
    {
        ServiceEvidenceDlg  GpBox;
        int                encrypt = 0, evidence = 0;
        struct list_t      *curSelectedKeyList = NULL;

        if (trans_id == kSpecialTrans)

            GpBox.ChangeViewMode(ServiceEvidenceDlg::DISABLE_EVIDENCE_AND_ENCRYPT)
;

        char *dbf = smime_db_path ();
        int cspType = gn_CspType (dbf);
        free (dbf);

        switch (cspType)
        {
            case 1:
                GpBox.initAlgMode(ServiceEvidenceDlg::ALG_STATE_1);
                break;
            case 2:
                GpBox.initAlgMode(ServiceEvidenceDlg::ALG_STATE_2);
                break;
        }
    }
}

```



```
        case 3:
            GpBox.initAlgMode (ServiceEvidenceDlg::ALG_STATE_3);
            break;
    }

    if (GpBox.DoModal() != IDOK)
    {
        return (EMSR_TRANS_FAILED);
    }

    if (GpBox.m_rBtnService == ServiceEvidenceDlg::ENCRYPT)
        encrypt = 1;
    else if (GpBox.m_rBtnService ==
ServiceEvidenceDlg::SIGN_AND_ENCRYPT)
        encrypt = 2;
    else if (GpBox.m_rBtnService ==
ServiceEvidenceDlg::ENCRYPT_AND_SIGN)
        encrypt = 3;
    else if (GpBox.m_rBtnService == ServiceEvidenceDlg::SIGN_W_EV ||
GpBox.m_rBtnService ==
ServiceEvidenceDlg::SIGN_W_EV_AND_ENCRYPT)
    {
        if (GpBox.m_rBtnService ==
ServiceEvidenceDlg::SIGN_W_EV_AND_ENCRYPT)
            encrypt = 1;

        if (GpBox.rBtnEvidence == ServiceEvidenceDlg::CREATION)
            evidence = 1;
        else if (GpBox.rBtnEvidence == ServiceEvidenceDlg::SENDING)
            evidence = 2;
        else if (GpBox.rBtnEvidence == ServiceEvidenceDlg::ORIGIN)
            evidence = 3;

        if (GpBox.rBtnEvidenceRequest == ServiceEvidenceDlg::DELIVERY)
            evidence |= (1 << 2);
        else if (GpBox.rBtnEvidenceRequest ==
ServiceEvidenceDlg::RECEIPT)
            evidence |= (2 << 2);
        else if (GpBox.rBtnEvidenceRequest ==
ServiceEvidenceDlg::APPROVAL)
            evidence |= (3 << 2);
    }

    int sig_qop;
    switch (GpBox.rBtnSigAlg)
    {
    case ServiceEvidenceDlg::MD2:
        sig_qop = 2;
        break;
    case ServiceEvidenceDlg::MD5:
        sig_qop = 1;
```

```

        break;
    case ServiceEvidenceDlg::SHA1:
        sig_qop = 3;
        break;
    }

    int enc_qop;
    switch (GpBox.rBtnEncAlg)
    {
    case ServiceEvidenceDlg::RC2:
        enc_qop = 1;
        break;
    case ServiceEvidenceDlg::DES:
        enc_qop = 2;
        break;
    case ServiceEvidenceDlg::SKIPJACK:
        enc_qop = 3;
        break;
    }

    curSelectedKeyList = smime_fetch_keys (); // get the keys from the
    database

    if (encrypt || GpBox.ckboxEvidenceRequest)
    {
        GetKeyInfoEx      keyDialogBox (curSelectedKeyList);

        if (!GpBox.ckboxEvidenceRequest)
            keyDialogBox.ChangeViewMode (GetKeyInfoEx::DISABLE_EV_TO |
            GetKeyInfoEx::DISABLE_EV_FROM);
        else if (!encrypt)
            keyDialogBox.ChangeViewMode (GetKeyInfoEx::DISABLE_ENCRYPT);

        if (keyDialogBox.DoModal() != IDOK)
            return (EMSR_TRANS_FAILED);
    }

    if (nStatus == FILE1847_OK)
        nStatus = DoTransmission (in_file,
            out_file,
            progress,
            out_mime,
            encrypt,
            curSelectedKeyList,
            evidence,
            (trans_id == kSpecialTrans),
            sig_qop,
            enc_qop);
    }
    else // arrival
    {

```

```

    nStatus = DoOnDisplay(in_file,
                          out_file,
                          progress,
                          out_mime,
                          out_desc,
                          (trans_id == kDecodeTrans));
}

if (nStatus == FILE1847_FAIL)
{
    *out_error = strdup(kTransFailedStr);
    *out_code = __LINE__;

    return (EMSR_TRANS_FAILED);
}

if (nStatus == FILE1847_ABORT)
{
    return (EMSR_ABORTED);
}

return (EMSR_OK);
}

/* - - - - - */

/*****\
 *
 *  ems_plugin_finish: End use of a plugin and clean up
 *
 \*****/

extern "C" long WINAPI ems_plugin_finish(
    void FAR* globals          /* Pointer to translator instance
    structure */
)
{
    // NOTE:
    // We no longer need the use of the globals parameter. Thus I
    // commented it out. I left it in the code for reference in case
    // future demands require the use of the parameter
    //////////////////////////////////////
    //safefree(globals);
    //////////////////////////////////////

    return (EMSR_OK);
}

/* - - - - - */

/*****\
 *
```

```

* ems_free: Free memory allocated by EMS plug-in
*
\*****/

extern "C" long WINAPI ems_free(
    void FAR* mem          /* Memory to free */
)
{
    if (mem)
        safefree(mem);

    return (EMSR_OK);
}
/* ----- */

/* ----- Call the plug-ins configuration Interface ----- */
extern "C" long WINAPI ems_plugin_config(
    void FAR* globals,      /* Pointer to translator instance
                             structure */
    char FAR* config_dir,   /* Optional directory for config file */
    char FAR* user_name,    /* Users full name from Eudora config */
    char FAR* user_addr     /* Users address (RFC-822 addr spec) */
)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    return (EMSR_OK);
}
/* ----- */

/* ----- Manage properties for queued translations ----- */
extern "C" long WINAPI ems_queued_properties(
    void FAR* globals,      /* Pointer to translator instance
                             structure */
    long context,           /* EMSF_Q4_COMPLETION or _TRANSMISSION */
    short trans_id,         /* ID of translator to call */
    short FAR* selected,    /* Returned flag - is trans selected? */
    char FAR* FAR* properties /* ASCII string encoded properties */
)
{
    // globals --
    // NOTE:
    // I removed the use of the globals but kept the commented line within
    // the code just as a reference.
    // Brian Giovannoni 6/10/98
    // int trsenc = *((int FAR *) (globals));

    if (*selected) // Toggle OFF
    {

```

```

    (*selected) = FALSE;
//  giCurrentTranslationType = giCurrentTranslationType - trans_id;
//  signencrypt = 0;
}
else // Toggle ON
{
    (*selected) = TRUE;
//  giCurrentTranslationType = giCurrentTranslationType + trans_id;
    // keep track of whether or not we are doing a dual operation
//  signencrypt = (giCurrentTranslationType == kEncryptTrans +
kTransmissionTrans);
}

return (EMSR_OK); // Return value somewhat meaningless
}

/*****
/* THE TRANSLATOR ALGORITHMS */

int DoTransmission(const char *pInFilename,
                  const char *pOutFilename,
                  emsProgress progress,
                  emsMIMEtypeP FAR* out_mime,
                  int encrypt,
                  struct list_t *recips,
                  int evidence,
                  int simple,
                  int sig_qop,
                  int enc_qop)
{
    int nStatus = FILE1847_OK;

    /*
    Originally, GP (and SP encrypt or evidence calls) requests went through
    this branch to process 1847 Multipart Encrypted messages. For the
    purpose of this prototype, the GP switch is used to signal PGP
    processing. Within PGP, we are only implementing the clear signing
    feature to demonstrate the feasibility of using IDUP to facilitate
    interoperability across cryptosystems at the MUA level. Thus, messages
    requesting GP services (PGP clear signing) are now directed to the
    "else" branch which processes 1847 Multipart Signed messages. The
    disadvantage of this approach is that the original GP services of the
    S/MIME plug-in are no longer fully available. On the other hand, this
    is immaterial with respect to the goal of this prototype.
    */

    if (encrypt || evidence) // Chafe Mod - Was (!simple || encrypt ||
                           // evidence). Disabled this branch for GP (i.e.
                           // PGP)
    {
        char *tmpfile = strdup (CreateTmpFile ());

```

```
*out_mime = make_mime_type ("application", "pkcs7-mime", "1.0");

    if (!smime_protect_message (tmpfile,
                                pInFilename,
                                encrypt,
                                recips,
                                evidence,
                                simple,
                                sig_qop,
                                enc_qop))

        nStatus = FILE1847_FAIL;

    if (nStatus == FILE1847_OK)
    {
        Enc64 e64;
        FILE *out = fopen (pOutFilename, "w");
        FILE *in = fopen (tmpfile, "rb");
        char *bin, *enc;
        DWORD binlen, enclen, realenclen;
        struct stat st;

        fprintf (out, "Mime-Version: 1.0\r\n");
        fprintf (out, "Content-Type: application/pkcs7-mime\r\n");
        fprintf (out, "Content-Transfer-Encoding: Base64\r\n\r\n");

        // read data to be encoded
        fstat (fileno (in), &st);
        bin = (char *) malloc (st.st_size);
        binlen = fread (bin, 1, st.st_size, in);
        fclose (in);

        // allocate space for the encoded string
        enclen = (4 * binlen + 2) / 3;
        enclen += 2 * (enclen / 68 + 2); // add a CR-LF for every 68
                                         // chars
        enc = (char *) malloc (enclen);

        memset (&e64, 0, sizeof (Enc64));
        realenclen = Encode64 (bin, binlen, enc, &e64);
        fwrite (enc, 1, realenclen, out);

        // write out any leftover data
        realenclen = Encode64 (NULL, 0, enc, &e64);
        if (realenclen)
            fwrite (enc, 1, realenclen, out);

        fputs ("\r\n", out);
        fclose (out);

        safefree (bin);
        safefree (enc);
    }
}
```

[illegible]

```

// Call IDUP routines to set up and use PGP "mechanisms"
if (!pgp_protect_message (pOutFilename, pInFilename, 0, NULL, 0, 0
                          /*do NOT use simple - signal GP (ie PGP)*/,
                          sig_qop, enc_qop))

    nStatus = FILE1847_FAIL;

else // pgp clear signing OK.
    // Set TempOutFilename to PGP output filename
    // (XXXXXX.tmp.asc or pInFilename.asc)
    {
        strcpy(TempOutFilename, pInFilename);
        char *name_ptr = TempOutFilename;
        while (*name_ptr != '\0') // go through entire pInFilename name
            // string
            ++ name_ptr;
        strcpy(name_ptr, ".asc"); // Add Suffix to pInFilename
    } // End of "Set TempOutFilename to PGP output filename"

// End of "Everything OK to Call pgp_protect_message"

if (nStatus == FILE1847_OK)
{
    /*
We don't need this call as PGP6 does all this 1847 formatting for us.
We will,
however, rename the PGP output file from XXXXXX.tmp.asc to the name
Eudora expects (XXXXXX.tmp) by calling a new function FileRename.

        nStatus = rfc1847_file_create(pOutFilename, mainMimePtr,
        NULL, CTE_NONE, pInFilename,
        sigMimePtr, CTE_Base64,          pSigFilename, progress);
    */

    // Chafe mod - function to change pOutFileName from xxxxxx.asc to
    // xxxxxx.tmp

    FileRename(TempOutFilename, pOutFilename);

    // If Eudora wants out mime type structure, give it up

    if ((out_mime) && (nStatus == FILE1847_OK))
        *out_mime = mainMimePtr;
    else
        free_mime_type(mainMimePtr);
}
// End of !simple PGP clear signed processing branch
else // create Multipart signed for smime (original code)

{ char *pSigFilename = NULL;

```



```

OutputDebugString("\nEntering Original Code for S/MIME Multipart signed
                    messages\n");

// Create the mime type and header string
emsMIMEtypeP mainMimePtr = make_mime_type(kMimeMainType,
                                           kMimeSubType, "1.0");
emsMIMEtypeP sigMimePtr = make_mime_type(kSigMainType, kSigSubType,
                                           "1.0");

/* Check if everything was allocated */
if ((!mainMimePtr) || (!sigMimePtr))
{
    free_mime_type(mainMimePtr);
    free_mime_type(sigMimePtr);

    nStatus = FILE1847_FAIL;
}

unsigned long nHashVal = 0;

if (nStatus == FILE1847_OK)
{
    pSigFilename = strdup(CreateTmpFile());

    add_mime_parameter(mainMimePtr, "protocol", kSignProtocol);
    if (!smime_protect_message(pSigFilename, pInFilename, 0, NULL, 0, 1
                              /* always use simple */, sig_qop,
                              enc_qop))
    {
        safefree(pSigFilename);
        nStatus = FILE1847_FAIL;
    }
}

if (nStatus == FILE1847_OK)
{
    nStatus = rfc1847_file_create(pOutFilename, mainMimePtr,
                                NULL, CTE_NONE, pInFilename,
                                sigMimePtr, CTE_Base64,
                                pSigFilename, progress);

    free_mime_type(sigMimePtr);

    // If Eudora wants out mime type structure, give it up
    if ((out_mime) && (nStatus == FILE1847_OK))
        *out_mime = mainMimePtr;
    else
        free_mime_type(mainMimePtr);
}

} // End of else "create Multipart signed for smime (original code)"

```

```
} // End of else "Create Multipart signed message for PGP and smime"

return(nStatus);
}

int DoOnDisplay(const char *pInFilename,
               const char *pOutFilename,
               emsProgress progress,
               emsMIMEtypeP FAR* out_mime,
               char FAR*FAR* out_desc,
               int decode)
{
    emsMIMEtypeP InMimePtr = NULL, SigMimePtr = NULL;
    int nStatus = FILE1847_OK;

    if (decode)
    {
        Dec64 d64;
        char *tmpfile = strdup (CreateTmpFile ());
        FILE *in = fopen (pInFilename, "rb");
        FILE *out = fopen (tmpfile, "wb");
        char *enc, *encptr;
        char *dec;
        long enclen, declen, realdeclen, errs = 0;
        struct stat st;

        fstat (fileno (in), &st);
        enc = (char *) malloc (st.st_size);
        enclen = fread (enc, 1, st.st_size, in);
        fclose (in);

        /* HACK - move forward in the stream past the MIME header.  It is
        assumed that the data is Base64 encoded */
        encptr = enc;
        while (encptr < enc + enclen && memcmp (encptr, "\r\n\r\n", 4) != 0)
            encptr++;
        encptr += 4;
        enclen -= (encptr - enc); // reset the length to the actual size

        declen = (3 * enclen) / 4;
        dec = (char *) malloc (declen);

        memset (&d64, 0, sizeof (d64));
        realdeclen = Decode64 (encptr, enclen, dec, &d64, &errs);
        fwrite (dec, 1, realdeclen, out);

        realdeclen = Decode64 (NULL, 0, dec, &d64, &errs);
        if (realdeclen)
            fwrite (dec, 1, realdeclen, out);
    }
}
```

```

fclose (out);

if (!smime_decode_message (pOutFilename, tmpfile, NULL, out_desc))
{
    nStatus = FILE1847_FAIL;
    *out_desc = strdup ("Decode failed.\n\n");
}
else
{
    // assume text/plain for now
    *out_mime = make_mime_type ("text", "plain", "1.0");
}
safefree (tmpfile);
}
else
{
    char *pSigFilename = strdup(CreateTmpFile());

    int nStatus = rfc1847_file_parse(pInFilename,
        &InMimePtr,

        pOutFilename,
        out_mime,
        FALSE,

        pSigFilename,
        &SigMimePtr,
        TRUE,

        progress);

    if (nStatus == FILE1847_OK)
    {
        if (!smime_decode_message (NULL, pOutFilename, pSigFilename,
out_desc))
            *out_desc = strdup("Signature failed.\n\n");
    }
}

return (nStatus);
}

/*****
/* HELPER FUNCTIONS */

static int CheckNotNowContext(
    short trans_id, /* ID of translator */
    long context /* Context for check; e.g. EMSF_ON_xxx */
)
{
    return ((EMSF_ON_DISPLAY & gTransInfo[trans_id-1].flags) &&
(EMSF_ON_ARRIVAL & context));
}

```

```

}

/* ----- */
static int CheckNowContext(
    short trans_id, /* ID of translator */
    long context    /* Context for check; e.g. EMSF_ON_xxx */
)
{
    return ((context & gTransInfo[trans_id-1].flags) ? 1 : 0);
}

/* ----- */

static int CheckValidMimeType(
    short trans_id, /* ID of translator to check */
    emsMimeTypeP in_mime /* MIME type of incoming data */
)
{
    const char *pType = gTransInfo[trans_id-1].MimeType;
    const char *pSubtype = gTransInfo[trans_id-1].MimeSubtype;
    const char *pProtocol = gTransInfo[trans_id-1].MimeProtocol;

    if (match_mime_type(in_mime, pType, pSubtype))
    {
        if (!pProtocol)
            return TRUE;

        const char *pParam = get_mime_parameter(in_mime, "protocol");

        if (pParam)
            if (strcmp(pProtocol, pParam) == 0)
                return TRUE;
    }

    return FALSE;
}

/* ----- */

static void DoIconInit(
    short trans_id, /* ID of translator to get info for */
    HICON FAR*FAR* icon /* Return for icon data */
)
{
    if (!icon)
        return;

    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    *icon = (HICON *)malloc(sizeof(HICON));

    if (trans_id < 0) /* Main plugin icon, not specific translator */

```

```

{
    **icon = theApp.LoadIcon(MAKEINTRESOURCE(kPluginIconID)); /* 32x32 */
}
else /* The actual translators */
{
    **icon = theApp.LoadIcon(MAKEINTRESOURCE(gTransInfo[trans_id-
1].nIconID)); /* 16x16 */
}
}

/*****
/* CHAFE MOD - NEWLY CREATED HELPER FUNCTIONS */

/*
    FileRename function definition

This function accepts an input and output file name. It changes the
input file name (in_name) to the name specified by the output file name
(out_name) pointer. It uses the rename stdlib call which will only
work if the input file already exists, the output file does NOT already
exist and the output file cannot be open when rename is called. The
rest of the code is here to make the above checks and set up the
correct conditions, if need be. The function returns an int status
code of either OK or FAIL to signal Eudora whether or not to continue.
*/

int FileRename(const char *in_name, const char *out_name)

{
    // Declare local variables

    int RenameStatus = FILE1847_OK;

    FILE *fileExist = NULL; // Flag indicating filename already exists

    /*
    Check if a file with the new name (output file name) already exists.
    Must delete it first as rename fails if the new file name (output file
    name) already exists (i.e. it will not rename the file to a name that
    already exists.

This will not be problematic for the S/MIME plug-in as the pOutFileName
(new name) created by Eudora and passed to DoTransmission is just an
empty file until DoTransmission creates an 1847 multipart message and
puts it into this empty file. In the case of the PGP code, this
function is called to delete this empty file and rename the 1847
multipart file newly created by PGP. The end result is a file with the
same name as that specified by Eudora but with the PGP content. Eudora
simply grabs the file and processes it the same way it handles S/MIME
content.
    */

```

```
if( (fileExist = fopen(out_name, "r" )) == NULL )

#ifdef _DEBUG

    OutputDebugString( "\nThe new (out_name) file name does not exist and
will now be used to rename the old file name.\n");
#else _DEBUG
    ; // do nothing for Win32 - Release (i.e. production version)
#endif _DEBUG

else // File having out_name already exists, must close then delete it
    // for rename to work
    {

#ifdef _DEBUG
        OutputDebugString( "\nThe new (out_name) file does exist and will now
be closed then deleted.\n");
#endif _DEBUG

        if( fclose( fileExist ) ) // File was not closed for some reason
        {
#ifdef _DEBUG
            OutputDebugString( "\nThe new (out_name) file was not closed and
will not be renamed.\n");
#endif _DEBUG

            RenameStatus = FILE1847_FAIL; // error code to tell plug in
            // something is wrong
            return (RenameStatus); // Game over, tell Eudora to quit
        }

        else // File was closed

#ifdef _DEBUG
            OutputDebugString( "\nThe new (out_name) file was closed.\n");
#endif _DEBUG

        // rename will not work if out_name already exists

        // Note also this call will not be made if file was not closed
        // (conditional upon else statement)

        RenameStatus = remove(out_name); // Remove returns: 0 = success,
            // -1 = failure

        if (RenameStatus) // Could be set to fail (-1) from either remove
            // call or "RenameStatus = FILE1847_FAIL;" (line
            // 1040) above
        {
#ifdef _DEBUG
```

```
        OutputDebugString( "\nThe previously existing file with the new
                           (out_name) file name was not removed.\n");
#endif _DEBUG

        RenameStatus = FILE1847_FAIL; // RenameStatus already -1 from
                                     // rename but set here anyway just in
                                     // case
        return (RenameStatus); // Game over, tell Eudora to quit
    }
    else // File deleted OK
    {
#ifdef _DEBUG
        OutputDebugString( "\nThe previously existing file with the new
                           (out_name) name was deleted\n");
#endif _DEBUG

        RenameStatus = FILE1847_OK; // RenameStatus already 0 (OK) from
                                    // rename but set here anyway just
                                    // in case
    }
    // End of else - File having new file name (out_name) already
    // exists, must delete

    // No problems with new file name (out_name). Now check out
    // in_name.

    // Check if the "old file" (in_name) or PGP (*.asc) file exists

    if( (fileExist = fopen(in_name, "r" )) == NULL )
    {
#ifdef _DEBUG
        OutputDebugString( "\nThe PGP *.asc file (in_name) does not
                           exist\n");
#endif _DEBUG
        RenameStatus = FILE1847_FAIL; // Game over, something wrong with PGP
                                     // mechanism.
        return (RenameStatus); // Tell Eudora to quit.
    }

    else
        fclose(fileExist); // close file before we change name.
                           // Rename will not rename open files.

        RenameStatus = rename(in_name, out_name); // Rename returns: 0 =
                                                  // successful, non-zero =
                                                  // fail

        if (RenameStatus) // rename failure
        {
#ifdef _DEBUG
            OutputDebugString( "\nThere was a problem with the rename function.
                               PGP .asc file was not renamed.\n");
#endif _DEBUG
        }
    }
}
```

```
#endif _DEBUG
    RenameStatus = FILE1847_FAIL; // Game over, something wrong with
                                // rename
    return (RenameStatus);      // Tell Eudora to quit.
}

else // rename OK
{
#ifdef _DEBUG
    OutputDebugString( "\nThe PGP .asc file was renamed.\n");
#endif _DEBUG
    RenameStatus = FILE1847_OK; // Game over, something wrong with rename
}

return (RenameStatus); // Give DoTransmission the signal to continue
                        // with the 1847 Multipart PGP clear-signed
                        // message now under the name originally
                        // specified by Eudora (which previously would
                        // have been an S/MIME clear-signed (or
                        // protected) message.
}
```



```

/*
 * Filename: smimeChafe.cpp
 *
 * Last Edited: Feb, 2000
 *
 * Author: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This is a modified version of the original smime.cpp file. It
 * creates a new function entitled pgp_protect_message. The original
 * smime.cpp code and headers are still needed so the entire smime.cpp
 * file is included with an include statement.
 */

#include "smime.cpp" // To include code from smime.cpp
extern const char *global_pInFilename; // defined in SignTranChafe.cpp
int pgp_protect_message (const char *out, /* output file */
                        const char *in, /* input file */
                        int encrypt, /* perform encryption
                                     services */
                        struct list_t *recips, /* recipients of
                                               message to encrypt
                                               to */
                        int evidence, /* what type of evidence to
                                     request */
                        int simple, /* use simplified
                                     functions */
                        int sig_qop,
                        int enc_qop)
{
    gss_name_t gss_name;
    gss_buffer_desc idu_buffer, pidu_buffer, name;
    OM_uint32 minor_status, result;
    gss_cred_id_t cred_handle;
    idup_env_id_t env_handle;
    IDUP_Protect_Options_desc protect_options;
    IDUP_Target_Info_desc target_info;
    _TCHAR* user;

    struct stat st;
    FILE *fp;

#ifdef _DEBUG
    OutputDebugString ("pgp_protect_message");
#endif _DEBUG

/*
Check for missing encryption/evidence recipients. This does not catch
the case where the user selects evidence recipients but not encryption
recipients.

```

```

*/
    if (!recips && (encrypt || ((evidence >> 2) & 0x03)))
        return 0;

/* read the file to be signed/encrypted */
if ((fp = fopen (in, "rb")) == NULL)
    return 0;
fstat (fileno (fp), &st);
idu_buffer.value = malloc (st.st_size);
idu_buffer.length = fread (idu_buffer.value, 1, st.st_size, fp);
fclose (fp);

/*****/

idu_buffer.value = (void *)global_pInFilename; // HACK to pass
                                                // pInFilename
                                                // through IDUP API to
                                                // pgp_sign_message

/*****/

/* first, tell the IDUP library who I am */
CSP_GetUserName (&user);
name.value = user;
name.length = strlen ((char *) name.value);

result = gss_import_name (&minor_status,
    &name,
    GSS_C_NULL_OID, /* use mech specific meaning */
    &gss_name);

free (user);

IDUPERR("gss_import_name");

result = gss_acquire_cred (&minor_status,
    gss_name,
    GSS_C_INDEFINITE, /* don't expire this credential */
    GSS_C_NULL_OID_SET, /* use mech specific default */
    GSS_C_BOTH,
    &cred_handle, /* returned credential */
    NULL,
    NULL);
IDUPERR("gss_acquire_cred");

/*
We want to call PGP policy but no OID yet defined. Define
GSS_C_PGP_OID to match same definition used in
idup_establish_env_chafe.c.
*/
gss_OID_desc GSS_C_PGP_OID = { 1, (void *) "\004" };

```

```

result = idup_establish_env (cred_handle,
                             GSS_C_NULL_OID, /* req_mech_type */
                             &GSS_C_PGP_OID, /* req_policy - Chafe mod
                             - default is smime,
                             therefore call PGP */
                             0, /* policy_time */
                             NULL /* requested services? - required
                             by spec, but the PER_* oid's listed
                             in the spec are not defined in dup.h
                             or gss.h */,
                             &minor_status,
                             &env_handle,
                             NULL, /* actual_mech_type - required by
                             spec */
                             NULL, /*actual_policy-required by spec*/
                             NULL, /*actual_policy_time-required by
                             spec*/
                             NULL); /*ret_services-required by spec*/
IDUPERR("idup_establish_env");

memset (&pidu_buffer, 0, sizeof (pidu_buffer));
memset (&target_info, 0, sizeof (IDUP_Target_Info_desc));
//if (encrypt)
// removed to test idup_end_protect
//{
// build the recipient list
// create_nameset (recips, &minor_status, &target_info.targ_names, 1);
//}

if (!simple) // Should always be 'GP' as that is GUI trigger used in
            // this prototype to call PGP clear signing mechanism
{
    // original S/MIME simplified functions now used for PGP clear
    // signing

    if (evidence) // Not Yet Applicable
    {
        result = GSS_S_FAILURE;
        IDUPERR("pgp_protect_message. No PGP-based evidence services yet
        implemented");
    }

    /*****
    *
    * This code not required for PGP clear signing mech but
    * it is left in here to facilitate the implementation
    * of IDUP-PGP based EVIDENCE services.
    *
    *****/

    // Evidence Only, use the EV simplified calls

```



```

        1, // form_complete_evidence
        1, // include_data_in_token
        &req,
        &minor_status,
        &pidu_buffer,
        NULL ); // evidence check, same as
                idu_buffer for S/MIME
    IDUPERR("idup_ev_singlebuffer_generate");

    ***** end of commented out evidence code *****/
    }
    else
    {
// Protection Only, use the SE calls

    memset (&protect_options, 0, sizeof (protect_options));

    switch (encrypt) // Set via GUI, results indeterminate if
    {
        // GP (PGP) and encryption services selected.
    case 3:
        protect_options.Protect_Operation = encrypt_then_sign;
        break;
    case 2:
        protect_options.Protect_Operation = sign_and_encrypt;
        break;
    case 1:
        protect_options.Protect_Operation = encrypt_only;
        break;
    default: // Used for prototype
        protect_options.Protect_Operation = clear_sign_only;
        break;
    }

    protect_options.sign_qop_alg = sig_qop; //Only MD5(1) and SHA-
                                           //1(3) valid for PGP
    protect_options.enc_qop_alg = enc_qop; //No valid PGP enc
                                           //algs. PGP offers
                                           //CAST (default), 3DES
                                           //and IDEA. In a full
                                           //implementation the
                                           //GUI would have to be
                                           //modified to reflect
                                           //and offer the 3
                                           //encryption algorithms
                                           //available in PGP6.

    result = idup_se_singlebuffer_protect (
        env_handle,
        &protect_options,
        encrypt ? &target_info : NULL, /* target_info,
                                           n/a when signing */
        &idu_buffer, // Using to pass pInFilename to
                    // pgp_sign_message

```

```

        &minor_status,
        encrypt ? &pidu_buffer : NULL, /* pidu_buffer
                                         - not used when
                                         clear signing*/
        encrypt ? NULL : &pidu_buffer); /* sig_token -
                                         only used when
                                         clear-signing*/

// Clear pInFilename so gss_release_buffer OK
memset (&idu_buffer, 0, sizeof (idu_buffer));

IDUPERR("idup_se_singlebuffer_protect for PGP");
    }
    else // Original GP code deleted, this block should never be executed
    {
/*
Send error code.  PGP was called using what was originally the "GP
icon".  Execution should never reach this code block.  There is no
other option for this prototype as currently implemented.  Once other
PGP-based services are implemented, this block may be used.
*/
        result = GSS_S_FAILURE;
        IDUPERR("pgp_protect_message - no GP services");

    }

/* write the pidu to disk */
fp = fopen (out, "wb");
fwrite (pidu_buffer.value, 1, pidu_buffer.length, fp);
fclose (fp);

result = gss_release_buffer (&minor_status, &pidu_buffer);
IDUPERR("gss_release_buffer");

result = gss_release_buffer (&minor_status, &idu_buffer);
IDUPERR("gss_release_buffer");

result = idup_abolish_env (&env_handle, &minor_status);
IDUPERR("idup_abolish_env");

result = gss_release_cred (&minor_status, &cred_handle);
IDUPERR("gss_release_cred");

result = gss_release_name (&minor_status, &gss_name);
IDUPERR("gss_release_name");

return 1; /* success */
}

```

```
/*
 * Filename: idup_abolish_env_chafe.c
 *
 * Last Edited: Feb, 2000
 *
 * Author: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This is a modified version of the original idup_abolish_env.c file.
 * The MUA calls this function to abolish an environment; regardless of
 * the protocol used. On this side of the IDUP API, this function
 * checks which protocol is being used and then calls the appropriate
 * function to abolish the environment. The file also contains
 * definitions for CSP (protocol) specific abolish functions. A new
 * function, pgpapi_abolish_env, was created and is included here.
 */

#include "idup.h"

OM_uint32 mscapi_abolish_env (idup_env_id_t* env_handle, OM_uint32*
                             minor_status)
{
    if (!CryptReleaseContext ((*env_handle)->CryptProvHandle, 0))
    {
        *minor_status = GetLastError ();
        return GSS_S_FAILURE;
    }
    return GSS_S_COMPLETE;
}

OM_uint32 cdsa_abolish_env (idup_env_id_t* env_handle, OM_uint32*
                             minor_status)
{
    return GSS_S_FAILURE;
}

OM_uint32 idup_abolish_env (idup_env_id_t* env_handle, OM_uint32*
                             minor_status)
{
    OM_uint32 result;

    if (!*env_handle)
        return (IDUP_S_NO_ENV);
    if ((*env_handle)->enc_qop == 4) // HACK - using enc_qop to signal PGP
                                    // mech
        result = (*env_handle)->pgp_abolish_env (env_handle, minor_status);
    else // SMIME mech
        result = (*env_handle)->smime_abolish_env (env_handle,
                                                    minor_status);

    free (*env_handle);
    return (result);
}
```

```
}

OM_uint32 pgpapi_abolish_env (idup_env_id_t* env_handle, OM_uint32*
                             minor_status)
{
/*
This function does nothing.  To conform to the IDUP API/mechanism, a
function is needed to simulate a PGP abolish environment function.  If
the PGP mech was implemented via an API (through the PGP SDK), then
there is likely a call to be made here; similar to the
mscapi_abolish_env for MSCAPI and those CSPs.  Because we are using PGP
at the command line level (no API), the environment is automatically
abolished by the PGP application as part of the signature service.
Thus, this function is merely a shell or place marker indicating where
the "abolish" routines would normally be called, if they were needed.
*/

    return GSS_S_COMPLETE;
}
```



```

/*
 * Filename: idup_establish_env_chafe.c
 *
 * Last Edited: Feb, 2000
 *
 * Author: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This is a modified version of the original idup_establish_env.c
 * file. The MUA calls this function to establish an environment;
 * regardless of the protocol used. This function checks minimal
 * requirements (e.g. must have cred handle or user name), sets the
 * requested policy (policies are the rules used in protecting messages
 * - i.e. CSP specific) to the default value unless otherwise
 * specified, then checks this value and sets a number of CSP specific
 * variables within the environment structure. A policy identifier and
 * appropriate variables were created for the PGP CSP. The
 * environmental structure was also modified to accommodate PGP
 * specific function pointers. The pointers are set here but called
 * later on in the message protection processing.
 */

#include "idup.h"

#define CAPI_CRYPTOAPIO 1
#define PGP_CRYPTOAPIO 2 // Chafe mod
// Declare new pgp functions
extern int pgp_sign_message(const char *in);
extern OM_uint32 pgpapi_abolish_env (idup_env_id_t* env_handle,
                                     OM_uint32* minor_status);

gss_OID_desc GSS_C_BASE_OID = { 1, (void *) "\001" };
gss_OID_desc GSS_C_ENHANCED_OID = { 1, (void *) "\002" };
gss_OID_desc GSS_C_FORTEZZA_OID = { 1, (void *) "\003" };
gss_OID_desc GSS_C_PGP_OID = { 1, (void *) "\004" }; // Chafe mod

/* compares two OIDs and returns 0 if they are equal, -1 otherwise */
int oidcmp (gss_OID a, gss_OID b)
{
    if (a->length == b->length && memcmp (a->elements, b->elements, a->
                                            length) == 0)
        return 0;
    return -1;
}

OM_uint32 idup_establish_env (
    gss_cred_id_t      claimant_cred_handle,
    gss_OID             req_mech_type,
    gss_OID             req_policy,
    time_t              policy_time,
    gss_OID_set          req_services,

```

```

OM_uint32*          minor_status,
idup_env_id_t*      env_handle,
gss_OID*            actual_mech_type,
gss_OID*            actual_policy,
time_t*             actual_policy_time,
gss_OID_set         ret_services
)
{
    char *provName;
    int provType, cryptoType;
    gss_OID_desc user_policy;
    char container[64];

    *minor_status = 0;

    if (!claimant_cred_handle)
        return GSS_S_NO_CRED;

    if (!req_policy)
    {
        /* use the default policy for this user */
        user_policy.length = ((CertInfo *) claimant_cred_handle->name->sign->
                               data)->policy.len;
        user_policy.elements = ((CertInfo *) claimant_cred_handle->name->
                                sign->data)->policy.val;
        req_policy = &user_policy;
    }

    *env_handle = malloc (sizeof (struct smime_env_id_t));
    (*env_handle)->credential = claimant_cred_handle;

    strncpy (container, claimant_cred_handle->name->email, sizeof
              (container) - 1);
    container[sizeof (container) - 1] = 0;

    if (!oidcmp (req_policy, &GSS_C_BASE_OID))
    {
        provName = "Microsoft Base Cryptographic Provider v1.0";
        provType = PROV_RSA_FULL;
        cryptoType = CAPI_CRYPTAPI;
        (*env_handle)->enc_qop = 1;
        (*env_handle)->sig_qop = 1;
    }
    else if (!oidcmp (req_policy, &GSS_C_ENHANCED_OID))
    {
        provName = "Microsoft Enhanced Cryptographic Provider v1.0";
        provType = PROV_RSA_FULL;
        cryptoType = CAPI_CRYPTAPI;
        (*env_handle)->enc_qop = 2;
        (*env_handle)->sig_qop = 1;
    }
    else if (!oidcmp (req_policy, &GSS_C_FORTEZZA_OID))

```

```

{
    provName = NULL;
    provType = PROV_FORTEZZA;
    cryptoType = CAPI_CRYPTAPI;
    (*env_handle)->enc_qop = 3;
    (*env_handle)->sig_qop = 3;
    /* need to pad the container name to 32 chars */
    sprintf (container, "%-32s", claimant_cred_handle->name->
            email);
}
else if (!oidcmp (req_policy, &GSS_C_PGP_OID))
{
    provName = "PGP Freeware Version 6.5.1 'CSP'";
    provType = 0; //Not used due to non programmatic interface with PGP
                //mech
    cryptoType = PGP_CRYPTAPI;
    (*env_handle)->enc_qop = 4; // HACK - used to flag PGP mech in
                                // environment struct (the mech type in
                                // the cred struct was not implemented
                                // in the Plug-in). This flag is checked
                                // in idup_se_singlebuffer_protect
    (*env_handle)->sig_qop = 0; // Not used
}
else
    return (GSS_S_FAILURE);
if (cryptoType = PGP_CRYPTAPI) // establish PGP environ
{
    (*env_handle)->pgp_abolish_env = pgpapi_abolish_env;
    (*env_handle)->pgp_sign = pgp_sign_message;

/*****
*The only PGP IDUP-based services implemented in this prototype are
*clear signing and abolishing the pgp environment. The abolish routine
*was only required so it could replace the S/MIME based abolish call.
*Consequently, the only function pointers initialized in the env struct
*are for "pgp_sign" and "pgpapi_abolish_env".
*
*Both function types were added to the env struct defined in Mech.h.
*
*If there is an equivalent establish environ routine in the PGP SDK, it
*would be called here to "log into" the PGP mech. This is not done in
*this prototype.
*
*****/

}
else // original smime code for smime environment
{
    (*env_handle)->smime_abolish_env = mscapi_abolish_env;
#ifdef MSCAPI2
    (*env_handle)->smime_sign = mscapi_sign;
    (*env_handle)->smime_encrypt = mscapi_encrypt;

```

```
(*env_handle)->smime_encrypt_then_sign = mscapi_encrypt_then_sign;
(*env_handle)->smime_decode = smime_decode;
#else
    (*env_handle)->smime_sign = tp_sign;
    (*env_handle)->smime_encrypt = tp_encrypt;
    (*env_handle)->smime_decode = tp_decode;
    (*env_handle)->smime_encrypt_then_sign = tp_encrypt_then_sign;
#endif /* MSCAPI2 */

if (!CryptAcquireContext (&(*env_handle)->CryptProvHandle,
    container,
    provName,
    provType,
    0))
{
    *minor_status = GetLastError ();
    return GSS_S_FAILURE;
}
// End of original (non-PGP) code
*minor_status = 0;

return GSS_S_COMPLETE;
}
```

```

/*
 * Filename: idup_se_singlebuffer_protect_chafe.c
 *
 * Last Edited: Feb, 2000
 *
 * Author: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This is a modified version of the original idup_se_singlebuffer
 * _protect.c file. The MUA calls this function to apply the SE (Sign
 * and Encrypt) IDUP services to a message; regardless of the protocol
 * used. These service are formally titled confidentiality with
 * integrity (CONF) and data origin authentication with integrity
 * (DOA).
 *
 * This function checks minimal requirements (e.g. must have an
 * environment handle), extracts the signature and encryption algorithm
 * identifiers from the environment structure and then calls the
 * appropriate function according to the service requested. In calling
 * these functions, the correct signature and/or encryption algorithm
 * is specified by passing the * identifiers previously assigned to the
 * environment (extracted from the environment structure). Note also
 * that this function depends upon a calling structure that must be
 * correctly initialised before this function is called. That is, the
 * structure entry (function pointer) corresponding to message signing
 * will be called whenever these services are required. The exact
 * function that is executed is unknown to idup_se_singlebuffer_protect.
 * The initialisation of this calling structure is done while
 * establishing the IDUP environment.
 *
 * To enable the PGP-based clear signature service, the calling
 * structure was augmented to include new entries for the PGP routines
 * and a check was added to determine which protocol to use before
 * calling the PGP or S/MIME routines.
 */

#include "idup.h"

OM_uint32 idup_se_singlebuffer_protect (
    idup_env_id_t          env_handle,
    IDUP_Protect_Options    Protect_Options,
    IDUP_Target_Info        Target_Info,
    const gss_buffer_t      idu_buffer,
    OM_uint32*              minor_status,
    gss_buffer_t            pidu_buffer,
    gss_buffer_t            sig_token
)
{
    LPSTR sigalg, encalg;

```

```

*minor_status = 0;

if (!env_handle)
    return (IDUP_S_NO_ENV);

sigalg = smime_GetSigOID (Protect_Options->sign_qop_alg ?
                          Protect_Options->sign_qop_alg : env_handle->
                          sig_qop);

if (!sigalg)
    return (GSS_S_BAD_QOP);
encalg = mscapi_GetEncOID (Protect_Options->enc_qop_alg ?
                            Protect_Options->enc_qop_alg : env_handle->
                            enc_qop);

if (!encalg)
    return (GSS_S_BAD_QOP);

if (env_handle->enc_qop == 4) // HACK - using this entry and value to
                            // signal PGP mech
{

    switch (Protect_Options->Protect_Operation)
    {
    case sign_only: // Not yet implemented
    case clear_sign_only:
        return (env_handle->pgp_sign ((const char *)idu_buffer->value));
    case encrypt_only: //Not yet implemented
    case sign_and_encrypt: // Not yet implemented
    case sign_then_encrypt_full: // Not yet implemented
    case encrypt_then_sign: // Not yet implemented
        break;
    }

} // End of PGP mech code

else // SMIME mech (original code)

{

    switch (Protect_Options->Protect_Operation)
    {
    case sign_only:
        return (env_handle->smime_sign (env_handle, sigalg, idu_buffer,
                                        minor_status, pidu_buffer, 0, NULL,
                                        0));

    case clear_sign_only:
        return (env_handle->smime_sign (env_handle, sigalg, idu_buffer,
                                        minor_status, sig_token, 1, NULL,
                                        0));

    case encrypt_only:
        if (!Target_Info || Target_Info->targ_names->count == 0)
            return GSS_S_FAILURE;
        return (env_handle->smime_encrypt (env_handle,

```

```
                                encalg,
                                NULL,
                                Target_Info,
                                idu_buffer,
                                minor_status,
                                pidu_buffer,
                                NULL,
                                0));

case sign_and_encrypt:
case sign_then_encrypt_full:
    if (!Target_Info || Target_Info->targ_names->count == 0)
        return GSS_S_FAILURE;
    return (env_handle->smime_encrypt (env_handle,
                                        encalg,
                                        sigalg,
                                        Target_Info,
                                        idu_buffer,
                                        minor_status,
                                        pidu_buffer,
                                        NULL,
                                        0));

case encrypt_then_sign: /* encrypt then sign ciphertext */
    if (!Target_Info || Target_Info->targ_names->count == 0)
        return GSS_S_FAILURE;
    return (env_handle->smime_encrypt_then_sign (env_handle,
                                                  encalg,
                                                  sigalg,
                                                  Target_Info,
                                                  idu_buffer,
                                                  minor_status,
                                                  pidu_buffer));
}

} // End of SMIME (original code)

/* s/mime doesn't support sign_then_encrypt_data */
return GSS_S_FAILURE;
}
```

```
/*
 * Filename: mech.h
 *
 * Last Edited: Feb, 2000
 *
 * Author: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This is a modified version of the original mech.h, the IDUP
 * mechanism header file. This file contains declarations for many of
 * the functions, variables and structures used by the mechanism. One
 * of the key structures declared here is the environment structure
 * (smime_cred_id_t). This structure was modified to include two
 * additional function pointers for the PGP routines.
 */

#ifndef mech_h
#define mech_h

struct smime_cred_id_t
{
    gss_name_t name; /* name this credential belongs to */
};

struct smime_env_id_t
{
    // used as temp storage for the multibuffer calls
    IDUP_Protect_Options Protect_Options;
    IDUP_Target_Info Target_Info;
    gss_buffer_t IDU_Buffer;

    IDUP_Special_Conditions Special_Conditions;

    // state vars for the EV multibuffer calls
    int nr_operation;
    IDUP_Nr_Options Nr_Options;
    int form_complete_evidence;
    int include_data_in_token;
    IDUP_Request_Features Request_Features;
    gss_buffer_t evidence_check;

    struct smime_cred_id_t *credential; /* credentials used to establish
                                         environment */

    /* MS CryptoAPI 2.0 */
    HCRYPTPROV CryptProvHandle;

    OM_uint32 enc_gop; /* default encryption gop */
    OM_uint32 sig_gop; /* default signature gop */
};
```



```

OM_uint32 (*smime_abolish_env) (struct smime_env_id_t** env_handle,
                                OM_uint32*          minor_status);

OM_uint32 (*smime_encrypt_then_sign) (
    struct smime_env_id_t * env_handle,
    LPSTR enc_alg,
    LPSTR sig_alg,
    IDUP_Target_Info      Target_Info,
    gss_buffer_t          idu_buffer,
    OM_uint32 *           minor_status,
    gss_buffer_t          pidu_buffer);

OM_uint32 (*smime_encrypt) (
    struct smime_env_id_t *env_handle,
    LPSTR enc_alg,
    LPSTR sig_alg,
    IDUP_Target_Info      Target_Info,
    gss_buffer_t          idu_buffer,
    OM_uint32 *           minor_status,
    gss_buffer_t          pidu_buffer,
    PCRYPT_ATTRIBUTE      auth_attr,
    DWORD                 auth_attr_len);

OM_uint32 (*smime_sign) (
    struct smime_env_id_t *env_handle,
    LPSTR sig_alg,
    const gss_buffer_t     idu_buffer,
    OM_uint32*             minor_status,
    gss_buffer_t           pidu_buffer,
    int                    detached,
    PCRYPT_ATTRIBUTE       auth_attr,
    DWORD                  auth_attr_len);

OM_uint32 (*smime_decode) (
    struct smime_env_id_t *env_handle,
    gss_buffer_t           pidu_buffer,
    OM_uint32              minor_status,
    gss_buffer_t           idu_buffer,
    IDUP_Services_To_Receive services,
    IDUP_Prot_Information Prot_information);

/* Chafe mod - PGP Prototype 'API' */

OM_uint32 (*pgp_sign) (const char *          in);

OM_uint32 (*pgp_abolish_env) (struct smime_env_id_t** env_handle,
                               OM_uint32*          minor_status);

};

```

```
/*
 * OIDs for req_policy argument to idup_establish_env() for selecting
 * among CSPs
 */

extern gss_OID_desc GSS_C_BASE_OID;

extern gss_OID_desc GSS_C_ENHANCED_OID;
extern gss_OID_desc GSS_C_FORTezza_OID;
extern gss_OID_desc GSS_C_CDSA_INTEL_OID;

extern gss_OID_desc ProofOfReceiptOid;
extern gss_OID_desc ProofOfDeliveryOid;
extern gss_OID_desc ProofOfApprovalOid;
extern gss_OID_desc ProofOfCreationOid;
extern gss_OID_desc ProofOfSenderOid;
extern gss_OID_desc ProofOfOriginOid;

extern gss_OID_desc PER_CONF;
extern gss_OID_desc PER_CONF_FULL;
extern gss_OID_desc PER_DOA;
extern gss_OID_desc PER_DOA_CIPH;
extern gss_OID_desc PER_POO;
extern gss_OID_desc PER_POD;
extern gss_OID_desc PER_POS;
extern gss_OID_desc PER_POC;
extern gss_OID_desc PER_POR;
extern gss_OID_desc PER_POA;

extern gss_OID_desc REC_CONF;
extern gss_OID_desc REC_CONF_FULL;
extern gss_OID_desc REC_DOA;
extern gss_OID_desc REC_DOA_CIPH;
extern gss_OID_desc REC_POO;
extern gss_OID_desc REC_POD;
extern gss_OID_desc REC_POS;
extern gss_OID_desc REC_POC;
extern gss_OID_desc REC_POR;
extern gss_OID_desc REC_POA;

extern LPSTR szOID_Evidence;
extern LPSTR szOID_EvidenceRequest;

/*
 * MSCAPI Functions
 */

OM_uint32 smime_generate_evidence (PCRYPT_ATTRIBUTE a,      /* out */
                                   gss_OID oid,             /* out */
                                   int duration,            /* in/optional */
                                   int identifier,          /* in/optional */
                                   ...)
```

```

                                gss_buffer_t  received_token
                                /*in/optional */);

OM_uint32 smime_generate_evidence_request (
    PCRYPT_ATTRIBUTE a,
    gss_OID oid,
    idup_name_set from,
    idup_name_set to,
    int include_received_token_in_evidence,
    int identifier);

OM_uint32 smime_decode (struct smime_env_id_t      *env_handle,
    gss_buffer_t      pidu_buffer,
    OM_uint32 *      minor_status,
    gss_buffer_t      idu_buffer,
    IDUP_Services_To_Receive services,
    IDUP_Prot_Information prot_information);

OM_uint32 mscapi_abolish_env (struct smime_env_id_t** env_handle,
    OM_uint32* minor_status);

OM_uint32 mscapi_sign ( struct smime_env_id_t  *env_handle,
    LPSTR sig_alg,
    const gss_buffer_t idu_buffer,
    OM_uint32* minor_status,
    gss_buffer_t pidu_buffer,
    int detached,
    PCRYPT_ATTRIBUTE auth_attr,
    DWORD auth_attr_len);

OM_uint32 mscapi_encrypt (struct smime_env_id_t *env_handle,
    LPSTR enc_alg, LPSTR sig_alg,
    IDUP_Target_Info Target_Info,
    gss_buffer_t idu_buffer,
    OM_uint32 * minor_status,
    gss_buffer_t pidu_buffer,
    PCRYPT_ATTRIBUTE auth_attr,
    DWORD auth_attr_len);

OM_uint32 mscapi_encrypt_then_sign (struct smime_env_id_t * env_handle,
    LPSTR enc_alg,
    LPSTR sig_alg,
    IDUP_Target_Info Target_Info,
    gss_buffer_t idu_buffer,
    OM_uint32 * minor_status,
    gss_buffer_t pidu_buffer);

OM_uint32 mscapi_import_cert (ASN_UNDECODED_BLOCK *name,
    HCERTSTORE hStore,
    HCRYPTPROV hProv,
    PCCERT_CONTEXT *pCert);

```

```

OM_uint32 mscapi_import_recips (HCERTSTORE hstore,
                                IDUP_Target_Info target,
                                PCCERT_CONTEXT **recips);

/*
 * TIS/PKCS7 library with MS CryptoAPI 1.0
 */
OM_uint32 tp_decode (
    struct smime_env_id_t*      env_handle,
    gss_buffer_t                pidu_buffer,
    OM_uint32 *                 minor_status,
    gss_buffer_t                idu_buffer,
    IDUP_Services_To_Receive    services,
    IDUP_Prot_Information        prot_information);

OM_uint32 tp_sign (
    struct smime_env_id_t*      env_handle,
    LPSTR                        sig_alg,
    const gss_buffer_t          idu_buffer,
    OM_uint32 *                 minor_status,
    gss_buffer_t                pidu_buffer,
    int                          detached,
    PCRYPT_ATTRIBUTE            auth_attr,
    DWORD                        auth_attr_len);

OM_uint32 tp_encrypt (
    struct smime_env_id_t*      env_handle,
    LPSTR                        enc_alg,
    LPSTR                        sig_alg,
    IDUP_Target_Info            Target_Info,
    gss_buffer_t                idu_buffer,
    OM_uint32 *                 minor_status,
    gss_buffer_t                pidu_buffer,
    PCRYPT_ATTRIBUTE            auth_attr,
    DWORD                        auth_attr_len);

OM_uint32 tp_encrypt_then_sign (
    struct smime_env_id_t *      env_handle,
    LPSTR enc_alg,
    LPSTR sig_alg,
    IDUP_Target_Info            Target_Info,
    gss_buffer_t                idu_buffer,
    OM_uint32 *                 minor_status,
    gss_buffer_t                pidu_buffer);

/*
 * Auxillary functions
 */
LPSTR mscapi_GetEncOID (OM_uint32 qop);
LPSTR smime_GetSigOID (OM_uint32 alg);

gss_buffer_t smime_append_buffer (gss_buffer_t out, gss_buffer_t in);

```

```
int oidcmp (gss_OID, gss_OID);
gss_OID dup_gss_OID (gss_OID oid);
gss_OID check_evidence_type (ASN_OBJECT_ID *asn);

/* convert from ASN.1 SET OF IA5String to idup_name_set */
idup_name_set extract_names (EvidenceTarget *asn);

/*
 * database functions
 */

OM_uint32 gn_fetch_dn (GssName **, gss_buffer_t, const char *);
OM_uint32 gn_fetch (GssName **, const char *, const char *);
char *smime_db_path (void);

/* find a certificate in the database by subject name only (needed for
FORTEZZA to find the issuer cert for the P,Q and G values */

OM_uint32 gn_fetch_in (Certificate **cert, Name *subject, const char
*dbfile);

struct list_t
{
    char *data;
    struct list_t *next;
    unsigned int encrypt : 1;
    unsigned int evidence_from : 1;
    unsigned int evidence_to : 1;
};

OM_uint32 gn_store (const char *dbfile, const char *container, const
char *email, gss_buffer_t policy, gss_buffer_t sign, gss_buffer_t
keyx);

int gn_CspType (const char *);

#endif /* mech_h */
```

```

/*
 * Filename: smime_sign_chafe.c
 *
 * Last Edited: Feb, 2000
 *
 * Author: Michael Elkins @ Network Associates, Inc.
 *
 * Modified last By: Kevin Chafe
 *
 * This is a modified version of the original smime_sign.c. This file
 * contained the definitions for the functions that called the CSP; to
 * begin cryptographic operations. The original mechanism used MSCAPI
 * to access both RSA and Fortezza libraries (CSPs). Thus, there was
 * only one MSCAPI call in this function. The multi-protocol prototype
 * uses two CSPs: MSCAPI and the PGP command line (i.e. DOS command
 * line). A second function was created to work within the IDUP API
 * paradigm, extracting any required parameters from the IDUP
 * structures and then invoking the PGP clear signing service on the
 * file containing the message that Eudora is processing.
 */

#include "idup.h"

#include <sys/stat.h>
#include <stdio.h>

LPSTR smime_GetSigOID (OM_uint32 alg)
{
    switch (alg)
    {
        case 1:
            return (szOID_RSA_MD5);
        case 2:
            return (szOID_RSA_MD2);
        case 3:
            return (szOID_INFOSEC_mosaicUpdatedInteg);
        default:
            return (NULL);
    }
    /* not reached */
}

OM_uint32 mscapi_sign (idup_env_id_t          env_handle,
                       LPSTR                  sig_alg,
                       const gss_buffer_t     idu_buffer,
                       OM_uint32*             minor_status,
                       gss_buffer_t           pidu_buffer,
                       int                     detached,
                       PCRYPT_ATTRIBUTE       auth_attr,
                       DWORD                   auth_attr_len)
{
    HCERTSTORE hStore;

```

```

PCCERT_CONTEXT pSignerCert;
PCCERT_CONTEXT MsgCertArray[1];
CRYPT_ALGORITHM_IDENTIFIER HashAlgorithm;
CRYPT_SIGN_MESSAGE_PARA signpara;
BYTE *MessageArray[1];
DWORD MessageLenArray[1];
BOOL fReturn;

OM_uint32 result;

// open an in-memory certificate store for temporary use
hStore = CertOpenStore (sz_CERT_STORE_PROV_MEMORY,
                       X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
                       env_handle->CryptProvHandle,
                       CERT_STORE_NO_CRYPT_RELEASE_FLAG,
                       (unsigned long) NULL); // FOO - what is the
                                           //real type here?

result = mscapi_import_cert (&((CertInfo *)env_handle->credential->
                             name->sign->data)->certificate, hStore,
                             env_handle->CryptProvHandle,
                             &pSignerCert);
if (result != GSS_S_COMPLETE)
{
    CertCloseStore (hStore, CERT_CLOSE_STORE_CHECK_FLAG);
    return (result);
}

memset (&HashAlgorithm, 0, sizeof (HashAlgorithm));
HashAlgorithm.pszoObjId = sig_alg;

// certificates to include in the signed message
MsgCertArray[0] = pSignerCert;

memset (&signpara, 0, sizeof (signpara));
signpara.cbSize = sizeof (signpara);
signpara.dwMsgEncodingType = X509_ASN_ENCODING | PKCS_7_ASN_ENCODING;
signpara.pSigningCert = pSignerCert;
signpara.HashAlgorithm = HashAlgorithm;
signpara.cMsgCert = 1;
signpara.rgpMsgCert = MsgCertArray;
signpara.cAuthAttr = auth_attr_len;
signpara.rgAuthAttr = auth_attr;

MessageArray[0] = idu_buffer->value;
MessageLenArray[0] = idu_buffer->length;

// determine how much memory is required for the output P-IDU */
fReturn = CryptSignMessage (&signpara,
                             detached,
                             1,
                             MessageArray,

```

```

    MessageLenArray,
    NULL,
    &pidu_buffer->length);
if (!fReturn)
{
    return GSS_S_FAILURE;
}

pidu_buffer->value = malloc (pidu_buffer->length);

// now really sign the message
fReturn = CryptSignMessage (&signpara,
    detached,
    1,
    MessageArray,
    MessageLenArray,
    pidu_buffer->value,
    &pidu_buffer->length);
if (!fReturn)
{
    return GSS_S_FAILURE;
}

// clean up the mess
fReturn = CertFreeCertificateContext (pSignerCert);
if (!fReturn)
{
    return GSS_S_FAILURE;
}

fReturn = CertCloseStore (hStore, CERT_CLOSE_STORE_CHECK_FLAG);
if (!fReturn)
{
    return GSS_S_FAILURE;
}

return GSS_S_COMPLETE;
}

/*****
/* CHAFE MOD - PGP PROTOTYPE FUNCTION*/

int pgp_sign_message (const char *in)
{
    int status = 0; // HACK in = global_pInFilename passed through
                  // idu_buffer->value; this is how get pInFilename
                  // from DoTransmission through the IDUP API.

    char command[62] = "pgp -s
";           // must be long enough for file path
    char *command_ptr = &command[7]; // Initialize ptr to start just after
                                     // "pgp -s "

```



```
while (*command_ptr != '\0')
{
    *command_ptr = *in; // fill in actual value for pInFilename.tmp
                        // (watch length of command buffer for
                        // "c:\windows\temp\etc\pInFilename")
    ++command_ptr;      // increment ptr into command string
    ++in;               // increment ptr into file name
}
status = system(command); // returns -1 if error, otherwise whatever
PGP      // returns
if (status == -1)
    return (GSS_S_FAILURE); // failure != 0
return (GSS_S_COMPLETE); // success = 0
}
```

GLOSSARY

Access Control: Process of limiting access to the resources of an automated information system only to authorised users, programs, processes, or other systems.

Access Control Policy: The set of rules that define the conditions under which an access may take place.

Application Programming Interface (API): An interface between two software implementations, across which services are provided.

Asymmetric Keys: In public key or asymmetric cryptosystems, key pairs are used. One key is used to encrypt and its mate is used to decrypt (i.e., public and private keys).

Attachment: Any file that accompanies an e-mail message. When the e-mail message is sent, both the text of the message and the attached file are transmitted.

Authentication: Security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's eligibility to receive specific categories of information.

Availability: The property of being accessible and usable upon demand by an authorised entity.

Bandwidth: A measurement of how much data can travel over a transmission medium (such as a telephone line or computer network) in a given period of time. Bandwidth is measured in bits per second (bps). The larger the bandwidth, the more data the network can handle.

Body: The text of an e-mail message. The body of a message follows the header information.

Browser: A client application program used to search through information provided by a specific type of server. A web browser lets you view and navigate through information on the World Wide Web of the Internet.

Cipher: Mathematical function used to encrypt and decrypt a set of plaintext according to a predetermined key (also called a cryptographic algorithm).

Ciphertext: Enciphered information, the output of a cipher.

Clear Text: Intelligible data that is not encrypted or encoded, also called plaintext.

Client: Any application program used to retrieve information from a server. Internet clients include World Wide Web browsers, Usenet newsreaders, and e-mail programs.

Client/server: The relationship between two application programs. One program, the server, is responsible for servicing requests from the other program, the client.

Confidentiality: Assurance that information is not disclosed to unauthorised entities or processes.

Credentials: Information, passed from one entity to another, that is used to establish the sending entity's access rights.

Context: See security context.

Cryptographic Algorithm: Well-defined procedure or sequence of rules or steps used to produce cipher text from plain text and vice versa using a pre-determined key.

Cryptographic Context: See security context.

Cryptographic Services Provider: Independent modules that perform cryptographic and/or cryptography support. CSPs are accessed via a cryptographic API.

Cryptography: The study of the mathematical principles, means, and methods used to provide information security services such as confidentiality, entity authentication, data

origin authentication and data integrity. Also, the art and science of securing communications.

Data integrity: Condition that exists when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.

Data origin authentication: Corroboration that the source of data is as claimed.

Data structure: The format in which the cryptographic material and/or attributes are stored.

Decipher: Convert enciphered text to the equivalent plain text by means of a cipher (i.e. decryption).

Decode: Convert encoded text to its equivalent plain text by means of a code.

Decrypt: Opposite of encrypt (see decipher).

Disconnected/Resynchronisation Mode: A mail-access mode where mail is synchronised between a server and a client computer. By synchronising mail on the server, users can access their own mail from any computer that has access to the server where the mail is stored.

Domain Name: The part of an e-mail address to the right of the @ sign. A domain name is a name that uniquely identifies an Internet host. Domain names are used to describe the source or destination of an Internet connection: for example “sce.carleton.ca” and “hotmail.com”.

Domain Name System (DNS): A behind-the-scenes Internet service that translates Internet domain names to their corresponding IP addresses, and vice versa.

Download: The electronic transfer of information from a host (or server) computer to your (a client) computer.

Electronic Mail (E-mail): A system that enables the exchange of messages between network users or groups of users.

Electronic Mail Address: The location where you send your e-mail messages. Internet addresses (for example, class_list@sce.carleton.ca) consist of:

- A user name (for example, class_list: this is actually the name of a mail distribution list)
- An @ sign
- The domain name (for example, sce.carleton.ca)

Encipher: Convert plain text to equivalent cipher text by means of a cipher and a pre-determined key.

Encode: Convert plain text to equivalent encoded text by means of a code. For example, MIME encoding translates a non-text (e.g., binary) or non-U.S. ASCII (e.g., Cyrillic) file into a text (usually U.S. ASCII) format suitable for sending through the Internet e-mail system.

Encrypt: Generic term encompassing encipher.

Forward: To pass along a received message to another e-mail address.

File Transfer Protocol (FTP): File Transfer Protocol, an Internet protocol that allows a user on one host to transfer files to and from another host.

File Transfer Protocol Site: An Internet server that has been set up to accept FTP connections.

Gateway: Software that translates data from the standards of one system to the standards of another. For example a gateway might exchange and convert Internet e-mail to X.400 e-mail.

Header: Part of an e-mail message that precedes the body of the message and provides the message originator, date, and time.

Hypertext Markup Language (HTML): Hypertext Markup Language, a markup language used for creating “rich” documents on the Internet. These documents can include formatting attributes such as bold, indentations and headings. They can also include references (or “links”) to other documents and references to embedded images, sounds, applets, and other multimedia. Although HTML was originally used on the Web, it is now being used in other areas on the Internet, such as e-mail.

Hypertext Transfer Protocol (HTTP): A protocol used by computers on the Internet to exchange information, such as a Web page. Your request to obtain information from the Web is sent using HTTP; the information you requested is sent back using HTTP, as well. When you see a URL that starts “http://”, it indicates that the address points to a Web page.

Internet Message Support Protocol (IMSP): A protocol designed to support functions necessary to implement a large-scale IMAP4-based infrastructure geared to highly mobile users. IMSP comprises three major components:

- Storage for client configuration information
- Storage for user address books
- Support for mailbox distribution and replication

Internet: A global networking system that connects millions of computers, people, software programs, databases, and files. For most of its existence, the Internet was primarily an academic and research network. In recent years, the Internet has become more consumer- and commercially oriented.

Internet Message Access Protocol (IMAP): A complex Internet protocol used by mail clients to access mail stored on servers. While IMAP-based applications can operate in offline mode, they typically operate in online or disconnected/resynchronisation mode.

Intranet: A small-scale adaptation of Internet technologies used to share information within a business or organisation. An Intranet provides an easy-to-use method for managing messaging and information on a company-wide basis.

Internet Protocol (IP): The Internet standard that provides a common means of communicating over dissimilar networks.

Internet Protocol Address: The unique numeric address of a computer connected to the Internet. Represented in “dotted decimal notation,” a series of four numbers separated by periods. For instance, “188.6.104.1” is an IP address.

Internet Service Provider (ISP): A business that sells Internet connections to individuals and businesses.

Key: Information (usually a sequence of random or pseudorandom binary digits) used with a cryptographic algorithm to specify a specific mapping from the plaintext space to the ciphertext space and/or back. Each cipher has an associated set of possible key values, called the keyspace.

Key Management: Process by which keys are generated, stored, protected, transferred, loaded, used, and destroyed.

Mail Access Protocol: A carefully standardised set of commands and responses responsible for the communication between the mail client and mail server.

Mail Client: The software used to read, file, send, and otherwise process e-mail, typically running on a desktop computer (also called a Mail User Agent).

Mailbox: A file where your e-mail messages are stored.

Mail Gateway: A computer that connects similar or dissimilar e-mail systems and translates and transfers messages between them.

Mail Server: A computer typically managed by an ISP or Information Services department that handles receipt and delivery of e-mail messages. It also may store mail for the user on a temporary or permanent bases.

Multipurpose Internet Mail Extensions (MIME): An Internet standard that provides the transfer of non-text information, such as sounds and graphics, and non-U.S. English (such as Cyrillic) or Japanese, via e-mail.

Multilevel Information System Security Initiative (MISSI): A NSA program for developing and delivering data and network security products for the U.S. government.

Mail Transfer Agent (MTA): The software running on a mail server. It relays and delivers mail.

Mail User Agent (MUA): The software used to read, file, send, and process e-mail, typically running on a desktop computer (also known as the Mail Client or E-mail program).

Network File System: A network file system provides access to files stored on a server via the network. This service is usually provided by network operating systems such as Novell. NFS and AFS provide similar services for UNIX. Network file systems can work well over local networks where bandwidth is plentiful, but not so well over the public Internet or dial-up network access.

Nonrepudiation: Method by which the sender of data is provided with proof of delivery and the recipient is assured of the sender's identity, so that neither can later deny having processed the data.

Offline Mode: A mail access mode where all new mail is downloaded from the server at one time, with all subsequent processing done “offline” (with no network connection) on the user’s local computer.

Online Mode: A mail access mode where all mail access is performed “online” (with a constant connection to the mail server).

Peer-entity Authentication: The corroboration that a peer entity in an association is the one claimed.

Plaintext: See cleartext.

Point of Presence: A location from which an Internet service provider provides local dial-up Internet access. Sometimes abbreviated as POP, this is not the same as the Post Office Protocol, defined later.

Post Office Protocol (POP): POP-based applications operate in offline mode. E-mail clients that implement the POP protocol download all new mail from a mail server, terminate the network connection, and subsequently processes all mail offline at the client computer. Compare with IMAP (Internet Message Access Protocol).

Point to Point Protocol (PPP): A method that enables your computer to talk TCP via modem, usually with your service provider's system. PPP is replacing SLIP for this purpose.

Private Key: One of two associated keys used in an asymmetric cryptographic algorithm. Possession of this key is restricted, usually to only one entity.

Protocol: A specific sequence of steps involving two or more entities designed to accomplish a task. A **cryptographic protocol** is a protocol that uses cryptography to achieve a specific security objective.

Public Key: One of two associated keys used in an asymmetric or public key cryptographic algorithm. The public is made widely available, hence its name. Knowledge of the public key does not reveal the private key.

Remote Access: The ability to access a computer from outside the building in which it is housed. Remote access requires communications hardware, software, and physical links.

Request For Comments (RFC): A series of technical documents that describes the Internet suite of protocols and related experiments, begun in 1969. Note that not all RFCs describe Internet standards, but all Internet standards are published as RFCs.

Secret Key: The shared key between two entities that is used in a symmetric cryptographic algorithm. This key is used for both encryption and decryption.

Secure Sockets Layer (SSL): A protocol implemented between the application and transport layers of the OSI stack to provide confidentiality, integrity and a weak form of authentication services.

Security Aware: The caller of an API that is aware of the security functionality and parameters which may be required and/or provided by an API.

Security Context: The relationship established between peers, using credentials established locally in conjunction with each peer or received by peers via delegations.

Security Mechanism: A general term that includes protocols, algorithms and non-cryptographic techniques or procedures used to achieve security objectives.

Security Policy: The set of laws, rules, and practices that regulate how an organisation manages, protects, and distributes sensitive information.

Server: A host computer that provides resources to client computers. Usually servers are up, running and connected to the network twenty-four hours a day.

Service: The term service is used to indicate a coherent set of abstract functionality.

Serial Line Interface Protocol, (SLIP): A method that enables your computer to talk TCP via modem, usually to a service provider (compare with Point to Point Protocol).

Simple Mail Transfer Protocol (SMTP): The Internet standard that governs how e-mail messages are sent from site to site.

Symmetric keys: see secret key.

Transmission Control Protocol/Internet Protocol (TCP/IP): The basic set of protocols that transfers data between computers on the Internet or on an intranet.

Telnet: The Internet standard protocol for remote login connections. It provides a text-only interface for accessing distant computer systems.

Token: The underlying hardware or software cryptographic device.

Untethered: The ability for a user to roam from one computer to another and access mail from different computers at various locations.

Universal Resource Locator (URL): An address pointing to a certain site on the Internet. Examples are <http://www.carleton.ca> and <ftp://mac.archive.umich.edu/>.

Web Page: An HTML file available on the World Wide Web and accessible with a unique URL.

World Wide Web: A system for navigating the Internet utilising hypertext links. With a Web browser, the World Wide Web appears as a collection of text, pictures, sounds, and digital movies. Sometimes called “WWW,” “W3,” or “the Web”.