APPENDIX B (Part 1 of 2)

RSA PRESS

RSA Security's Official Guide to
CRYPTOGRAPHY

- Learn how secure data-encryption techniques work
- Protect confidential information on your network
- Get official current cryptography standards on enclosed CD-ROM

Steve Burnett & Stephen Paine

Apple 1121 (Part 1 of 2)
Apple v. USR
IPR2018-00813

RSA Security's Official Guide to Cryptography

Steve Burnett and Stephen Paine

Osborne/**McGraw-Hill**
New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

Osborne/**McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

To arrange bulk purchase discounts for sales promotions, premiums, or fund-raisers, please contact Osborne/**McGraw-Hill** at the above address. For information on translations or book distributors outside the U.S.A., please see the International Contact Information page immediately following the index of this book.

RSA Security's Official Guide to Cryptography

1234567890 FGR FGR 01987654321

Book p/n 0-07-213138-1 and CD p/n 0-07-213137-3
parts of
ISBN 0-07-213139-X

Publisher
Brandon A. Nordin

Vice President & Associate Publisher
Scott Rogers

Executive Editor
Steven Elliot

Senior Project Editor
LeeAnn Pickrell

Acquisitions Coordinator
Alexander Corona

Technical Editors
Blake Dournaee
Jessica Nelson

Copy Editor
Betsy Hardinger

Composition and Indexer
MacAllister Publishing Services, LLC

Illustrators
Michael Mueller
Beth Young
Lyssa Sieben-Wald

To Pao-Chi, Gwen, Ray, Satomi, Michelle, Alexander,
Warren, Maria, Daniel, and Julia

—Steve Burnett

To Danielle, thanks for understanding while I worked on this book

To Alexis and Elizabeth, a father could not ask for better children

—Stephen Paine

Contents

Credits

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

The ALX 300 is courtesy of Compaq Computer Corporation.

The ikey 2000 and the CryptoSwift accelerator is courtesy of Rainbow Technologies, Inc.

Data Key is courtesy of Datakey Inc.

The Java Ring is courtesy of Dallas Semiconductor Corp.

The box blue accelerator and card reader is courtesy of nCipher Inc.

The Luna CA3—Photos courtesy of Chrysalis-ITS®, Inc.

The Smarty Smart Card Reader is courtesy of SmartDisk Corporation.

The RSA SecurID Card and token are courtesy of RSA Security Inc.

The BioMouse Plus is courtesy of American Biometric Company.

The XyLoc proximity card is courtesy of Ensure Technologies.

The Trusted Time products are courtesy of Datum.

Foreword

Welcome to the second book from RSA Press, RSA Security's Official Guide to Cryptography!

As the Internet becomes a more pervasive part of daily life, the need for e-security becomes even more critical. Any organization engaged in online activity must assess and manage the e-security risks associated with this activity. Effective use of cryptographic techniques is at the core of many of these risk-management strategies. This book provides a practical guide for the use of cryptographic e-security technologies to provide for privacy, security, and integrity of an organization's most precious asset: data.

It is an exciting time for cryptography, with important technical, business, and legal events occurring in quick succession. This book can help the reader better understand the technology behind these events.

In January 2000, the United States Government announced a significant relaxation in restrictions on the export of strong cryptography. This decision has permitted U.S. companies to now compete for cryptographic business on a worldwide basis. Previously, many of the algorithms discussed in this book were treated as munitions and were subject to severe restrictions on their export from the U.S.

In September 2000, the patent on the RSA algorithm, arguably the most important patent in cryptography, expired. Now any firm or individual can create implementations of this algorithm, further increasing the pervasiveness of one of the most widespread technologies in the history of computing.

In October 2000, the United States National Institute of Standards and Technology announced its selection of the winner of the *Advanced Encryption Standard* (AES) selection process, an algorithm called Rijndael developed by two Belgian researchers. The AES algorithm is intended to replace the venerable, and increasingly vulnerable *Data Encryption Standard* (DES) algorithm. AES is expected to become the most widely used algorithm of its type in a short time.

The security technology industry has undergone explosive growth in a short period of time, with many new options emerging for the deployment of e-security techniques based on cryptography. Ranging from new developments in cryptographic hardware to the use of personal smart cards in public key infrastructures, the industry continues to increase the range of choices available to address e-security risks. This book provides the reader with a solid foundation in the core cryptographic techniques of e-security—including RSA, AES, and DES mentioned previously, and many others—and then builds on this foundation to discuss the use of these techniques in practical applications and cutting-edge technologies.

While this book does discuss the underlying mathematics of cryptography, its primary focus is on the use of these technologies in familiar, real-world settings. It takes a systems approach to the problems of using cryptographic techniques for e-security, reflecting the fact that the degree of protection provided by an e-security deployment is only as strong as the weakest link in the chain of protection.

We hope that you will enjoy this book and the other titles from RSA Press. We welcome your comments as well as your suggestions for future RSA Press books. For more information on RSA Security, please visit our web site at `www.rsasecurity.com`; more information on RSA Press can be found at `www.rsapress.com`.

Burt Kaliski
Director and Chief Scientist
RSA Laboratories
bkaliski@rsasecurity.com

Acknowledgments

The first person I'd like to thank is Stephen Paine. He did the work of putting together the original proposal and outline. Later on, he reorganized the structure to make the book better. He planned; I just wrote.

Betsy Hardinger and LeeAnn Pickrell at Osborne/McGraw Hill are the two editors who made many suggestions (most of which we accepted) to improve the language, readability, and flow of the content. Stephen Paine and I have our names on the book, but I think they deserve plenty of credit for their contributions.

Blake Dournaee of RSA did a great job of reviewing. If it hadn't been for Blake, I would be suffering from great embarrassment for a couple of mistakes he caught. Of course, any errors still residing in this book belong entirely to Stephen and me.

We received help from many people for the examples. Mark Tessin of Reynolds Data Recovery and Dennis Vanatta of 4Sites Internet Services gave me the information and screen shot for the data recovery discussion in Chapter 1. Mary Ann Davidson and Kristy Browder of Oracle helped me put together the example in Chapter 2. For the Keon example, Peter Rostin and Nino Marino of RSA were my sources.

The people at Osborne/McGraw Hill said we had complete control over the acknowledgments, so I'd like to thank some people who didn't contribute to the book so much as contributed to my career. If it hadn't been for Dave Neff at Intergraph, I don't think I would have been much of a programmer and hence never could have been successful enough at RSA to be chosen to write this book. It was Victor Chang, then the VP of engineering at RSA, who hired me, let me do all kinds of wonderful things in the field and industry of cryptography, and made RSA engineering a great place to work. The geniuses of RSA Labs, especially Burt Kaliski and Matt Robshaw, taught me most of the crypto I know today, and the engineers at RSA, especially Dung Huynh and Pao-Chi Hwang, taught me all about the crypto code.

—Steve Burnett

The first person I'd like to thank is Steve Burnett. I am positive that if he had not agreed to co-author this book with me, I might have given up before I began.

RSA Press definitely must be thanked for giving Steve Burnett and me a chance to write this book. Also, I'd like to thank Steve Elliot, Alex Corona, Betsy Hardinger, LeeAnn Pickrell, and all of the other employees of Osborne/McGraw Hill who worked to make this book possible.

Both Jessica Nelson and Blake Dournaee did an excellent job providing technical review—thank you. I'd like to offer a special thanks to Mohan Atreya and Scott Maxwell of RSA Security; both were a source of excellent ideas and technical input.

Thanks to my friends at RSA Security for being patient and understanding while I worked long hours on the book.

I especially want to thank Jerry Mansfield, a great friend who taught me to take life as it comes. Finally, I would like to thank my family for their support.

—Stephen Paine

Preface

Application developers never used to add security to their products because the buying public didn't care. To add security meant spending money to include features that did not help sales. Today, customers demand security for many applications. The Federal Bureau of Investigation published the following Congressional Statement on February 16, 2000:

> "There were over 100 million Internet users in the United States in 1999. That number is projected to reach 177 million in United States and 502 million worldwide by the end of 2003. Electronic commerce has emerged as a new sector of the American economy, accounting for over $100 billion in sales during 1999; by 2003 electronic commerce is projected to exceed $1 trillion."

At the same time, the *Computer Security Institute* (CSI) reported an increase in cybercrime, "55% of the respondents to our survey reported malicious activity by insiders." Knowing this, you can be sure growing corporations need security products.

The most important security tool is cryptography. Developers and engineers need to understand crypto in order to effectively build it into their products. Sales and marketing people need to understand crypto in order to prove the products they are selling are secure. The customers buying those products, whether end users or corporate purchasing agents, need to understand crypto in order to make well-informed choices and then to use those products correctly. IT professionals need to understand crypto in order to deploy it properly in their systems. Even lawyers need to understand crypto because governments at the local, state, and national level are enacting new laws defining the responsibilities of entities holding the public's private information.

This book is an introduction to crypto. It is not about the history of crypto (although you will find some historical stories). It is not a guide to writing code, nor a math book listing all the theorems and proofs of the underpinnings of crypto. It does not describe everything there is to know about crypto; rather, it describes the basic concepts of the most widely used crypto in the world today. After reading this book, you will know what computer cryptography does and how it's used today. For example, you will

- Understand the difference between a block cipher and a stream cipher and know when to use each (if someone tries to sell you an application that reuses a stream cipher's key, you will know why you shouldn't buy it).
- Know why you should not implement key recovery on a signing-only key.
- Understand what SSL does and why it is not the security magic bullet solving all problems, which some e-commerce sites seem to imply.
- Learn how some companies have effectively implemented crypto in their products.
- Learn how some companies have used crypto poorly (smart people learn from their own mistakes; brilliant people learn from other people's mistakes).

There are, of course, many more things you will learn in this book.

Chapter 1 delves into why cryptography is needed today; Chapters 2 through 5 describe the basic building blocks of crypto, such as symmetric keys and public keys, password-based encryption, and digital signatures. In Chapters 6 through 8, you will see how these building blocks are used to create an infrastructure through certificates and protocols. In Chapter 9, you will learn how specialized hardware devices can enhance your security. Chapter 10 explores the legal issues around digital signatures. Finally, Chapters 11 and 12 show you some real-world examples of companies doing it wrong and doing it right.

Throughout this book we use some standard computer hexadecimal notation. For instance, we might show a cryptographic key such as the following:

```
0x14C608B9 62AF9086
```

Many of you probably know what that means, but if you don't, read Appendix A. It's all about how the computer industry displays bits and bytes in hexadecimal. It also describes ASCII, the standard way letters, numerals, and symbols are expressed in computers.

In Chapter 6, you'll find a brief description of ASN.1 and BER/DER encoding. If you want to drill down further into this topic, read Appendix B.

In Appendix C, you will find further detailed information about many of the topics discussed in the book. These details are not crucial to understanding the concepts presented in the main body of the book; but for those who wish to learn more about the way crypto is used today, this appendix will offer interesting reading.

Finally, the accompanying CD contains the RSA Labs *Frequently Asked Questions* (FAQ) about cryptography. The FAQ contains more detailed information about many of the concepts presented in this book. For instance, the FAQ describes much of the underlying math of crypto and the political issues surrounding export, and it offers a glossary and bibliography. Our goal in writing this book was to explain the crypto that the vast majority of you need to know. If you want more detail, start with the FAQ.

About the Authors

Steve Burnett With degrees in math from Grinnell College in Iowa and The Claremont Graduate School in California, Steve Burnett has spent most of his career converting math into computer programs, first at Intergraph Corporation and now with RSA Security. He is currently the lead crypto engineer for RSA's BSAFE Crypto-C and Crypto-J products, which are general purpose crypto software development kits in C and Java. Burnett is also a frequent speaker at industry events and college campuses.

Stephen Paine Stephen Paine has worked in the security field throughout most of his career—formerly for the United States Marine Corps and SUN Microsystems. He is currently a systems engineer for RSA Security, where he explains security concepts to corporations and developers worldwide and provides training to customers and RSA employees.

About the Reviewers

Blake Dournaee Blake Dournaee joined RSA Security's developer support team in 1999, specializing in support and training for the BSAFE cryptography toolkits. Prior to joining RSA Security, he worked at NASA-Ames Research Center in their security development group. He has a B.S. in Computer Science from California Polytechnic State University in San Luis Obispo and is currently a graduate student at the University of Massachusetts.

Jessica Nelson Jessica Nelson comes from a strong background in computer security. As an officer in the United States Air Force, she spearheaded the 12 Air Force/Southern Command Defensive Information Warfare division. She built programs that integrated computer and communications security into the DoD's Information Warfare. She graduated from UCSD with a degree in physics and has worked with such astrophysicists as Dr. Kim Griest and Dr. Sally Ride. She currently acts as technical sales lead in the western division of a European security company.

CHAPTER 1

Why Cryptography?

"According to the affidavit in support of the criminal complaint, the Secret Service began investigating this matter when it learned that there had been unauthorized access to [online brokerage] accounts of several [anonymous company] employees. One [anonymous company] employee told authorities that approximately $285,000 had been drained from his [online brokerage] account when an unknown person was able to access his account by calling the online broker and providing a name and social security number. It was later determined that at least eight [anonymous company] employees had been victimized this past spring, and that these eight had lost a total of $700,000 from their stock accounts . . . [anonymous company] officials revealed that while working in the financial department, [the accomplice] had access to confidential employee information such as social security numbers and home addresses."*

If someone tells you, "I don't need security. I have no secrets, nothing to hide," respond by saying, "OK, let me see your medical files. How about your paycheck, bank statements, investment portfolio, and credit card bills? Will you let me write down your Social Security number,

---

*Source: U.S. Department of Justice, July 20, 2000 credit card numbers, and bank account numbers? What's the PIN for your ATM, credit card, or phone card? What's your password to log on to the network at work? Where do you keep your spare house key?"

The point is that we all have information we want kept private. Sometimes the reason is simply our natural desire for privacy; we would feel uncomfortable if the whole world knew our medical history or financial details. Another good reason is self-protection—thieves could use some kinds of information to rob us. In other words, the motives for keeping a secret are not automatically nefarious.

Corporations also have secrets—strategy reports, sales forecasts, technical product details, research results, personnel files, and so on. Although dishonest companies might try to hide villainous activities from the public, most firms simply want to hide valuable information from dishonest people. These people may be working for competitors, they might be larcenous employees, or they could be *hackers* and *crackers*: people who break into computer networks to steal information, commit vandalism, disrupt service, or simply to show what they can do.

Security Provided by Computer Operating Systems

In the past, security was simply a matter of locking the door or storing files in a locked filing cabinet or safe. Today, paper is no longer the only medium of choice for housing information. Files are stored in computer databases as well as file cabinets. Hard drives and floppy disks hold many of our secrets. How do you lock a hard drive?

How Operating Systems Work

Before we talk about how computer data is protected, let's take a brief look at how computers get and store information. The usual way to access data on a computer or network is to go through the *operating system* (OS), such as DOS, Windows, Windows 95, Windows NT, MacOS, UNIX, Linux, Solaris, or HP/UX. The OS works like an application, taking input, performing operations based on the input, and returning output. Whereas, for example, a spreadsheet application takes the numbers you type into it, inserts them into cells, and possibly performs calculations such as adding columns, an OS takes your commands in the form of mouse clicks, joysticks, touch screens, or keyboard input-commands such as "show a listing of the files in this directory"—and performs the request, such as printing to the screen a list of files. You can also ask the OS to launch a particular application—say, a text editor. You then tell the text editor to open a file. Behind the scenes, the editor actually asks the OS to find the file and make its contents available to the editor.

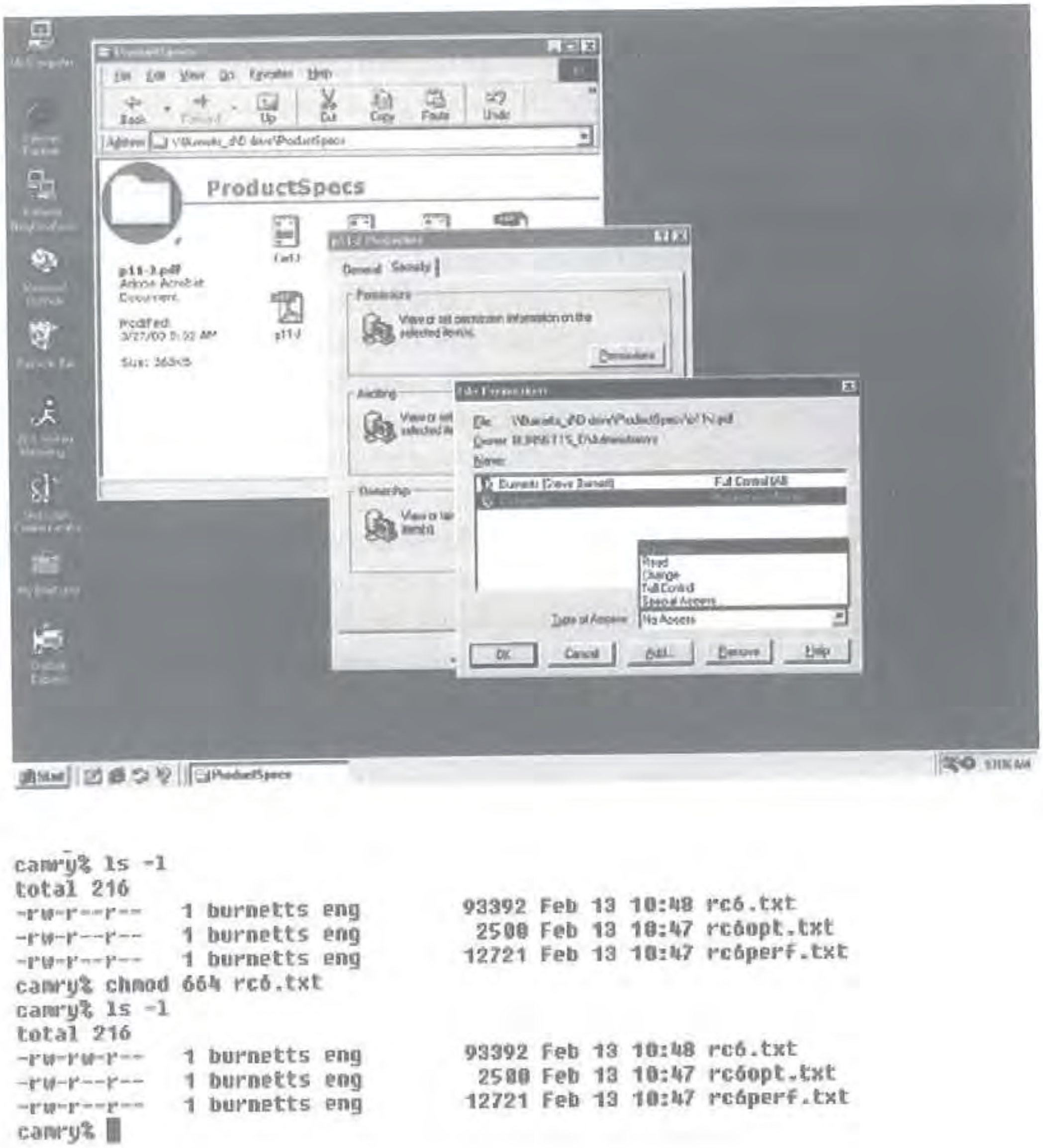
Virtually all computers built today include some form of protection courtesy of the OS. Let's take a look at how such protection works.

Default OS Security: Permissions

Virtually all operating systems have some built-in *permissions,* which allow only certain people access to the computer (its hard drive, memory, disk space, and network connection). Such access is implemented via a *login* procedure. If the user does not present the appropriate credentials (perhaps a user name and password), the OS will not allow that individual to use the computer. But even after a user is logged in, certain files may still be off-limits. If someone asks to see a file, the OS checks to see whether that requester is on the list of approved users; if not, the OS does not disclose the contents (see Figure 1-1).

Access to most business computers and networks is controlled by someone known as a *superuser* or *system administrator* (often shortened to *sys admin*). This system administrator is the person charged with creating and closing user accounts and maintaining the systems and network. A typical task of this superuser account is to override protections. Someone forgot a password? A file is read-protected (meaning that it cannot be opened and read)? The superuser has permission to circumvent the OS permissions to respond to these problems. (This is where the name "superuser" comes from; this individual can do anything.)

How does the OS know that the person requesting such system overrides is the superuser? The OS grants this access by user name and password. The superuser user name is usually "su" or "root" or "administrator." Unfortunately, techniques for circumventing these default defenses are widely known.



```
camry% ls -1
total 216
-rw-r--r--   1 burnetts eng          93392 Feb 13 10:48 rc6.txt
-rw-r--r--   1 burnetts eng           2500 Feb 13 10:47 rc6opt.txt
-rw-r--r--   1 burnetts eng          12721 Feb 13 10:47 rc6perf.txt
camry% chmod 664 rc6.txt
camry% ls -1
total 216
-rw-rw-r--   1 burnetts eng          93392 Feb 13 10:48 rc6.txt
-rw-r--r--   1 burnetts eng           2500 Feb 13 10:47 rc6opt.txt
-rw-r--r--   1 burnetts eng          12721 Feb 13 10:47 rc6perf.txt
camry% █
```

Figure 1-1

(a) In Windows NT, a file's permission is given in its Properties screen. (b) In UNIX, you type **ls -l** to see a file's permission

Attacks on Passwords

Many computers or operating systems come with a preset superuser account and password. In many cases, several passwords are used for various superuser functions. The superuser may have a password to create accounts, a different password to control network functionality, another to conduct or access nightly backups, and so on.

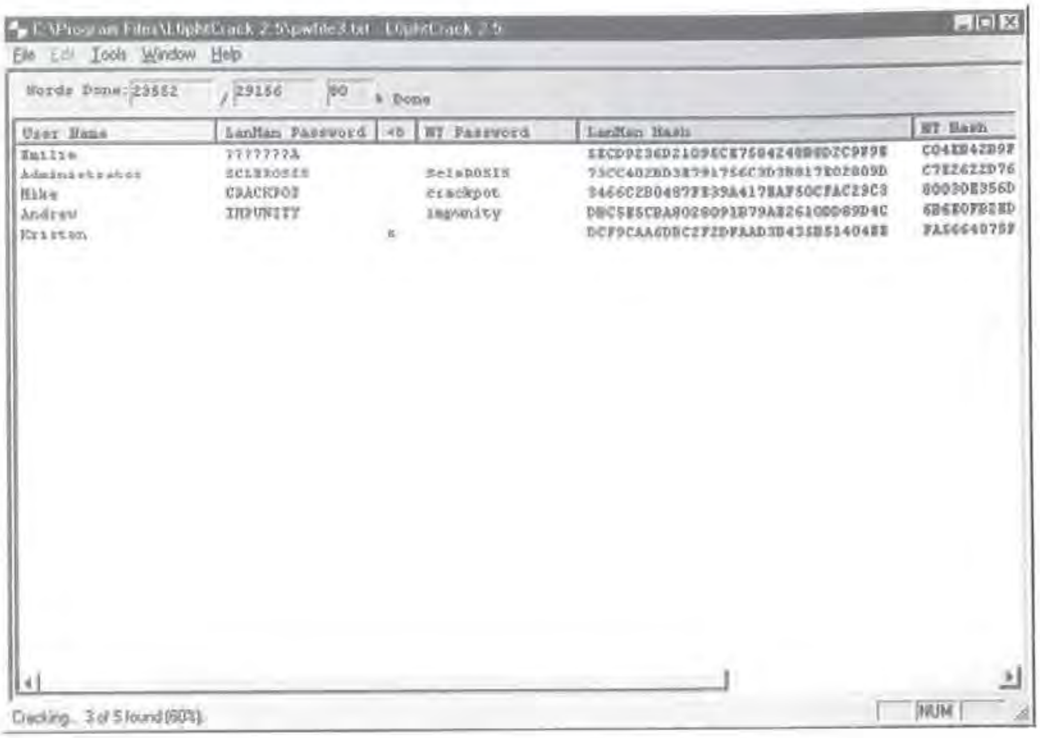
For a cracker, logging on to a system as the superuser is possibly the best way to collect data or do damage. If the superuser has not changed an operating system's preprogrammed passwords, the network is vulnerable to attack. Most crackers know these passwords, and their first attempt to break into a network is simply to try them.

If an attacker cannot log on as the superuser, the next best thing might be to figure out the user name and password of a regular user. It used to be standard practice in most colleges and universities, and in some commercial companies, to assign every student or employee an account with a user name and an initial password—the password being the user name. Everyone was instructed to log on and change the password, but often, hackers and crackers logged on before legitimate users had a chance. In other cases, some people never actually used their accounts. Either way, intruders were able to gain access. This "user name as password" system is still used on many campuses and corporate settings to this day.

If the password of a particular user name is not the user name itself, crackers may try to guess the correct password. Guessing a password might be easy for an insider (such as a fellow employee), who probably knows everyone's user name. It's common for people to use a spouse's name or a birthday as a password. Others write down their passwords, and a quick search of a desk might yield the valuable information. Some systems have guest accounts, with a user name of "guest" and a password of "guest."

But even if the intruder is not very good at guessing passwords, applications are available that automate exhaustive password searches. These applications, called *password cracking* software, are made by a variety of people for various reasons—some legitimate and others not so legitimate. To use one of these tools, the intruder needs access to your computer (network access may be sufficient). Once connected, the hacker simply runs the password cracking application. If the password is weak, within minutes the hacker will have privileged access.

Figure 1-2 shows a popular application known as l0phtCrack. This application is designed to allow systems administrators to test the passwords in use by their users. The idea is that if a sys admin can crack a password, so can crackers.



Words Done: 23552 / 29156 80 % Done

User Name	LanMan Password	<8	NT Password	LanMan Hash	NT Hash
Emilie	???????A			[illegible]	[illegible]
Administrator	SCLEROSIS		ScleROSIS	[illegible]	[illegible]
Mike	CRACKPOT		crackpot	[illegible]	[illegible]
Andrew	IMPUNITY		impunity	[illegible]	[illegible]
Kristen		x		[illegible]	[illegible]

Cracking... 3 of 5 found (60%).

Figure 1-2

l0phtCrack is used to test passwords for vulnerability

Attacks That Bypass Operating Systems

An operating system tags certain files and prevents unapproved people from seeing the contents. Although a cracker or thief might be able to gain access to such files by posing as the superuser or a regular user, another possibility is to ignore the OS altogether and get the contents in some other way.

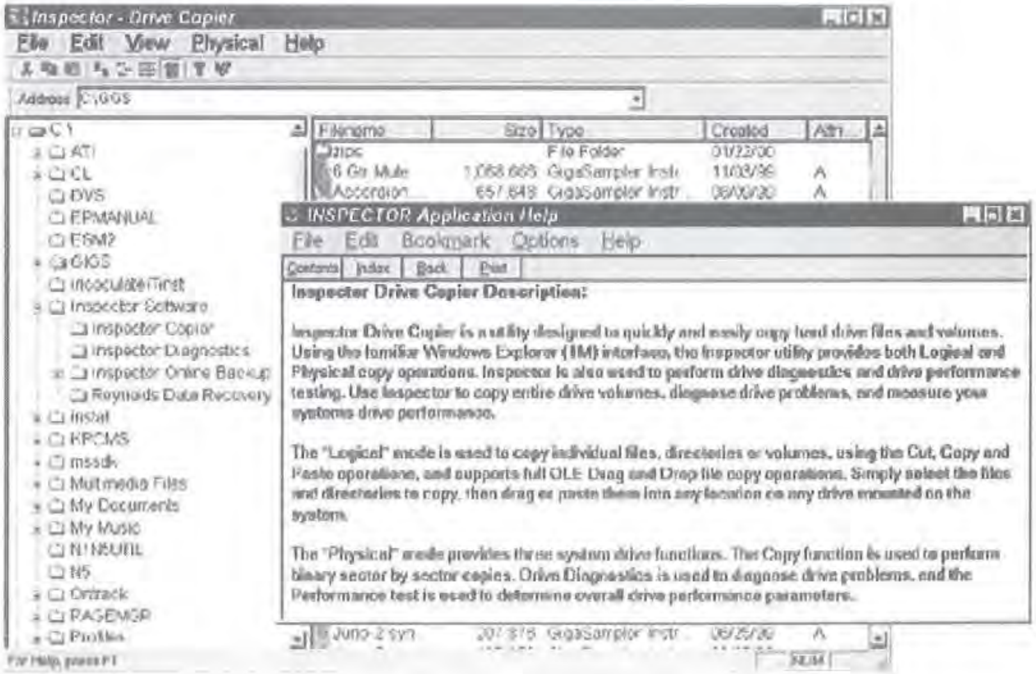
Data Recovery Attack

One function of a computer's operating system is to help users find and use the specific data or application they want. In this way, an OS works like the index of a book. Just as an index directs you to the specific page where you'll find the piece of information you want out of all the pages in a book, the OS organizes data under a directory file structure and uses file extensions to direct you to the data you want on the hard disk. But as far as the computer is concerned, the data is simply so many electronic bits.

If you don't care what order they're in, it's possible to read those bits as bits and not as files of text or numbers. Human beings can't read bits in this way, but software and hardware devices are available that can scan storage media and read the bits. These tools bypass the OS and grab the raw bits of data, which can then be reconstructed into the original files.

In fact, an entire industry has been built on the concept of reading bits as bits, a process called *data recovery.* When you have a system crash or some kind of physical damage to a hard drive, you can take your computer to a data recovery expert, who often can reconstruct the files on the disk. These companies provide a valuable service, helping to prevent total losses in the event of a natural disaster or computer failure.

Reynolds Data Recovery of Longmont, Colorado, performs data recovery and also sells software that allows you to perform your own recovery (see Figure 1-3). According to the company's advertising, one of its products, Inspector Copier, "does not reference the OS installed on the devices, [and] this allows copies of different systems such as NT, Novell, UNIX, Linux or Windows 2000!"



Inspector Drive Copier Description:

Inspector Drive Copier is a utility designed to quickly and easily copy hard drive files and volumes. Using the familiar Windows Explorer (TM) interface, the Inspector utility provides both Logical and Physical copy operations. Inspector is also used to perform drive diagnostics and drive performance testing. Use Inspector to copy entire drive volumes, diagnose drive problems, and measure your systems drive performance.

The "Logical" mode is used to copy individual files, directories or volumes, using the Cut, Copy and Paste operations, and supports full OLE Drag and Drop file copy operations. Simply select the files and directories to copy, then drag or paste them into any location on any drive mounted on the system.

The "Physical" mode provides three system drive functions. The Copy function is used to perform binary sector by sector copies. Drive Diagnostics is used to diagnose drive problems, and the Performance test is used to determine overall drive performance parameters.

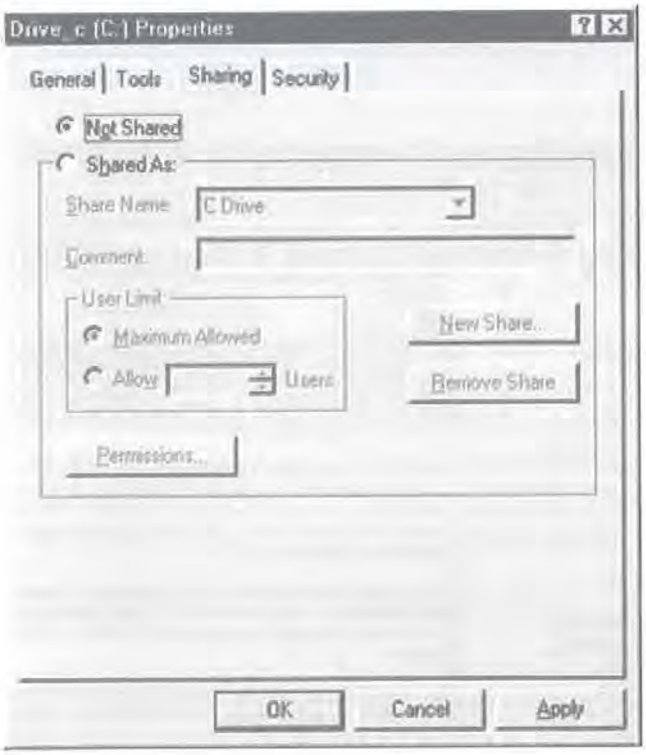
Figure 1-3

Inspector Copier from Reynolds Data Recovery (courtesy of Mark Tessin of Reynolds Data Recovery)

But the techniques of data recovery can also be used by attackers to circumvent OS protections. To extend Inspector Copier, Reynolds sells a network backup service that remotely backs up data on hard drives. It uses Inspector Copier to extract the bits so that even if a hard drive is damaged, a clean backup can be made. Although this service can be valuable to many companies, it also means that the data recovery program can be run remotely. Mark Tessin of Reynolds points out that the service can even circumvent Windows NT security. Suppose your PC is connected to a network but you don't want the outside world to see your C: drive. You can set the permissions on your drive so that only you have read or write permission to it (see Figure 1-4). The Reynolds network backup service can circumvent that permission and read the files anyway. This is not to imply that Reynolds Data Recovery will steal your data, only to illustrate that it is possible.

Figure 1-4

Setting network permissions on a local drive using Windows NT



For serious disk drive failures (such as fire damage), data recovery might be possible only through specialized hardware devices. But an attacker is not trying to steal your data from a damaged drive. Data recovery software is so sophisticated and effective that it's all anyone needs to extract bits from a healthy storage medium.

To ensure the security of your data, you must assume that even though some protections may be sufficient against some opponents, there will likely be someone out there with the resources to mount a successful attack. Only if such an individual never comes after your data are you safe.

Memory Reconstruction Attack

Often, sensitive material is not stored on hard drives but does appear in a computer's memory. For example, when the program you're running allocates some of the computer's memory, the OS tags that area of memory as unavailable, and no one else can use it or see it. When you're finished with that area of memory, though, many operating systems and programs simply "free" it—marking it as available—without overwriting it. This means that anything you put into that memory area, even if you later "deleted" it, is still there. A memory reconstruction attack involves trying to examine all possible areas of memory. The attacker simply allocates the memory you just freed and sees what's left there.

A similar problem is related to what is called "virtual memory." The memory managers in many operating systems use the hard drive as virtual memory, temporarily copying to the hard drive any data from memory that has been allocated but is momentarily not being used. When that information is needed again, the memory manager *swaps* the current virtual memory for the real memory. In August 1997, *The New York Times* published a report about an individual using simple tools to scan his hard drive. In the swap space, he found the password he used for a popular security application.

On UNIX systems, the OS "dumps core" in response to certain system errors. Core dump has become almost synonymous with a program exiting ungracefully. But on UNIX, the core file that results from a core dump is actually a snapshot of memory at the time the error occurred. An attacker who wants to read memory may be able to induce a core dump and peruse the core file.

Figure 1-5 illustrates how memory reconstruction attacks work.

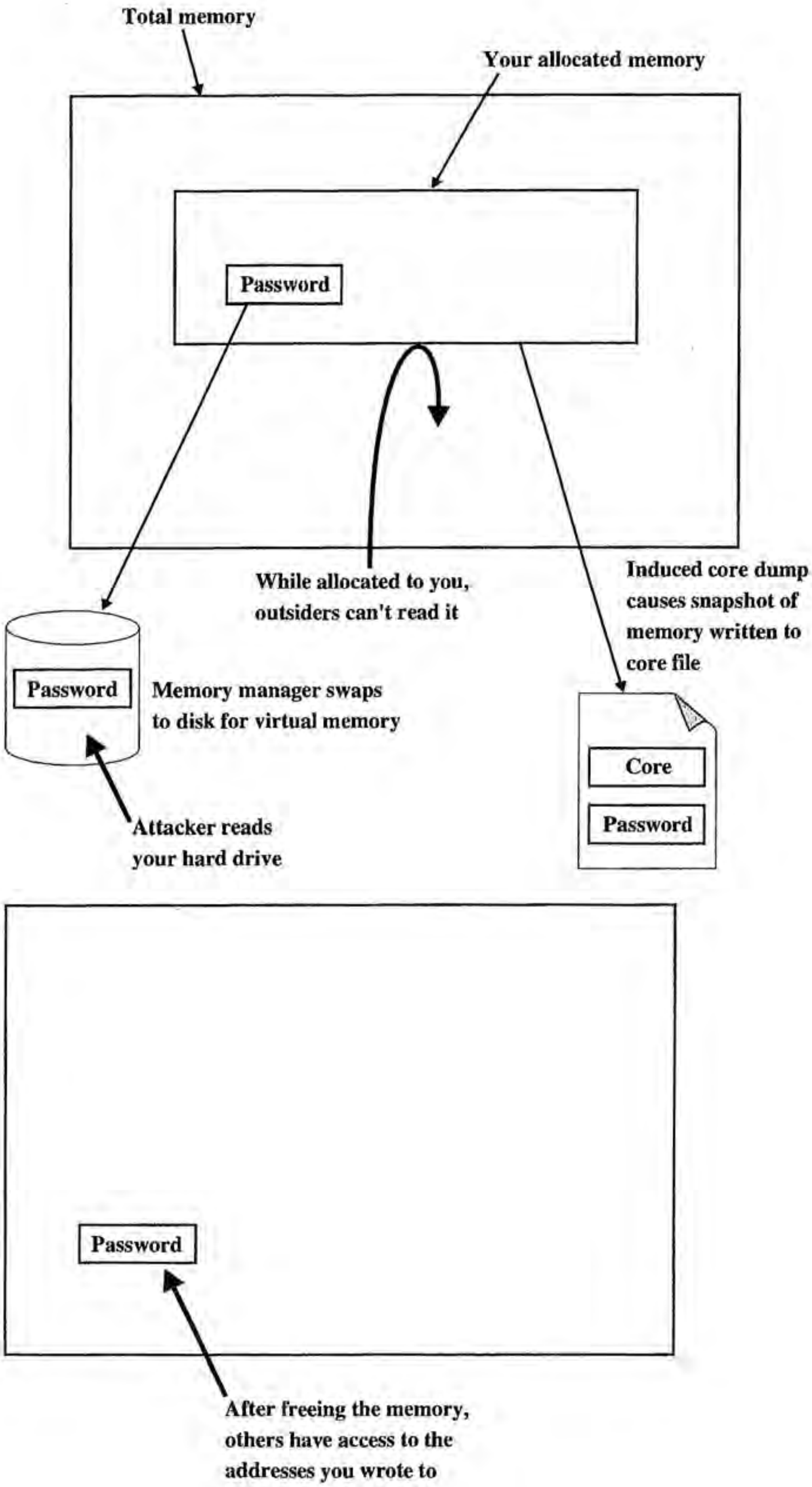


Figure 1-5

Your sensitive material, such a password, is not stored on a hard drive but does appear in memory. An attacker may read the data in memory in the swap space, in a core file, or simply after you free it

Added Protection Through Cryptography

For your secrets to be secure, it may be necessary to add protections not provided by your computer system's OS. The built-in protections may be adequate in some cases. If no one ever tries to break into or steal data from a particular computer, its data will be safe. Or if the intruder has not learned how to get around the simple default mechanisms, they're sufficient. But many attackers do have the skills and resources to break various security systems. If you decide to do nothing and hope that no skilled cracker targets your information, you may get lucky, and nothing bad will happen. But most people aren't willing to take that risk.

As you'll learn in the chapters to come, one of the most important tools for protecting data is *cryptography,* any of various methods that are used to turn readable files into gibberish. For example, suppose your sensitive material looks like this:

```
do not believe that the competition can match the new feature set,
yet their support, services, and consulting offerings pose      a
serious threat to our salability. We must invest more money in our
```

Here is what the data looks like when it's encrypted:

```
ú?SdÏ:1/41Yïõ´]Y çmúcA‡[<_b:vH˜_ô UGØ>e´œ_%` ,<_lo¡`üùØ_"G
ri§õêÌqY_Ë•ùK_æ7ÁFT1≅Ó_ . . . À*R8′» ÿÄh . . . o-
2ñ?Í•ÇÕ(tm)ÇvéR]’Î_¬’(r)<Ñ_UéR`q3/4¥Ü_Ã‡ÁuÉ·¶ _>FômÈÕ6_cêàB1/28#ùh&(G
[gh_!>¶≅Oædtn*´bô1/4jWM1/4B-Â_≅_¬1/4<"-ÏEÿåb{=.AÛH__
```

Even if an attacker obtains the contents of the file, it is gibberish. It does not matter whether or not the OS protections worked. The secret is still secret.

In addition to keeping secrets, cryptography can add security to the process of authenticating people's identity. Because the password method used in almost all commercial operating systems is probably not very strong against a sophisticated (or even an unsophisticated) attacker, it's important to add protection. The cryptographic techniques for providing data secrecy can be adapted to create strong digital identities. If attackers want to pose as someone else, it's not a matter simply of guessing a password. Attackers must also solve an intractable mathematical problem (see Figure 1-6).

Figure 1-6

To pose as Steve Burnett of RSA Security, you'd have to factor this number (see also Chapter 4)

111,103,906,294,152,860,689,339,031,055,865,718,
797,834,178,049,634,993,529,562,676,343,628,611,
324,998,912,180,711,483,651,242,218,389,147,835,
598,353,467,199,134,664,870,577,824,583,579,439,
533,042,724,963,790,890,892,988,756,173,576,982,
820,529,088,558,175,928,394,148,986,383,304,407,
218,632,861,415,573,872,050,375,072,884,180,285,
838,244,342,451,974,820,729,610,630,901,524,541,
854,611,490,009,870,503,127

The Role of Cryptography in Data Security

In the physical world, security is a fairly simple concept. If the locks on your house's doors and windows are so strong that a thief cannot break in to steal your belongings, the house is secure. For further protection against intruders breaking through the locks, you might have security alarms. Similarly, if someone tries to fraudulently withdraw money from your bank account but the teller asks for identification and does not trust the thief's story, your money is secure. When you sign a contract with another person, the signatures are the legal driving force that impels both parties to honor their word.

In the digital world, security works in a similar way. One concept is *privacy,* meaning that no one can break into files to read your sensitive data (such as medical records) or steal money (by, for example, obtaining credit card numbers or online brokerage account information). Privacy is the lock on the door. Another concept, *data integrity,* refers to a mechanism that tells us when something has been altered. That's the alarm. By applying the practice of *authentication,* we can verify identities. That's comparable to the ID required to withdraw money from a bank account (or conduct a transaction with an online broker). And finally, *nonrepudiation* is a legal driving force that impels people to honor their word.

Cryptography is by no means the only tool needed to ensure data security, nor will it solve all security problems. It is one instrument among many. Moreover, cryptography is not foolproof. All crypto can be broken, and, more importantly, if it's implemented incorrectly, it adds no real security. This book provides an introduction to cryptography with a focus on the proper use of this tool. It is not intended as a complete survey of all there is to know about cryptography. Rather, this book describes the most widely used crypto techniques in the world today.

CHAPTER 2

Symmetric-Key Cryptography

Cryptography converts readable data into gibberish, with the ability to recover the original data from that gibberish. The first flavor of crypto is called symmetric-key. In this approach, an algorithm uses a key to convert information into what looks like random bits. Then the same algorithm uses the same key to recover the original data.

Pao-Chi is a sales rep for a company that makes printing machinery. He sells to newspapers, magazines, independent printing houses large and small, and even universities. His product line includes presses, tools, replacement parts, repair services, and training. The end of the quarter is coming up in a couple of weeks, and he's just received a memo from Gwen, the vice president of sales. The company is having difficulty "making its numbers," the memo says. Then it outlines a new, complex pricing policy.

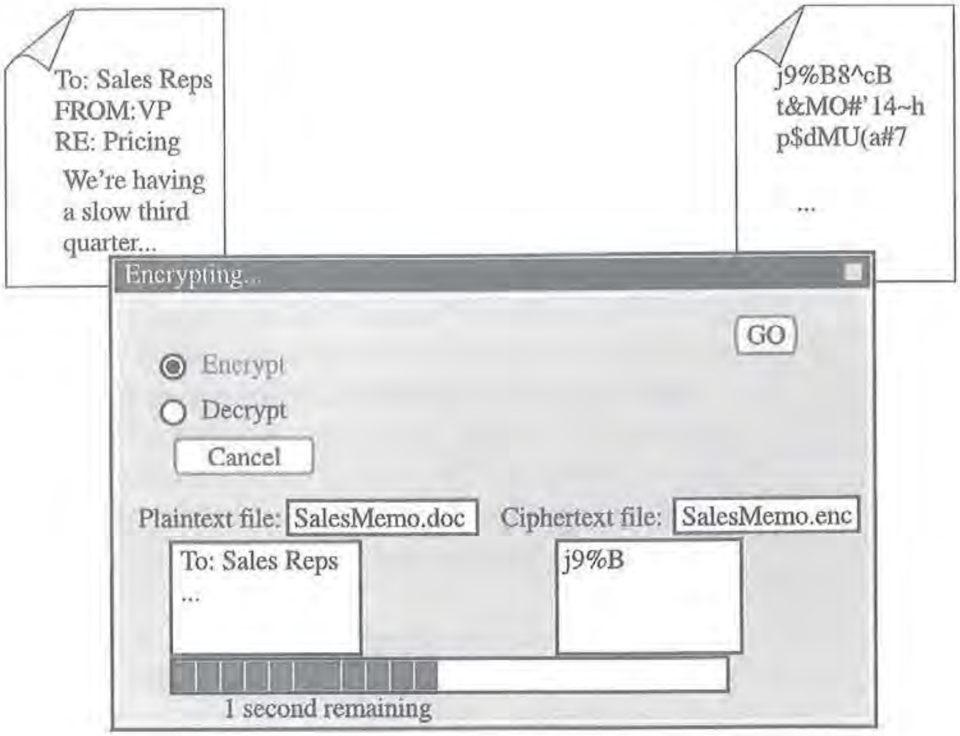
This new policy lists the asking prices for all their products and also indicates the lowest prices sales reps are allowed to negotiate. In the past, they've based the amount of the discounts they give on the size of the order, expectations of future sales with a given client, and other factors. But now, the memo states, sales reps have the authority to give even bigger discounts.

Pao-Chi wants to closely limit who has access to this information. If potential customers knew how far he was willing to go in discounting, they would have the edge in negotiations. Existing customers might demand rebates, and competitors would gain knowledge that could aid them in winning contracts. In addition, stock analysts or business reporters could report the company's slow sales this quarter, affecting its reputation.

How can Pao-Chi and Gwen keep this memo secret? They could choose not to let it leave the office, or maybe Pao-Chi could simply memorize it. But it's more than 20 pages long and too complex to memorize, and he'll need to consult it while trying to make a sale.

So Pao-Chi keeps an electronic copy of the memo on his laptop, and takes steps to protect the file. In Chapter 1, we saw that typical protection techniques are not sufficient. Pao-Chi can lose his laptop, or someone might steal it or simply look through the files while he's at lunch. To protect the file, he decides to encrypt it.

Let's say Pao-Chi buys a computer program to encrypt sensitive files. When running the program, he simply flips the switch to "Encrypt" and feeds the file to the program (see Figure 2-1). When the file comes out of the program, it looks like gibberish. If intruders get their hands on it, they will have no idea what it means.



To: Sales Reps
FROM: VP
RE: Pricing
We're having a slow third quarter...

j9%B8^cB
t&MO#'14~h
p$dMU(a#7
...

● Encrypt
○ Decrypt

Plaintext file: SalesMemo.doc
Ciphertext file: SalesMemo.enc

To: Sales Reps
...

j9%B

1 second remaining

Figure 2-1

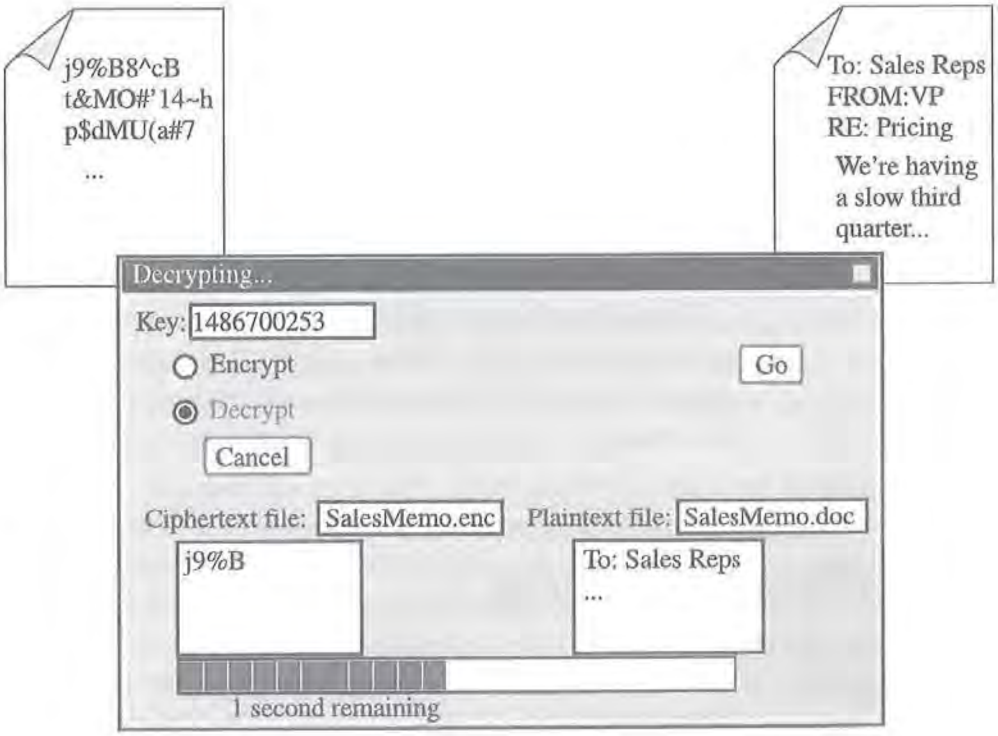
If you feed your sensitive files to an encryption program, you get what looks like gibberish

The problem is that as long as the file is gibberish Pao-Chi won't be able to read it either. To read it, he must somehow convert it back to its original form. The program has just such a feature: he flips the switch to "Decrypt," feeds in the gibberish, and out comes the file in its former condition.

But there's one problem with this scenario. If intruders are able to obtain the encrypted file, surely they can obtain the program that converts it back. Even if they can't, where can Pao-Chi safely store the program? If he can keep the program out of the hands of attackers, why not store his file there as well?

No, he doesn't have a place where he can keep the encrypting and decrypting program safe. And if Pao-Chi has access to it, he must assume that attackers can gain access. That's why he uses encryption in the first place. By itself, an encryption machine cannot protect secrets. Pao-Chi needs additional protection.

That additional protection is a secret number. If he feeds the file *and* a secret number to the program, the program will encrypt the file. Until the program has a secret number, it will not run. To decrypt the file, Pao-Chi must present the gibberish and the same secret number (see Figure 2-2).



j9%B8^cB
t&MO#'14~h
p$dMU(a#7
...

To: Sales Reps
FROM:VP
RE: Pricing
We're having
a slow third
quarter...

Decrypting...

Key: 1486700253

○ Encrypt

● Decrypt

Ciphertext file: SalesMemo.enc

j9%B

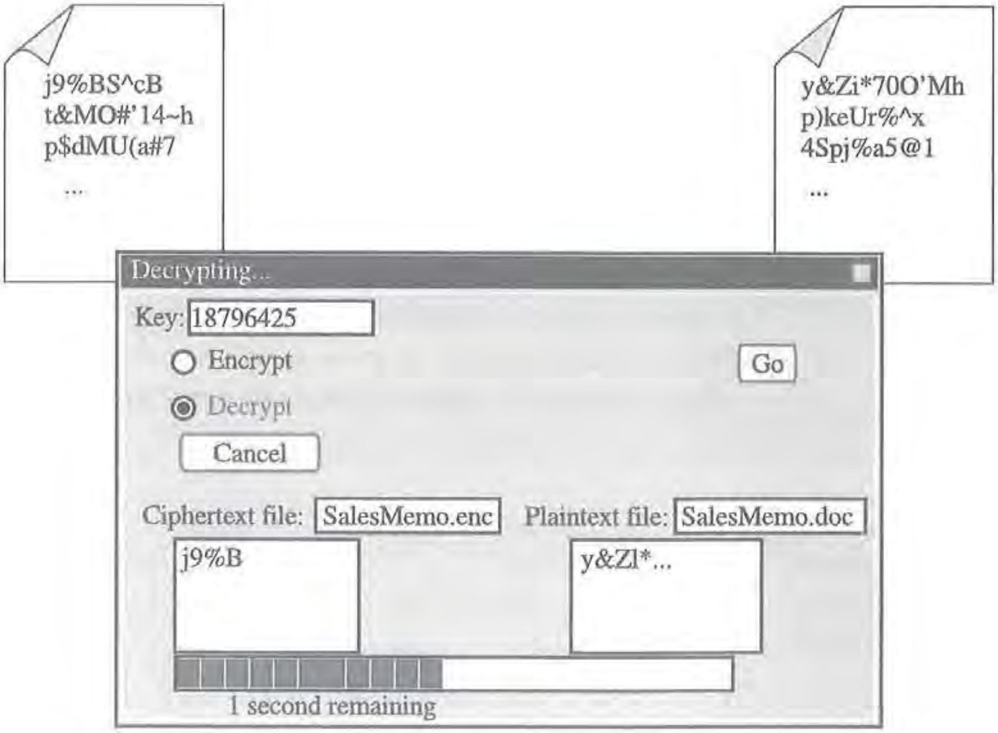
Plaintext file: SalesMemo.doc

To: Sales Reps
...

Figure 2-2

To get encrypted gibberish, you feed sensitive data and a secret number to the encryption machine. To recover the file, you flip the switch to "Decrypt" and then feed it the gibberish and the secret number

If an attacker somehow obtains a copy of the gibberish and feeds it to the program for recovery, it won't work. The program asks for the number, which the attacker does not know. It's possible to try numbers at random (or to try all possible numbers systematically), but every time a wrong number is inserted, the application simply spits out different gibberish (see Figure 2-3).



Key: 18796425

○ Encrypt

● Decrypt

Ciphertext file: SalesMemo.enc

Plaintext file: SalesMemo.doc

Figure 2-3 If attackers try numbers other than the secret value, they get only more gibberish

Even though someone can use the same program Pao-Chi used, it never re-creates the original file without the secret number. Even if the attacker guesses a number close to the original number, even if it is off by only 1, the program will not produce anything close to the correct encrypted file.

Some Crypto Jargon

The system we've just described is known as *symmetric-key cryptography*. Some people call it *secret-key cryptography*. Here are some official terms.

When you want to convert sensitive information to gibberish, you *encrypt* the data. To convert it back, you *decrypt* it.

To do this, you use an *algorithm*. The word "algorithm" is a scientific term for a recipe or step-by-step procedure. It is a list of instructions or things to do in a particular order. An algorithm might have a rigid list of commands to follow, or it might contain a series of questions and depending on the answers, describe the appropriate steps to follow. A mathematical algorithm might list the operations to perform in a particular order to "find x." For example, an automobile diagnostic algorithm may ask questions about oil pressure, torque, fluid levels, temperature, and so on, to determine what's wrong. A computer program can also implement an algorithm, meaning the program converts the algorithm's list of commands, questions, and operations into the computer's language, enabling it to perform the steps in the appropriate order. In computer cryptography, algorithms are sometimes complex mathematical operations or simply bit manipulations. Many encryption algorithms exist, and each one has its own particular list of commands or steps. Just as you can have a program that plays Solitaire or one that computes the trajectory of satellites, you can have a program that implements an encryption algorithm that takes your data and converts it to gibberish.

The data that you want to keep secret is called *plaintext* (some call it *cleartext*). Your plaintext could be a human-readable text file, such as the memo. Or it could be a binary file, which looks like nonsense to human eyes but makes perfect sense to a computer program. For example, if you open a PowerPoint file using Windows' Edit text editor, the file looks like gibberish because the program can't convert the PowerPoint formatting information; but if you open the same file in PowerPoint, it appears as intended. Whether or not your information is readable by a human or a given program, it's called plaintext.

After the data is encrypted, it's known as *ciphertext*.

The algorithm encrypts your plaintext into ciphertext, but it needs one more thing—a *key*. In our sales rep example, the secret number used to encrypt the pricing memo was its key. In computer crypto, the key is always a number or a set of numbers.

We've also met the *attacker*, someone trying to steal information. Actually, an attacker may try to do more than simply uncover someone else's secrets. Some attackers try to pose as people they are not, disable Web sites, delete someone else's information, prevent customers from buying at a particular online merchant, slow down systems, and on and on and on. The term "attacker" is simply a catchall for the individual from whom you must protect your digital assets.

The study of breaking cryptographic systems is known as of *cryptanalysis*. Similar to the attacker, the *cryptanalyst* looks for weaknesses in algorithms. All algorithms can be "broken;" the good ones are simply the algorithms strong enough to withstand an attack for so long the break comes "too late." So a cryptanalyst's job is to find weaknesses that may help someone break the algorithm faster. Attackers may use cryptanalytic techniques to do damage, but they may also use other tools.

The *cryptographer* develops crypto systems; the cryptanalyst looks for weaknesses. It's important for the crypto community to know about the weaknesses because attackers are looking for them as well. Attackers are almost certainly not going to announce their discoveries to the world, so cryptanalysts perform a service, letting us all know what attackers probably know but won't tell us.

What Is a Key?

The term "key" comes from the fact that the secret number you choose works in the same way that a conventional key works. To protect the contents of your house, you install a lock on the door. To operate the lock, you insert the key and turn it. The lock's tumblers and mechanisms work with the key in a prescribed way to activate a barrier that prevents the door from being opened. To unlock the door, you insert the key and turn it in the opposite direction. The tumblers and mechanisms work with the key to reverse the process and allow the door to be opened.

In cryptography, to protect the contents of your files, you install a lock (an encryption algorithm) on your door (the computer). To operate the lock (encrypt the data), you insert the key (the secret number) and execute it (instead of turning this key, you operate the program by double-clicking, clicking OK, or pressing ENTER). The algorithm performs its steps using the key to alter the plaintext and convert it to ciphertext. To unlock the encrypted file, you insert the same key and execute. The algorithm reverses the steps and converts the ciphertext back to the original plaintext.

Just as only the correct house key can open your front door, only the correct encryption key can decrypt data. In symmetric-key cryptography, the key that is used to encrypt data is the same key that is used to decrypt it. "Symmetric" essentially means "the same on two sides," and that's what we have here: the same key on two sides of the encryption process. Figure 2-4, a picture you'll see quite a bit in this book, is the image we use to visualize cryptography.

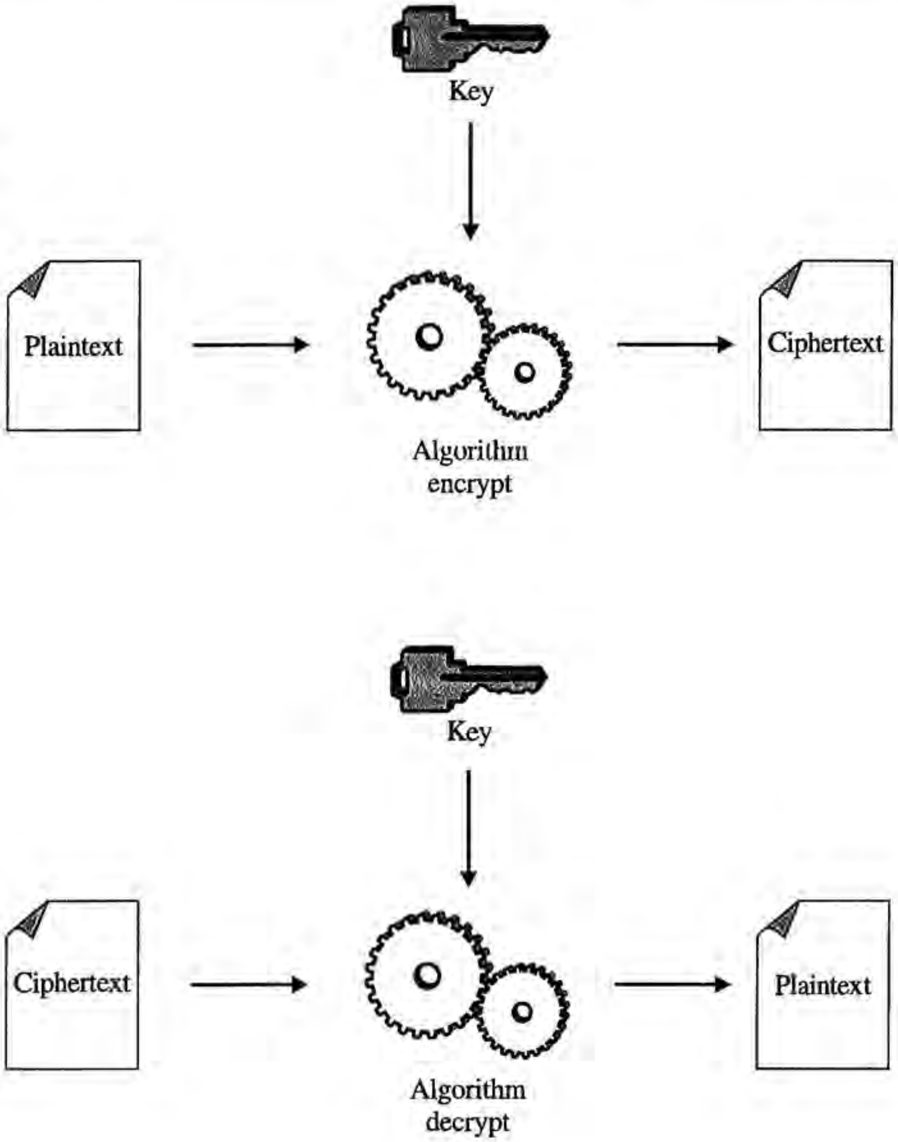


Figure 2-4

This encryption algorithm uses the key to convert plaintext to ciphertext. In symmetric-key cryptography, the key used for encryption is also necessary for decryption

In this book we talk about some of the many different encryption algorithms you have to choose from, but remember that keys are not interchangeable among algorithms. For example, suppose that you encrypt data using the Triple *Digital Encryption Standard* (DES) algorithm (discussed later in the section titled "Triple DES"). If you try to decrypt the data using the *Advanced Encryption Standard* (AES) cipher (discussed later in the section titled "Advanced Encryption Standard"), even if you use the same key, you will not get the correct result.

Why Is a Key Necessary?

All computer crypto operates with keys. Why is a key necessary? Why not create an algorithm that doesn't need a key?

As you saw in the memo example, if attackers can understand the algorithm, they can recover secret data simply by executing the algorithm. That's like installing a deadbolt on your front door with the lock on the outside. It's true that when the deadbolt is in place, the door cannot be opened. But anyone can open the door simply by turning the lock.

It might seem that the solution is to keep the algorithm secret, but that approach has several problems. First, attackers always crack the algorithm (see "Historical Note: They Always Figure Out The Algorithm," later in this chapter). What's more, suppose you do manage to keep the algorithm secret. Unless you are a cryptography expert and develop your own algorithms, you also must trust the company that wrote your algorithm never to reveal it deliberately or accidentally. Does anyone have that much trust in a corporate entity?

Here's the real question: Which would you trust more to keep secrets—an algorithm that must be kept secret, or an algorithm that can do its job even if everyone in the world knows exactly how it works? That's where keys come in.

Keys relieve you of the need to worry about the algorithm used in your encryption scheme. If you protect your data with a key, you need protect only the key, something that's easier to do than protecting an algorithm. In this book you'll learn a lot about key protection. Also, if you use keys to protect your secrets, you can use different keys to protect different secrets. This means that if someone breaks one of your keys, your other secrets are still safe. If you're depending on a secret algorithm, an attacker who breaks that one secret gets access to all your secrets.

Generating a Key

In a symmetric-key cryptographic system, the key is only a number. It can be any number as long as it's the right size, so you simply pick a number at random. Then, the next time you need a key, you pick another number at random. The question is, how do you pick a number at random?

Historical Note: They Always Figure Out the Algorithm

Cryptographers are often asked a key question: "Can't I just encrypt my data and simply not tell the attackers what algorithm I used and how big the key is? How can they break my message then?" There are three answers.

Answer 1: They Always Figure It Out Anyway

Attackers can deduce your algorithm without any help from you. Eventually, they always figure it out. Always. Without exception. Never in the history of cryptography has someone been able to keep an algorithm secret.

In war, spies have always found ways of discovering the algorithm, whether it originates in a mathematical operation or a machine. They steal it or get someone to reveal it, maybe through blackmail, extortion, or the time-tested cryptanalytic technique known as "the rubber-hose attack." Agents have always uncovered the algorithm or gotten a copy of the machine. For example, in World War II, Polish soldiers captured the German Enigma machine early in the war. Enigma was the crypto machine the German military used. The allies (namely the British) were able to crack the code more easily because they had the machine in their possession.

Alternatively, the cryptanalysts simply figure out the algorithm. In World War II, U.S. codebreakers were able to determine the inner workings of the Japanese code machines without having one of the machines in their possession.

In modern times, a company called Gemstar Development created a code that converted date, time, and channel indicators into a single code number. These code numbers were published in TV listings as "VCR+." People who bought a GemStar control box could program their VCRs simply by punching in the numbers, simplifying the process and thus benefiting people who owned the product. Only the Gemstar box knew how to decrypt the code numbers. But Ken Shirriff, Curt Welch, and Andrew Kinsman broke the Gemstar algorithm, and they published it in the July 1992 issue of *Cryptologia,* a trade journal. Now, anyone who wants to decode those numbers

continues

(such as VCR manufacturers) can do it without buying a Gemstar control box.

Another example is RC4, an algorithm invented in 1987 but never published. Cryptanalysts and other experts studied it and determined that RC4 was a good way to keep data secret. But the company that created it, RSA Data Security, never made the inner workings of the RC4 algorithm public. This secrecy was for monetary and not security reasons; the company hoped that by keeping it secret no one else would implement and sell it. In 1994, anonymous hackers posted the algorithm on the Internet. How did they figure it out? It was probably by stepping through a copy of the object code with an assembly language debugger. Incidentally, RC4 is now used as part of *Secure Socket Layer* (SSL), the World Wide Web's secure communication protocol (see Chapter 7). RC4 is arguably the most commonly used symmetric cipher, even more so than DES, discussed later in this chapter in the section "Digital Encryption Standard."

If a cryptographic system is hardware-based, engineers open it and look at the internals. In 1998, David Wagner and Ian Goldberg, at the time graduate students at the University of California at Berkeley, opened a supposedly secure digital cell phone and cracked its code.

Sometimes it is possible to keep an algorithm secret long enough to be effective, but eventually the enemy figures it out. For example, in World War II, the U.S. Army used Navajo soldiers to communicate. They simply spoke in Navajo. The Japanese military did not have anyone in its employ who spoke Navajo, nor did it have dictionaries or other reference material. The encryption worked because the algorithm (the Navajo language itself) was kept secret.

Now, of course, any large military has linguists on staff who either know or can easily learn any language used to encrypt secrets.

Answer 2: You Can't Make Money Developing Secret Algorithms

Gemstar did make money for a while using a secret algorithm, but only until someone cracked it. The ultimate problem, though, goes deeper. Think about it this way: How can you sell something without letting buyers see what they're buying?

continues

Suppose, for example, that you sell a software cryptographic system to an e-mail vendor, enabling it to encrypt messages. How could you prevent this client, or anyone else, from looking at your code? There are plenty of ways to reverse-engineer software, as shown in the RC4 story.

"Fine," you may counter, "I won't sell my algorithm to just anyone. I'll make sure that only people I trust can use it." Is it possible to trust enough people to make money that way? And how are your trusted clients going to use your algorithm? About the only thing they could do so is store their data and talk to each other. But people want to communicate with others who do not purchase their algorithm from the same vendor. As a result, the algorithms must be standardized, and that means they must be public.

The other problem with trying to sell algorithms arises on the buyer's side of the arrangement. If you want to use cryptography, you must employ a hardware device or a software program. The problem is this: Just as you have access to the product, so do attackers. Where did you get your hardware or software—a retail software store, a business-to-business vendor? Attackers can go to the same source and get their own copies.

In short, if you use your own algorithm and want to keep it secret, you can't sell it. As a result, you can't make any money.

Answer 3: Publicly Known Algorithms Are More Secure

Let's say you're the purchasing agent for your company and it's up to you to decide which cryptographic algorithm to buy. Your company will use this algorithm to store data and communicate securely. Two sales reps offer their products. One warns, "This algorithm is secure as long as the attacker does not know its inner workings." The other proclaims, "You can tell attackers what the algorithm is and how long the key is, but they can never retrieve your sensitive data without the key."

Which one would you buy?

If it is possible to build a cryptographic system in which the algorithm is completely known, and if attackers still can't break it without the key, isn't that system more secure than one that can be broken if the algorithm is uncovered? Well, it is possible to build such cryptographic systems.

continues

When algorithms are made public, cryptanalysts and computer engineers get a chance to examine them for weaknesses. If an algorithm is vulnerable, you can choose not to use it. Otherwise, you can be confident that your data is safe. If an algorithm is kept secret, on the other hand, analysts will not be able to find any weaknesses it may have. Does that mean it has no weaknesses? Not necessarily; it simply means that you don't know whether or not it is vulnerable. Maybe a cracker, lurking somewhere in a basement, has obtained a copy of the algorithm (remember, they always do) and has already found a successful attack. But this cracker has decided not to share the information. If you use the secret algorithm, all your data is compromised but you don't know it.

When an algorithm is made public, however, that's no guarantee that it is secure. Maybe analysts have not yet found the weakness, and the basement-dwelling cracker has found it. But great minds thrive on finding flaws in public cryptographic systems. There's prestige (and sometimes a little money) in finding chinks in the armor. If the cryptographic community cannot find something wrong with an algorithm, there's a good chance that no one else will.

Sources: See David Kahn's *The CodeBreakers* for the histories of the Enigma, Purple, and Navajo codetalkers. See Cecil Adams' *Return of the Straight Dope* for the Gem-Star story.

To answer that question, let's consider what the word "random" means. You probably have an intuitive idea of randomness, and most likely it's correct. To be more formal than intuition, we could put it this way: "If someone knows what the *current* numbers are, is it possible to predict the *next* numbers?" To put it the way cryptographers prefer, random values are simply sets of numbers that pass statistical tests of randomness and are unrepeatable.

Suppose that you choose a few thousand numbers and ask a mathematician, "Are these numbers random?" To simplify things and to conform to computer conventions, you make the numbers *binary*, meaning that they are sequences of 1's and 0's. The mathlete will draw on a set of tests

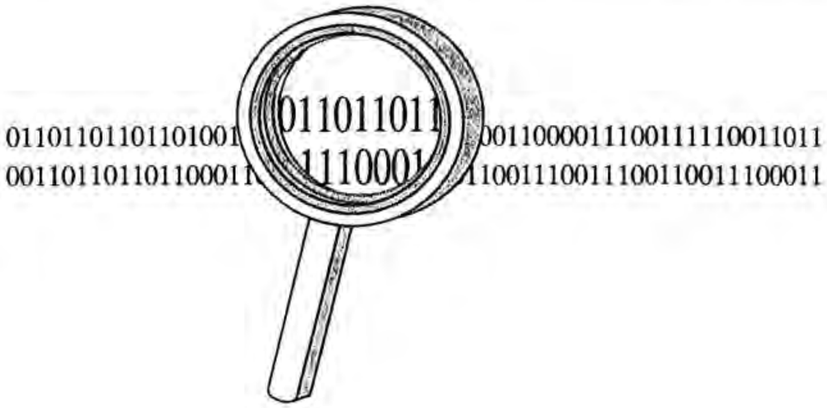


Figure 2-5

Testing numbers for randomness. Here, the pattern 110 appears too often, so it fails

that examine the numbers. Among these tests (see Figure 2-5) are questions such as these: Are there roughly the same count of 1's and 0's? Do some patterns of 1's and 0's appear "too often"? Do some patterns of 1's and 0's appear "not often enough"? If the numbers pass the tests, we say that the numbers are probably random. "Probably" random? Can't we say "definitely" random? No, we can't, and in a few paragraphs you'll see why.

A Random Number Generator

If you have a few thousand numbers, you can test them for randomness. But where do you get those few thousand numbers in the first place? One source is a *random number generator* (RNG). These devices work by gathering numbers from various kinds of unpredictable inputs, such as by measuring radioactive decay, examining atmospheric conditions in the vicinity, or calculating minute variances in electrical current. These numbers pass the tests of randomness.

If you ask the machine for a second group of numbers, you will virtually never receive the same sequence again. That's because the output is based on input that's always changing. The numbers are unrepeatable.

So to return to our original definition, we can ask, "Can anyone predict what the next numbers will be?" To do that, someone would have to predict the minor variations in the radioactive decay, atmospheric conditions, or electricity of the current. These are things we assume that no one can do.

Intel produces an RNG that uses system thermal noise as its variable and unpredictable input. Currently, this device does not ship automatically with every Pentium-based PC, although maybe in the future it will. Other companies (such as nCipher, Chrysalis, and Rainbow) sell devices known as cryptographic accelerators (discussed in Chapters 3 and 9). These devices come with RNGs.

A Pseudo-Random Number Generator

Where can you get random numbers if you don't have an RNG? It turns out there are algorithms called *pseudo-random number generators* (PRNGs). Just as there are algorithms that convert plaintext into ciphertext, there are algorithms that produce what are called "pseudo-random" numbers.

If you use one of these algorithms to generate a few thousand numbers and apply the statistical tests, the numbers pass. What makes these numbers pseudo-random and not random is that they are repeatable. If you install the same PRNG on another computer, you get the same results. If you run the program two weeks later, you get the same results.

This is one reason we say that numbers that pass statistical tests of randomness are "probably" random. Even if they pass, do we know whether they are repeatable? The math tests give us only part of the answer.

If the numbers are repeatable, what good is a PRNG? The answer is that you can change the output by using what is known as a *seed*. Just as RNGs take input (radioactive decay, atmospheric conditions, electrical variances), a PRNG takes input (the seed). If you change the input, you change the output. With RNGs, the input is constantly changing on its own, unpredictably. With a PRNG, it's up to you to make sure the input changes each time you want to generate new numbers.

What is this seed? In the real world, a seed can be lots of things: the time of day down to the millisecond, various constantly changing computer state measurements, user input, and other values. Maybe you've seen a user-input seed collector. An application may ask you to move the mouse around. At selected intervals, the program looks at where, on the screen, the arrow is located. This value is a pair of numbers: how many pixels up from the bottom of the screen and how many pixels over from the left. Any one input is not sufficient, but if you put them all together you have unpredictability (see Figure 2-6).

You may be thinking, "Why use a PRNG to generate the numbers? Why not just use the seed?" There are two main reasons. The first reason is the need for speed. Seed collection is often time-consuming. Suppose you need

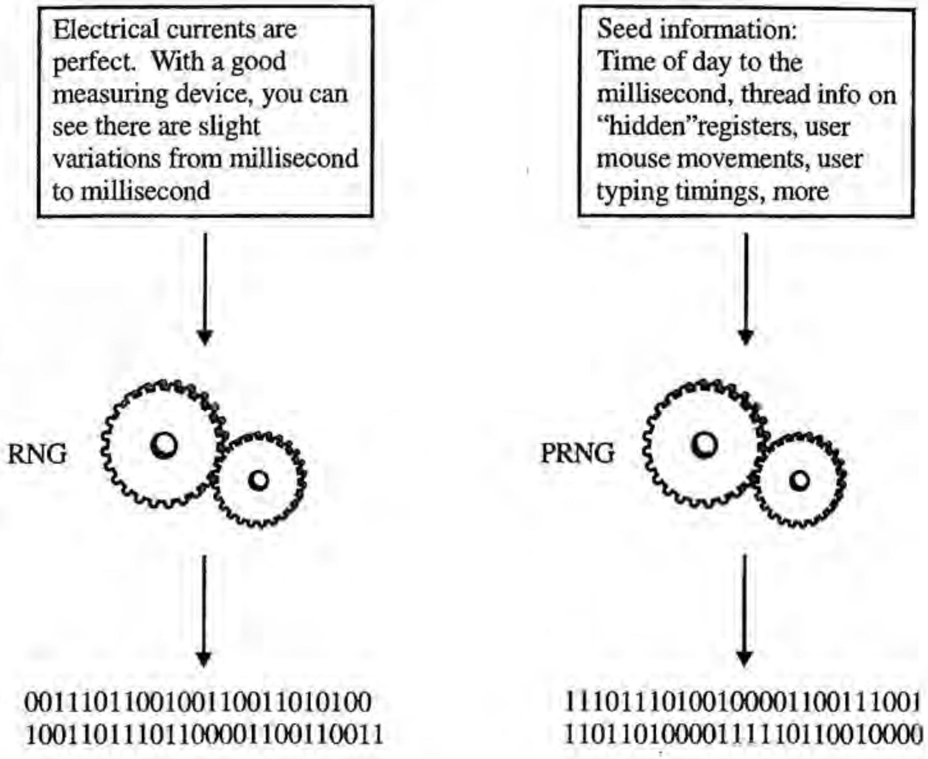


Figure 2-6

A random number generator (left) collects unpredictable information and converts it into random numbers. A pseudo-random number generator (right) collects seed information and converts it into numbers that pass statistical tests of randomness but can be repeated

only a few thousand bits of random data. A seed collector may take several minutes to gather the necessary numbers. When was the last time you waited several minutes for a program to do something without getting frustrated? To save time, you can gather 160 or so bits of seed (which may take little time), feed it to the PRNG, and get the required thousands of bits in a few milliseconds.

The second reason to use a PRNG is *entropy*, a term that describes chaos. The greater the entropy, the greater the chaos. To put it another way, the more entropy, the more random the output. Suppose you want 128 bits of entropy. A seed may have that, but it is spread over 2,400 bits. For example, the time of day down to the millisecond is represented in 64 bits. But the year, the month, the date, and maybe even the hour and minute might be easy to guess. The millisecond—two or three bits of the time of day—is where the entropy is. This means that out of 64 bits of seed, you have 2 bits of entropy. Similarly, your other seed data may suffer the same condition. A PRNG will take that 2,400 bits of seed and compress it to 128 bits.

Well, then, why not take the seed and throw away the low-entropy bits? In a sense, that's what a PRNG does. You can do it, or you can have a PRNG do it, and the latter means less work for you.

By the way, most PRNGs use *message digests* to do the bulk of the work. We talk about the details of digests in Chapter 5, but for now, let's just say that they are the "blenders" of cryptography. Just as a blender takes recognizable food and purees it into a random, unrecognizable blob, a message digest takes recognizable bits and bytes and mixes them up into a random, unrecognizable blob. That sounds like what we look for in a PRNG.

A good PRNG always produces pseudo-random numbers, regardless of the seed. Do you have a "good" seed (one with lots of entropy)? The PRNG will produce numbers that pass tests of randomness. Do you have a "bad" seed (or no seed at all)? The PRNG will still produce good numbers that pass the tests.

Then why do you need a good seed? The answer is given in the next section.

Attacks on Encrypted Data

Someone wants to read the data you've encrypted. This person, known as the *attacker*, must first decrypt the data. To do that, the attacker must either identify the key or break the algorithm.

Attacking the Key

If attackers can figure out what your key is, they can decrypt your data. One approach, the *brute-force attack,* is to try every possible key until the right one is identified. It works this way. Let's say your key is a number between 0 and 100,000,000,000 (one hundred billion). The attacker takes your ciphertext (perhaps only 8 or 16 bytes' worth) and feeds it to the decryption algorithm along with the "alleged key" of 0. The algorithm does its job and produces a result. If the resulting data appears reasonable, 0 is probably the correct key. If it's gibberish, 0 is not the true key. In that case, you try 1, and then 2, 3, 4, and so on (see Figure 2-7).

Remember, an algorithm simply performs its steps, regardless of the input. It has no way of knowing whether the result it produces is the correct one. Even if the value is close to the key, maybe off by only 1, the result is gibberish. So it's necessary to look at the result to tell whether it might be the key. Smart attackers write programs to examine the result. Is it a series of letters of the alphabet? Yes? Pass this key to the attacker. No? Try the next key.

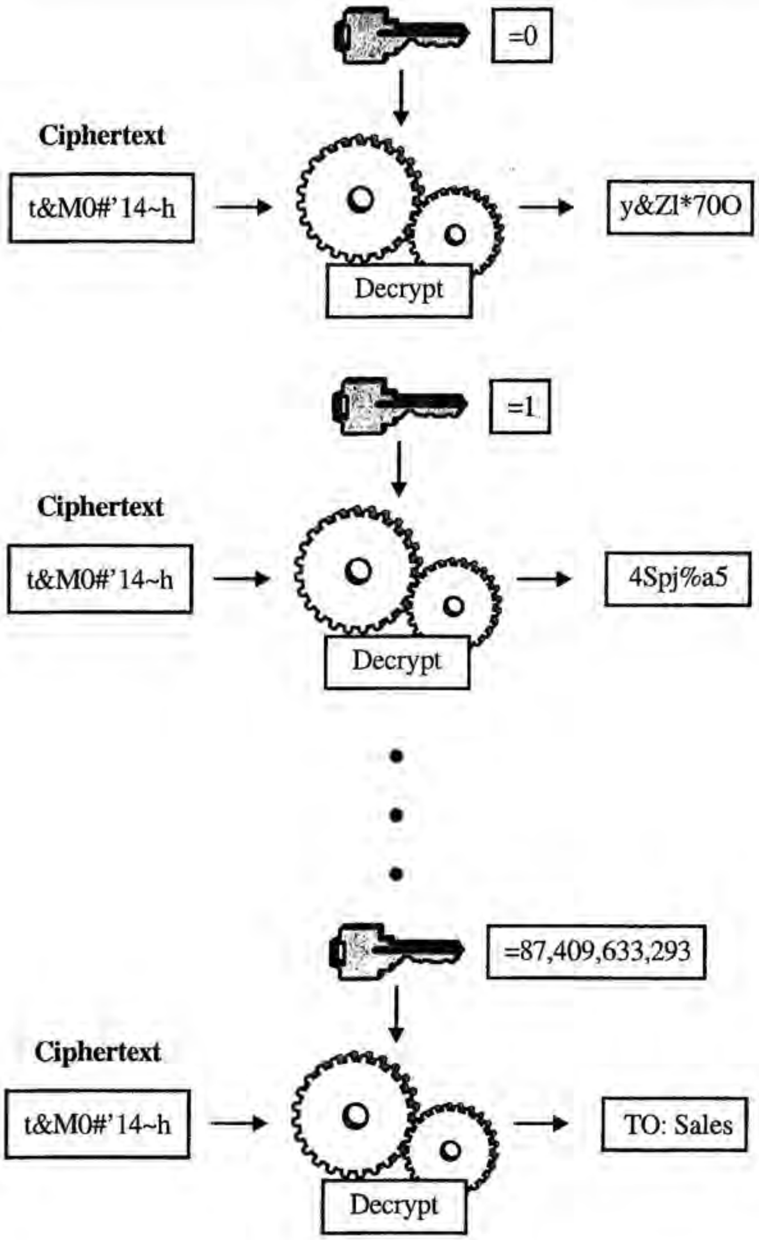


Figure 2-7

The brute force attack. If you know that the key is a number between 1 and 100,000,000,000, you try each number in turn until a number produces something that's not gibberish

It usually takes very little time to try a key. The attacker can probably write a program that tries many keys per second. Eventually, the attacker could try every possible number between 0 and 100 billion, but that may not be necessary. Once the correct key is found, there's no need to search any more. On average, the attacker will try half of all possible keys—in our example, 50 billion keys—before finding the correct one. Sometimes it takes more time, sometimes less, but, on average, about half the possible keys must be tried.

How long would it take an attacker to try 50 billion keys? Three years? Three days? Three minutes? Suppose you want to keep your secret safe for at least three years, but it takes an attacker only three minutes to try 50 billion values. Then what do you do? You choose a bigger range. Instead of finding a number between 0 and 100 billion, you find a number between 0 and 100 billion billion billion billion. Now the attacker will have to try, on average, many more keys before finding the right one.

This concept of the range of possible keys is known as *key size*. Gold is measured in troy ounces, atoms are measured in moles, and cryptographic keys are measured in bits. If someone asks, "How big is that key?" the answer might be 40 bits, 56 bits, 128 bits, and so on. A 40-bit key means that the range of possible values is from 0 to about 1 trillion. A 56-bit key is 0 to about 72 quadrillion. The range of a 128-bit key is so large that it's easier just to say it's a 128-bit key (see Figure 2-8).

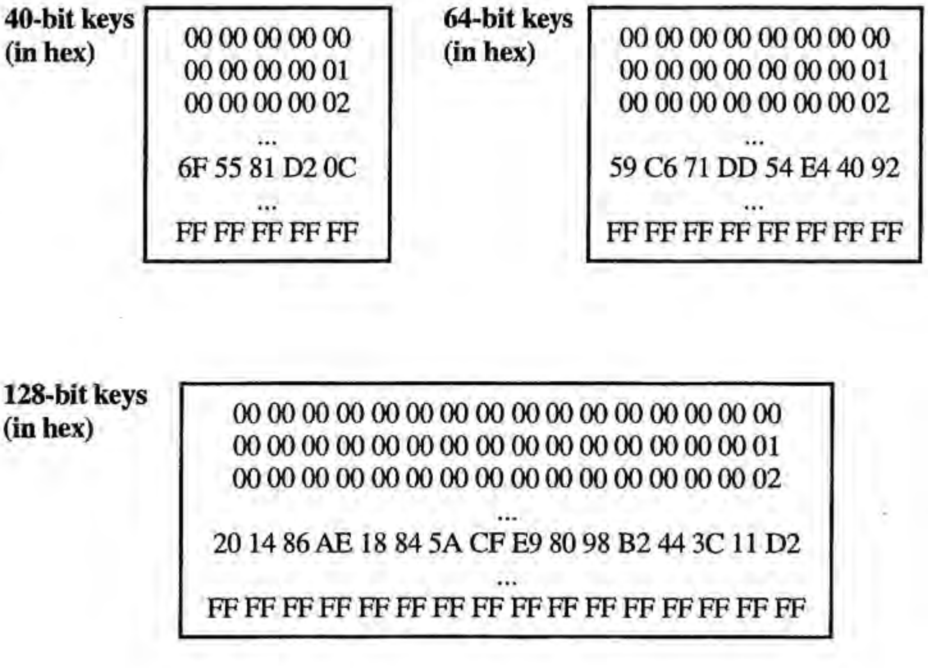


Figure 2-8

The larger the key size, the greater the range of possible values a key can be. Each bit in each position, whether 0 or 1, is important

Each bit of key size you add doubles the time required for a brute-force attack. If a 40-bit key takes 3 hours to break, a 41-bit key would take 6 hours, a 42-bit key, 12 hours, and so on. Why? Each additional bit doubles the number of possible keys. For example, there are eight possible numbers of size 3 bits:

```
000  001  010  011  100  101  110  111
```

These are the numbers from zero to seven. Now add one more bit:

```
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
```

Every number possible with 3 bits is possible with 4 bits, but each of those numbers is possible "twice": once with the first bit not set, and again with it set. So if you add a bit, you double the number of possible keys. If you double the number of possible keys, you double the average time it takes for brute-force attack to find the right key.

In short, if you want to make the attacker's job tougher, you choose a bigger key. Longer keys mean greater security. How big should a key be? Over the years, RSA Laboratories has offered challenges. The first person or organization to crack a particular message wins a money prize. Some of the challenges have been tests of brute-force time. In 1997, a 40-bit key fell in 3 hours, and a 48-bit key lasted 280 hours. In 1999, the Electronic Frontier Foundation found a 56-bit key in 24 hours. In each case, a little more than 50 percent of the key space was searched before the key was found. In January 1997, a 64-bit challenge was issued. As of December 2000, it has still not been solved.

In all these situations, hundreds or even thousands of computers were operating cooperatively to break the keys. In fact, with the 56-bit DES challenge that the Electronic Frontier Foundation broke in 24 hours, one of those computers was a custom-built DES cracker. This kind of computer does only one thing: check DES keys. An attacker working secretly would probably not be able to harness the power of hundreds of computers and might not possess a machine built specifically to crack a particular algorithm. That's why, for most attackers, the time it takes to break the key would almost certainly be dramatically higher. On the other hand, if the attacker were a government intelligence agency with enormous resources, the situation would be different.

We can devise worst-case scenarios. Let's use as our baseline an exaggerated worst-case scenario: examining 1 percent of the key space of a 56-bit key takes 1 second, and examining 50 percent takes 1 minute (see Table 2-1). Each time that we add a bit to the key size, we double the search time.

Currently, 128 bits is the most commonly used symmetric-key size. If technology advances and brute-force attackers can improve on these numbers (maybe they can reduce the 128-bit times to a few years), then we would need to use a 256-bit key.

You may be thinking, "Technology is always advancing, so I'll have to keep increasing key sizes again and again. Won't there come a time when I'll need a key so big it becomes too unwieldy to handle?" The answer is

Table 2-1

A Worse Than Worst-Case Scenario: How Long a Brute-Force Attack Will Take for Various Key Sizes

Bits	1 percent of Key Space	50 percent of Key Space
56	1 second	1 minute
57	2 seconds	2 minutes
58	4 seconds	4 minutes
64	4.2 minutes	4.2 hours
72	17.9 hours	44.8 days
80	190.9 days	31.4 years
90	535 years	321 centuries
108	140,000 millennia	8 million millennia
128	146 billion millennia	8 trillion millennia

that you'll almost certainly never need a key longer than 512 bits (64 bytes). Suppose that every atom in the known universe (there are about 2^{300} of them) were a computer and that each of these computers could check 2^{300} keys per second. It would take about 2^{162} millennia to search 1 percent of the key space of a 512-bit key. According to the Big Bang theory, the amount of time that has passed since the universe came into existence is less than 2^{24} millennia. In other words, it is highly unlikely that technology will ever advance far enough to force you to use a key that's "too big."

That may not matter, though, because there's another attack on the key. Instead of trying to reproduce the key, attackers can try to reproduce the PRNG and seed that were used to produce the key. It works like this. Attackers know the particular PRNG and seed-collection method you used. (Remember, as discussed earlier in this chapter in "Historical Note: They Always Figure Out the Algorithm," the attacker will always know your algorithms and methods.) If attackers can guess your seed, they can seed the PRNG and produce the same key. If you used a small seed, attackers will try every possible value until they find the correct one. This happened to Netscape, as described in "Historical Note: Netscape's Seed."

Your defense against this kind of attack is to use a good seed. A PRNG will always produce good pseudo-random numbers regardless of seed. But the seed must also be strong enough to withstand a brute-force attack.

Historical Note: Netscape's Seed

Symmetric-key cryptography is one component of SSL (see Chapter 7), which was invented by researchers at Netscape. Not surprisingly, Netscape offered an implementation of SSL that is part of all Netscape browsers (after version 1.0).

At some point in an SSL session, the code must generate a key. To do so, Netscape's implementation uses a PRNG. In version 1.1 (released in 1995), the code collected the time of day, the process ID, and the parent process ID as the seed for the PRNG.

Ian Goldberg and David Wagner (remember them from the earlier historical note?) decided to test how good a seed these three sources would produce. They discovered that the process IDs were easy to capture if one had access to the computer. If one did not have access to the computer, all it took was a little brute-force testing because each ID was only 15 bits. The time of day? Well, the year, the month, the date, and even the hour and minute were known; an attacker simply had to look at when the SSL session occurred. The second? There were only 60 possible values (Netscape used time of day only down to the second and not the millisecond).

On September 17, 1995, Goldberg and Wagner reported to the Cypherpunks newsgroup that they could find the seed, and hence the key, in less than a minute. Whether the key was 40 bits or 128 bits, it took only one minute.

Netscape fixed the problem in version 2.0 by adding more seed. Each platform (Windows, Mac, and UNIX) has different seed sources, but among the many platform-dependent seeds Netscape now uses are cursor or mouse position, memory status, last key pressed, audio volume, and many others.

Sources: Gary McGraw and John Viega, "Make Your Software Behave: Playing The Numbers," *Reliable Software Technologies,* April 4, 2000.

Keith Dawson, "Tasty Bits from the Technology Front," http://www.tbtf.com, Sept. 20, 1995.

Taher El Gamal, letter to the Internet community posted on many Web sites, Sept. 25, 1995. El Gamal was, at the time, director of security for Netscape.

Breaking the Algorithm

Suppose that someone figured out that with a given algorithm, every 14th bit of a given ciphertext is the same as every 12th bit of its plaintext. In other words, if the 14th bit of ciphertext is 1, the 12th bit of plaintext is 1, the 28th bit of ciphertext is 0, the 24th of plaintext is 0, and so on, no matter what the key. Furthermore, the attacker sees that if certain combinations of bits appear in certain locations in the ciphertext, a corresponding portion of the plaintext must be another pattern.

If an algorithm had such weaknesses, an attacker could look at the ciphertext and decipher parts of the plaintext even without knowing the key. This knowledge might be enough to enable the attacker to recover enough of the original message to do damage (see Figure 2-9).

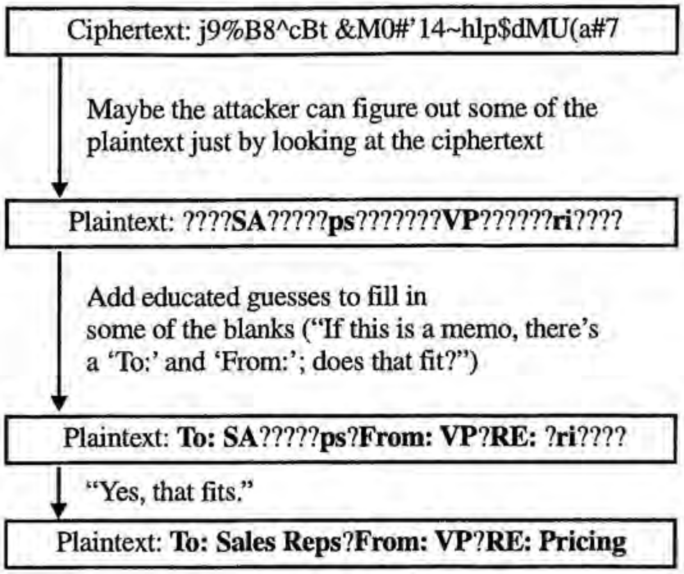


Figure 2-9

If an algorithm has a weakness, an attacker might figure out portions of plaintext without the key, reconstructing most or all of the message

Here's another possible weakness. Suppose the attacker knows what some of the plaintext and its corresponding ciphertext is. And suppose this attacker is able to therefore deduce the key. But if the attacker knows what the plaintext is, why bother figuring out the key? The answer is that the attacker might know, or be able to guess, only a portion of the plaintext. Recall the memo at the beginning of the chapter. An attacker might see the ciphertext, realize it's a Word for Windows document, and guess some of the control characters at the beginning.

Furthermore, the attacker guesses the document is a memo from the conventional "TO:", "FROM:", and, "RE:" In short, if someone can compute the key from a chunk of ciphertext and its corresponding plaintext, the rest of the message will follow. This is known as a *known-plaintext* attack. Obviously, you don't want to use an algorithm that might be susceptible to such an attack.

Measuring the Time It Takes to Break Your Message

How long will your secret remain secret? The answer is, as long as it takes the attacker to break it. The attacker has two kinds of tools: the brute-force attack and attacks that exploit weaknesses in your algorithm.

In analyzing the security of your message, a key question is how long would a successful brute-force attack take. There's no rigid, specified time, since the attacker may get lucky and find it early or may get unlucky and find it later, but as shown in Table 2-1, you can estimate the variables based on worst-case scenarios. In general, the bigger the key, the longer a brute-force attack will take. But if the algorithm is weak, it doesn't matter how long the key is. The statement "Longer keys mean more security" doesn't apply to a weak algorithm. The point is this: If you pick a weak algorithm, you have no control over how strongly your secret is protected.

So the best strategy is to pick an algorithm that is not weak and further deter an attacker by using a longer key.

That statement may seem so obvious that it's not worthwhile even to mention it. If you're curious about what happens when people overlook these obvious protections, however, read "Crypto Blunders" in the accompanying CD for a couple of stories on using weak algorithms and small keys.

Symmetric Algorithms: The Key Table

Virtually all symmetric ciphers use the key to build a *key table*, which is usually a pseudo-random array of a particular size in a particular format. This process is known as *key setup*, or initialization. It's the key table that does the encryption.

Why have a key table? One reason is that you might want to use keys of varying lengths depending on the application. The algorithm needs a *key value* that is the same size from one use to the next, but your key might vary from 64 bits to 128 to 192 or even 256 bits. For that reason, you build a key table (which is bigger than the biggest possible key size) from the key. It's easier to create a constant-sized key table at the beginning of your encryption session than to do it repeatedly while encrypting data.

Another reason to use a key table is to prevent attacks on the algorithm. Recall that there are two ways to break security: a brute-force attack and attacks on an algorithm's weaknesses. If you use a big, pseudo-random key table, it's easier to do serious scrambling. With good scrambling, the ciphertext looks nothing like the plaintext. If the algorithm cannot do a good job of creating gibberish unless it has a good key, that is be an algorithmic weakness. A good algorithm will simply expand the key into a bigger value and make sure that no matter what key it's given, the key table is random. An attacker could try a brute-force attack on the key table, but that would be more time-consuming than an attack on the key.

The user should give the algorithm a good key. But even with a bad key, it is possible to create a good key table. Just as a PRNG produces good numbers no matter what the seed is, a good encryption algorithm produces a good key table no matter what the key is. With a good key table, the algorithm produces a good scramble, the resulting ciphertext is not at all close to the plaintext, and the attacker cannot exploit an algorithm's weakness.

Symmetric Algorithms: Block Versus Stream Ciphers

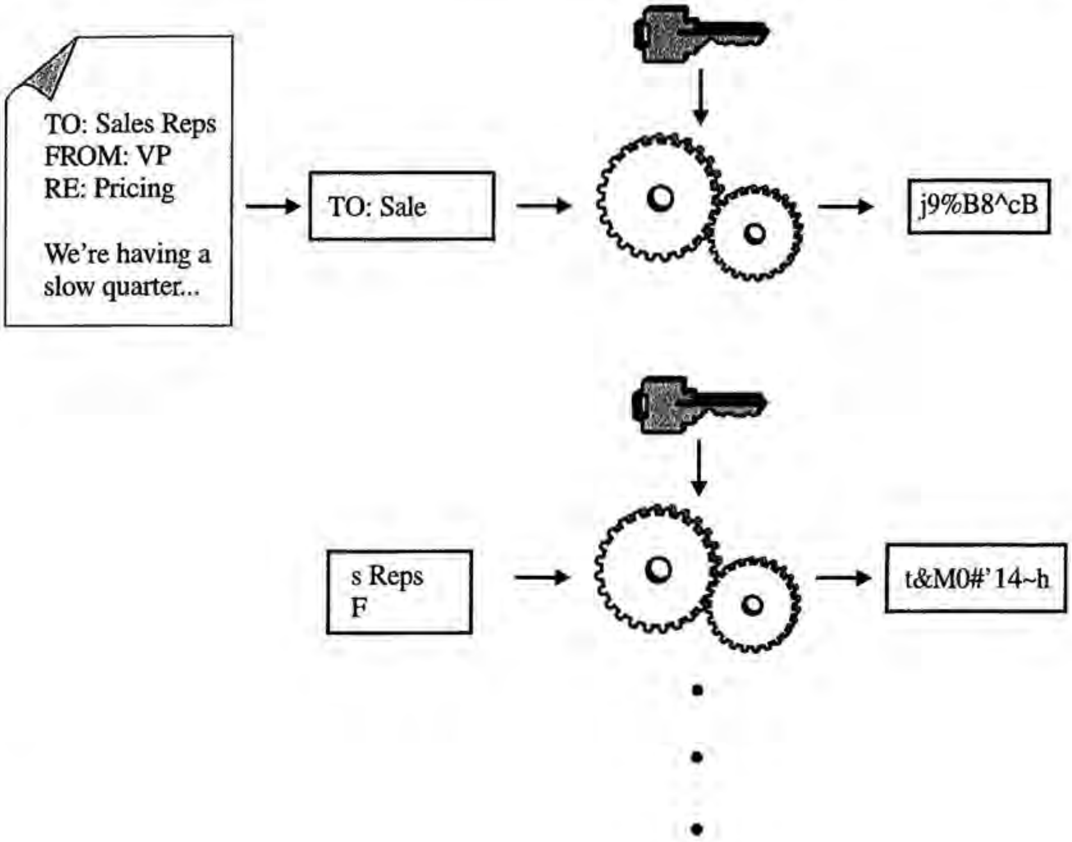
If you're using symmetric-key cryptography, how do you choose a good algorithm? There are two types of symmetric-key algorithms: block and stream ciphers. What are they, and which is better?

Block Ciphers

A *block cipher* operates on blocks of data. When you give the algorithm a chunk of data to encrypt or decrypt, it breaks the plaintext into blocks and operates on each block independently (see Figure 2-10). Usually, blocks are 8 or 16 bytes long.

Figure 2-10

A block cipher grabs each block of the input data (usually 8 or 16 bytes) and uses the key table to produce a unique block of output, continuing until all the blocks are encrypted



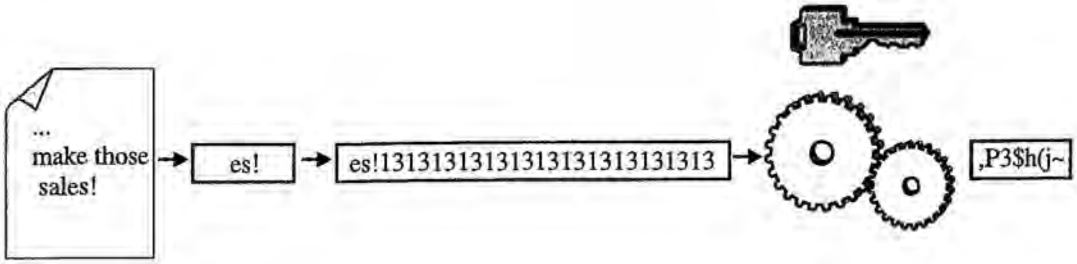
Suppose that your plaintext is 227 bytes long and the cipher you're using operates on 16-byte blocks. The algorithm grabs the first 16 bytes of data, encrypts them using the key table, and produces 16 bytes of ciphertext. Then it starts over, encrypting the next 16 bytes of plaintext. No matter which block it is working with, the cipher encrypts it by starting over from scratch. The key table does not change from block to block.

After encrypting 14 blocks (224 bytes), the algorithm is left with 3 more bytes. But your block cipher cannot operate on 3 bytes; it needs 16 bytes. To encrypt the last 3 bytes, you must *pad* the data: add extra bytes to an incomplete block to make it complete. Whoever decrypts the ciphertext must be able to recognize (and ignore) the padding.

The most popular padding scheme determines the number of bytes to be padded and repeats that value in the final bytes in the data. In our example, the padding scheme must add 13 bytes to the plaintext so that it has a full block. So it repeats the byte "13" in each of the final 13 otherwise empty spaces. During decryption, you look at the last byte of decrypted data; this byte, a number from 1 to 16, indicates how many pad bytes have been added. In this example, after decrypting, we would know that the last 13 bytes of data should be discarded (see Figure 2-11). (Each of the last 13 bytes should be the number 13, so as an extra check, we make sure that each of them is 13.) If the length of the plaintext had been a multiple of 16, there would have been no need to pad. Nevertheless, it makes sense to always pad your data. Then, when decrypting, you know that the last byte decrypted is indeed a pad byte. To do that, you tack on 16 bytes, each of them the number 16.

Figure 2-11

When the last block of plaintext ends in blank bytes, use padding to bring it up to size



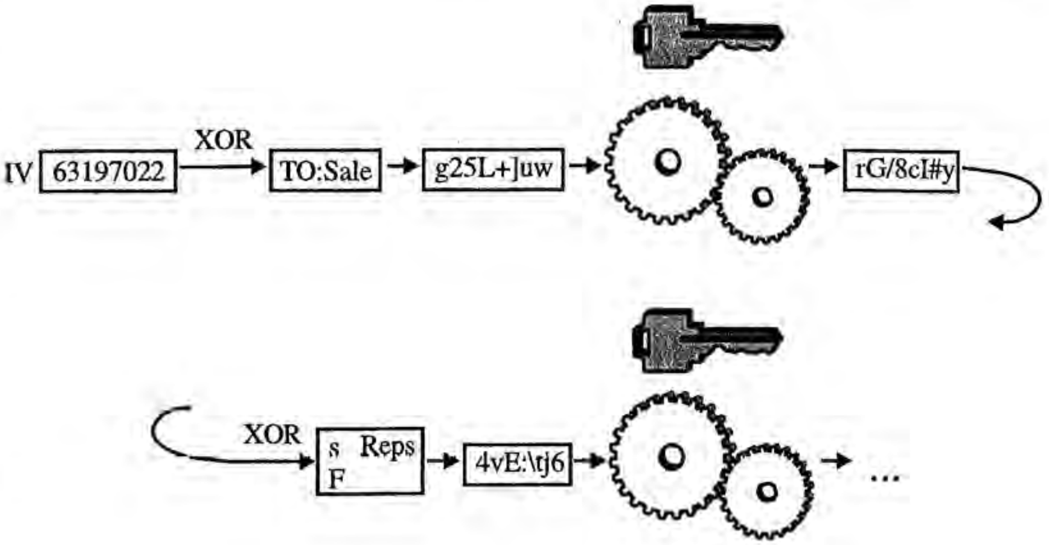
Remember the known-plaintext attack? If an algorithm is susceptible, that doesn't mean an attacker will automatically be able to break a message; it's necessary to find a plaintext/ciphertext pair first. The last block of data might be that known plaintext, because it contains padding. Of course, it's easy to simply use an algorithm that is not susceptible to the known-plaintext attack.

One problem with block ciphers is that if the same block of plaintext appears in two places, it encrypts to the same ciphertext. In our printing machinery company memo, for example, the phrase "slow third quarter" may show up a number of times. Each time the first 16 bytes of that phrase is encrypted, it will produce the same ciphertext, and an attacker might identify this repeated pattern. To avoid having these kinds of copies in the ciphertext, you can use *feedback modes*. A number of these modes are discussed in the FAQ contained in the accompanying CD.

The most common feedback mode is *cipher block chaining* (CBC), shown in Figure 2-12. In this scheme, you XOR the current block of plaintext with the preceding block of ciphertext (see "Technical Note: XOR" later in this chapter). For the first block of plaintext, there is no preceding block of ciphertext, so you XOR with an *initialization vector* (IV). When you decrypt the data, you copy a block of ciphertext, decrypt it, and XOR

Figure 2-12

Cipher block chaining. The first block of plaintext is XOR'd with the IV and then encrypted. Each successive block is XOR'd with the preceding block of ciphertext.



the result with the preceding block of ciphertext (which you saved right before you decrypted it). This technique ensures that any duplicate block in the plaintext does not encrypt to the same ciphertext. That's all it does. It adds no other security. The encryption algorithm provides the security.

Stream Ciphers

To understand stream ciphers, the second type of symmetric-key algorithm, you need to first understand the cryptographic technique called a *one-time pad,* which is popular with spies. In one variation of this technique, you generate a bunch of random numbers, each from 0 to 25. Then you print two copies of the series. That's the "pad." One copy stays at your headquarters, and the spy takes the other copy out into the field.

To send a message back home, the spy encrypts each letter of the message with a number on the pad. The first letter of the message is encrypted with the first number on the pad, the second letter with the second number, and so on. Encryption is simply a matter of adding a numeric value assigned to the letter plus the number. Here's how the numeric value is assigned. If the plaintext letter is *G* and the number on the pad is 11, the ciphertext letter is *R* (*R* is the eleventh letter after *G*, or $G + 11 = R$). If the plaintext letter is *Y* and the number is 4, the ciphertext letter is *C*, or $Y + 4$ (*Y, Z, A, B, C*; when you reach the end of the alphabet, you start over at *A*).

Technical Note: XOR

The term *XOR* stands for "exclusive OR," a type of bit manipulation. The first concept to understand is an OR. An OR is a bit manipulation that says, "Look at two bits. If one OR the other is set, set the result."

```
0 OR 0 = 0 (zero OR zero equals 0)
0 OR 1 = 1 (zero OR  one equals 1)
1 OR 0 = 1 ( one OR zero equals 1)
1 OR 1 = 1 ( one OR  one equals 1)
```

An exclusive OR says, "Look at two bits. If one is exclusively set, OR if the other is exclusively set, set the result." If both bits are set, then there's no exclusivity, so the result bit is not set.

```
0 XOR 0 = 0 (zero XOR zero equals 0)
0 XOR 1 = 1 (zero XOR  one equals 1)
1 XOR 0 = 1 ( one XOR zero equals 1)
1 XOR 1 = 0 ( one XOR  one equals 0)
```

XOR is a useful bit manipulation in cryptography because half of the time the result is 1, and the other half of the time it's 0. If one bit is plaintext, and one bit is key stream, then the key stream sometimes changes the bit and sometimes doesn't change the bit.

In grade school, we learned how to add, subtract, and multiply using columns:

```
1,482        77       204
+ 319       - 5       * 8
1,801        72      1632
```

Similarly, we can perform XOR operations on longer numbers. Computers, of course, see all numbers as binary values.

```
values as binary text                            values as hex text
    0111 0100 0110 0101 0111 1000 0111 0100         0x74 65 78 74
XOR 1001 1011 0010 1100 0110 0011 1000 0100         0x9B 2C 63 84
    1110 1111 0100 1001 0001 1011 1111 0000         0xEF 49 1B F0
```

continues

The first row (the row that begins 0111 0100) in the preceding table is the ASCII bit formation of the word "text." ASCII gives us a standard way to map characters to numbers. For example, lowercase *t* is represented as the number 0x74 (binary 0111 0100), which is decimal 116. Punctuation marks are also included; a comma, for example, is 0x2C, which is decimal 44. You see 0111 0100 and so on, but the computer sees the word "text." Suppose that word "text" is our plaintext. To encrypt it, we perform the steps the algorithm prescribes, namely XOR it with the key stream. If the second row (the row with the binary values beginning 1001 1011) is the key stream and we perform the XOR operation, what do we get? We get the bottom row (the row beginning 1110 1111)—that would be the ciphertext.

What does this ciphertext say? It says "?9??" As it happens, the first, third, and fourth characters are not standard characters (they are numbers outside the ASCII range). The second is the character 9. So the algorithm converted the "e" in "text" to a "9", but what about the other characters? Because the numbers are not standard character numbers, each computer or software package gets to decide what they mean. One computer or software package might print the ciphertext as "µ9←≡". Another computer or software package might print it as "□9□□". Whichever you use, it looks like gibberish; it's nothing like the plaintext.

If you start with the ciphertext and XOR it with the key stream, what do you get? You get the plaintext.

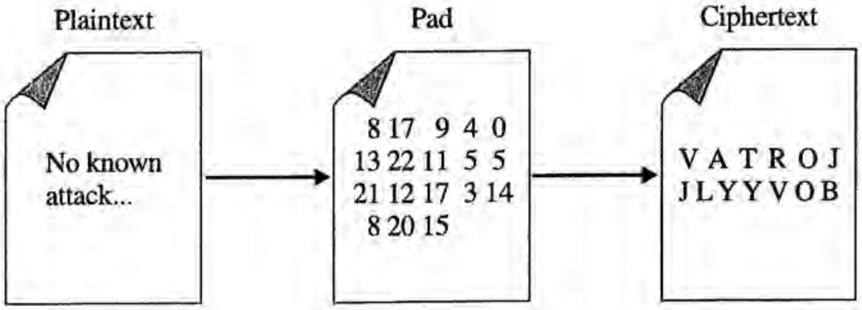
```
values as binary text                           values as hex text
    1110 1111 0100 1001 0001 1011 1111 0000        0xEF 49 1B F0
XOR 1001 1011 0010 1100 0110 0011 1000 0100        0x9B 2C 63 84
    0111 0100 0110 0101 0111 1000 0111 0100        0x74 65 78 74
```

That's another reason that the XOR operation is popular in cryptography: It's symmetric.

When the home office gets the encrypted message, the translator simply reverses the algorithm. If the ciphertext is *R* and the associated number in the pad is 11, compute R–11 = *G*. As long as the spy and the home office use the same pad, the communication will be successful. Figure 2-13 shows an example of the one-time pad. Where does the pad come from? Probably an RNG.

Figure 2-13

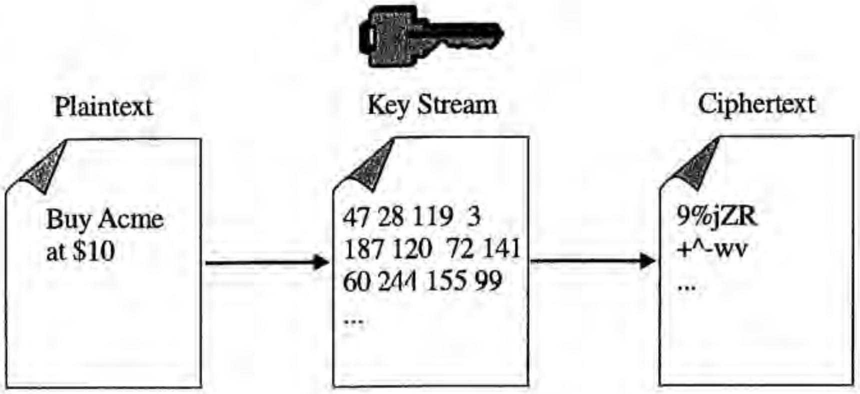
A one-time pad



A *stream cipher* is similar to a one-time pad. To encrypt data, the algorithm generates a pad based on the key. The pad can be as big as it needs to be. The algorithm will XOR the plaintext with the pad (see Figure 2-14 and the technical note on the XOR function). With the one-time pad, the spy and the home office generate a pad (actually, probably many pads) in advance. The stream cipher generates its pad on-the-fly, only when needed. In cryptography circles, the "pad" is called a *key stream*. A true pad would be random; a stream cipher produces pseudo-random values and technically can't be called a pad.

Figure 2-14

A stream cipher



Most stream ciphers work this way. First, you use the key to build a key table. Then to encrypt the data, you take one byte of plaintext, go to the key table, somehow get a byte of key stream, and XOR it with the plaintext byte. Next, you throw away the key stream byte and remix the key table. Then you get the next byte of data and continue. The key table, and hence the key stream, does not depend on the input data.

In the example of the one-time pad, the spy added numbers to letters to encrypt the data and the home office subtracted them to decrypt. A stream cipher uses the XOR operation because encrypting and decrypting are the same operation. Only one program and not two exist.

Block Versus Stream: Which Is Better?

Stream ciphers are almost always faster and generally use far less code than do block ciphers. The most common stream cipher, RC4, is probably at least twice as fast as the fastest block cipher. RC4 can be written in perhaps 30 lines of code. Most block ciphers need hundreds of lines of code.

On the other hand, with a block cipher, you can reuse keys. Remember that the stream cipher is rather like a one-time pad. "One-time" implies that you should use a pad only once (see "Crypto Blunders" on the accompanying CD for a story of multiple uses of one-time pads). Similarly, you should use a stream cipher key only once. Generally, that's not a problem, but sometimes it will be necessary to encrypt many things using the same key. For example, an e-commerce company may have a database of customer information, including credit card numbers. Rather than encrypt each entry with a different key (and hence manage hundreds or even thousands of keys), the company can encrypt all of them with one key. When one entry is needed, decrypt it with the one key. Key management is much easier when there's only one key to manage.

Another factor is standardization. Everyone has two algorithms—DES and AES-both of which are block ciphers. For reasons of interoperability, you may want an algorithm that is widely used. The entity on the other end of your data link may or may not have RC4, but it's almost a guarantee that it has DES and AES. You choose a block cipher because it's a standard.

In other words, neither type is "better." If you need to reuse keys, use a block cipher. If you must guarantee interoperability, it's best to use AES. Otherwise, use a stream cipher. Table 2-2 lists some applications and the type of cipher you might want to use with each one.

Table 2-2

Choosing an Algorithm by Application

Application	Cipher to Use	Comments
Database	Block	Interoperability with other software is not an issue, but you will need to reuse keys.
E-mail	AES	Although each e-mail message has its own key and you could use a stream cipher, you gain interoperability with all e-mail packages by using the standard AES.
SSL (secure connections on the Web)	RC4 (stream cipher)	Speed is extremely important, each connection can have a new key, and virtually all Web browsers and servers possess RC4.
File encryption (storing your files securely)	Block	Interoperability is not an issue, but you can encrypt each file with the same key and then protect that key (see Chapter 3).

Digital Encryption Standard

A computer can be programmed to perform any encryption algorithm. By the 1970s, though, it was known that the old algorithms were not very strong. They had weaknesses and were difficult to implement.

The advent of computers made it possible to throw out the old rules of cryptography and create a new paradigm. Researchers at IBM decided to develop a new algorithm for the computer age, and built on a scheme called Lucifer, an algorithm invented by cryptographer Horst Feistel. They also enlisted the help of the *National Security Agency* (NSA), the agency charged with protecting the U.S. government's secret data, a duty that includes cryptography. The fruit of the group's labor was DES.

DES is a block cipher that uses a 56-bit key—no more, no less—to build a key table. Using the key table, DES performs bit manipulations on plaintext. To decrypt ciphertext, it simply does everything in reverse.

After its introduction, DES became freely available and widely studied. Throughout the 1980s, the consensus among cryptographers was that it had no weaknesses. This meant that the fastest way to break a message encrypted with DES was to use the brute-force attack. Because a 56-bit key is a number between 0 and about 72 quadrillion, even the fastest computers took years to break a single message.

By the 1990s, though, cryptographers knew that DES couldn't last. Computers were becoming faster and eventually would be fast enough to mount a brute-force attack on a 56-bit key in a reasonable amount of time. In addition, researchers discovered potential weaknesses that led them to conclude that someday it might be possible to break the algorithm. The brute-force attack was still the fastest attack, but those potential weaknesses were troubling.

In 1999, at the RSA Conference, the Electronic Frontier Foundation broke a DES key in less than 24 hours. The world needed a replacement.

Triple DES

One widely used replacement for DES is Triple DES. The name says it all: Triple DES performs the DES algorithm three times. That's it. You run your block of data through DES using a key, and then you encrypt that result with another DES key. Then you do it a third time (see Figure 2-15).

You use three keys, each 56 bits. That's essentially the same as using one 168-bit key. You may be thinking, "If it takes 24 hours to break one key, then shouldn't it take 72 hours to break three keys?" Here's the answer. It takes 24 hours to break one key if you know you've broken it. But with Triple DES, you don't know you've stumbled onto the first key until you combine it with the other two correct keys.

Think of it this way. Suppose that the three keys are called A, B, and C, and each possible key value is numbered from 0 to 72 quadrillion. Suppose also that the correct key combination is A = 1, B = 33,717, and C = 1,419,222. An attacker could try value 0 with key A, value 0 with key B, and value 0 with key C. That doesn't produce the correct answer, so try A = 1, B = 0, C = 0. As shown in Figure 2-16, the first key is correct. But the value the attacker got from trying the three-key combination is not the right value. The correct plaintext appears only when all three keys are correct. So how can the attacker know that the first key is correct?

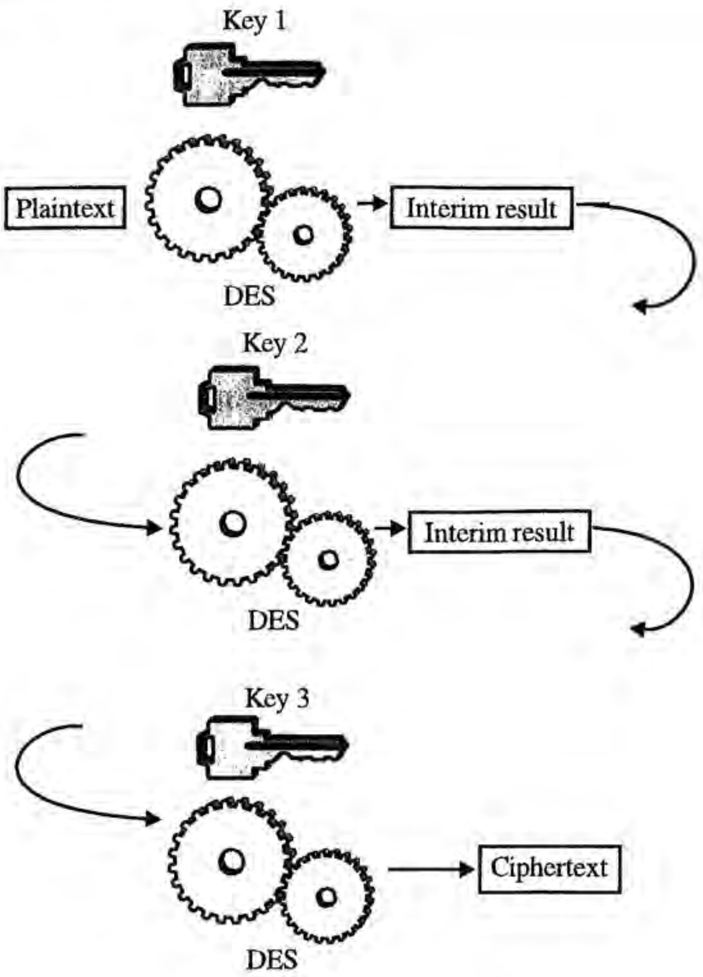


Figure 2-15

Triple DES is simply DES run on the data three times

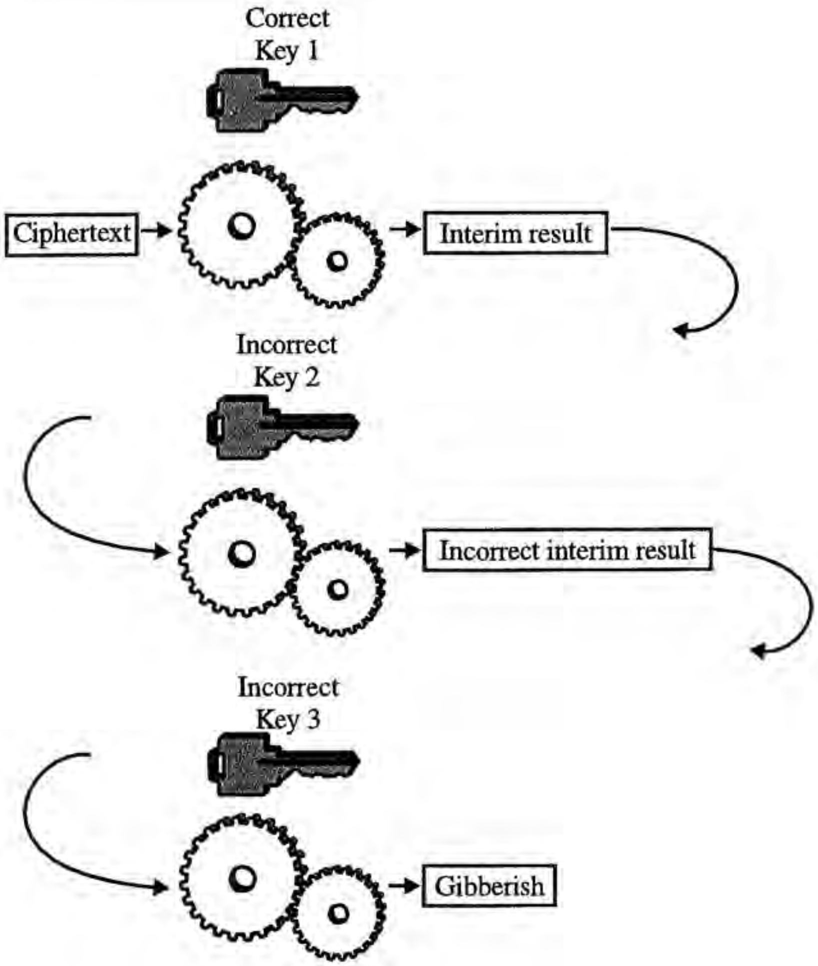
Triple DES, however, presents two problems. First, cryptanalysts have figured out a way to streamline the brute-force attack. You'd think it would require a "168-bit" brute-force attack, but there are clever ways to reduce it to the equivalent of a 108-bit brute-force attack. A key that is equivalent to 108 bits is still secure (see Table 2-1 for worst-case estimates of a 108-bit brute-force attack), but this "weakness" is troubling. Will more research expose more cryptanalytic weaknesses? Will the security of Triple DES be compromised even further?

The second problem is speed. DES takes a long time to encrypt or decrypt data, and Triple DES is three times as slow. Some applications need high-speed throughput of many megabytes worth of information. Triple DES reduces the performance so much that some applications cannot function.

For these two reasons, people needed a new algorithm.

Figure 2-16

To break Triple DES, you must know all three keys



Commercial DES Replacements

In response to the key size and performance problems of Triple DES, many cryptographers and commercial companies developed new block ciphers. The most popular offerings were RC2 and RC5 from RSA Data Security, IDEA from Ascom, Cast from Entrust, Safer from Cylink, and Blowfish from Counterpane Systems.

All these algorithms were faster than Triple DES, and they were able to operate with variable-sized and bigger keys. Whereas DES and Triple DES keys require fixed-length keys, the new algorithms could be made stronger. Recall that you can choose a key size that is big enough to make your cryptographic system immune to the brute-force attack or at least to make the brute-force attack unfeasible. At one time, a 56-bit key was big enough. But when that was no longer secure enough, 64 bits was a popular key size. Even though DES cannot increase its key size, the commercial replacements can.

The various commercial DES replacements caught on to some degree, and companies built products using the algorithms. But none became a worldwide standard comparable to DES and Triple DES.

In response, the U.S. government, through the *National Institute of Standards and Technology* (NIST), set about creating a new standard. The idea was to name a particular algorithm as the U.S. government standard. Once the U.S. government adopted a standard, the thinking went, the rest of the world would almost certainly follow.

Advanced Encryption Standard

The NIST plan was formally announced on January 2, 1997, when the agency invited anyone to submit an algorithm as the new standard, to be known as AES. As a condition for entry into the AES process, developers promised to give up any intellectual property rights to the selected algorithm. Many individuals and companies responded, and on August 20, 1998, NIST named 15 candidates.

The next step was for the world to analyze the algorithms. The criteria were security (no algorithmic weaknesses), performance (it had to be fast on many platforms), and size (it couldn't take up much space or use much memory). Many of the original 15 algorithms did not last long. Weaknesses were discovered, and some were shown to be simply too big or too slow.

In August 1999, NIST trimmed the list to five candidates. For the next year, researchers, cryptanalysts, and vendors of computer hardware and software tested the algorithms to decide which they liked best. Many papers were published, and volumes of statistics were released comparing the finalists. Each had its strengths and weaknesses.

Finally, on October 2, 2000, NIST announced the winner: an algorithm called Rijndael (commonly pronounced "Rhine-doll") invented by two Belgian researchers: Vincent Rijmen and Joan Daemen.

From now on, the AES algorithm is free for anyone to develop, use, or sell. As with DES, it is expected that AES will become a worldwide standard. You can expect that within a short time, if someone has cryptography, he or she has AES.

Summary

If you want to encrypt something, follow these steps.

1. Select a symmetric algorithm and a PRNG. You should choose an encryption scheme that is not susceptible to attacks on the algorithm. It should also allow key sizes big enough to thwart a brute-force attack. If you need to reuse your cryptographic keys, choose a block cipher. If you need to guarantee interoperability with other cryptographic programs or products, choose AES. Otherwise, you might want to choose a stream cipher for performance reasons.
2. Collect your seed value and feed it to the PRNG. Make sure that your seed contains enough entropy to thwart a brute-force attack. It's best to combine several seeds, including user input.
3. Using the PRNG, generate a key. Choose a key size that requires a brute-force attack that is so time-consuming that it is unfeasible. Currently, the most popular key size is 128 bits.
4. Apply the symmetric algorithm, which will work with the key to encrypt your plaintext.
5. Save and protect your key. The next chapter talks about how to protect keys.

To recover the data you encrypted, follow these steps.

1. Retrieve your key.
2. Apply the symmetric algorithm, which will work with the key to decrypt your plaintext.

Real-World Example: Oracle Databases

How do people and companies use symmetric-key cryptography today? Here is one example.

Most companies store volumes of sensitive information in databases. A database is a software package that stores data in a systematic way and enables users to easily and quickly find what they're looking for. For example, a company may have personnel files containing names, addresses, salaries, and Social Security numbers of all employees.

A hospital may keep medical records of hundreds of patients. An e-commerce company might store credit card numbers and customers' purchasing histories.

The owners of the databases may want to make sure that only the appropriate people have access to the information. One way to protect the data is to encrypt it. If attackers break into the database, they still can't read the sensitive material.

Oracle sells a database product, Oracle *8i*, release 8.1.6, that comes with an encryption package. If you are a developer using the database, and you want to encrypt the elements before storing them, you generate some random or pseudo-random bytes to be used as the key and then call on the package to perform the encryption. The calls to the encryption function are PL/SQL, which are standard database language conventions. For instance, to encrypt the data, you would add a line of code that looks something like this.

```
dbms_obfuscation_toolkit.DESEncrypt(input_string => plaintext,
                key => keyData, encrypted_string => ciphertext);
```

And that's it. Well, you also need to save the key somewhere (not in the same location). The next chapter talks about how to do that. If your application was using SQL, it would now have the opportunity to store the data in the clear (plaintext) or encrypted (ciphertext). This line shows that you are using DES, but Triple DES is also available. When your program needs to retrieve data, you recall it from the database, recover your key, and make something like the following call:

```
dbms_obfuscation_toolkit.DESDecrypt(input_string => ciphertext,
                key => keyData, decrypted_string => plaintext);
```

Thanks to Mary Ann Davidson and Kristy Browder of Oracle for providing this example.

CHAPTER 3

Symmetric-Key Management

Symmetric-key encryption can keep your secrets safe, but because you need your keys to recover encrypted data, you must also keep them safe. The process of keeping all your keys safe and available for use is known as key management. This chapter is about managing symmetric keys.

In Chapter 2, "Symmetric-Key Cryptography," Pao-Chi generated a random or pseudo-random key, and used it to encrypt data. If he wants to decrypt the data, he must use the same key. This means he has to either memorize the key or store it somewhere. Memorizing it isn't practical, so he must store it so that *he* can recall it when he wants to, but no one else can. Right now you're probably asking, "If there's some place Pao-Chi can keep his key safe, why doesn't he just put his sensitive information there as well?" The answer is that it's easier to protect a small key than many megabytes worth of information. In fact, some of the key storage solutions you'll see in this chapter are small devices designed in part to protect keys. So the idea is to use symmetric-key crypto to protect the megabytes of information and some other technique to protect the 16 bytes (or so) of keys.

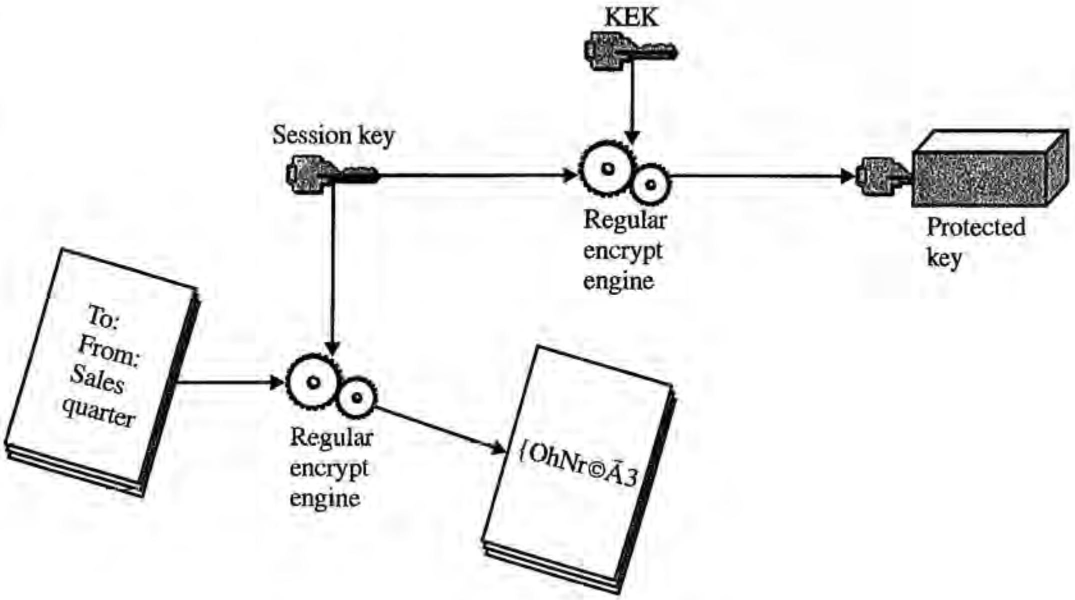
Password-Based Encryption

The key used to encrypt the megabytes of information, or bulk data, is generally known as the *session key*. A session is simply an instance of encryption, possibly during an email exchange, a World Wide Web connection, or a database storage. In Pao-Chi's case, a session involves encrypting a file before storing it on his hard drive. Some systems generate a new key for each session; others use the same key from session to session. One way to store the session key securely is to encrypt it using a symmetric-key algorithm. Someone who finds the session key has really found the encrypted key. The attacker would have to break the encryption to get the key that protects the megabytes of information. Of course, the process of encrypting the session key itself needs a key. That is, the key needs a key. There's the session key and then the *key encryption key*, as shown in Figure 3-1. In the crypto literature, not surprisingly, the latter is often known as the KEK.

You may be thinking that if Pao-Chi uses a KEK, he now has to store and protect it as well. Actually, he does not store the KEK, and therefore does not need to protect it. When he needs a KEK to encrypt, Pao-Chi will generate it, use it, and then throw it away. When he needs to decrypt the data, he generates the KEK again, uses it, and throws it away. He is able to generate the KEK a second time and produce the same value as before because it is based on a password. Pao-Chi uses an RNG or PRNG to gen-

Figure 3-1

A session key protects data, and a *key encryption key* (KEK) protects the session key



erate a session key, he uses *password-based encryption* (PBE) to build the KEK. It usually works like this (see Figure 3-2).

1. Enter the password.
2. Use an RNG or PRNG to generate a *salt*.

NOTE:

What's a salt? We describe the salt and its purpose in a few paragraphs.

---

3. Using a mixing algorithm, blend the salt and password together. In most cases, the mixing algorithm is a message digest. And that's the second time we've mentioned this tool—the message digest. The first time was in discussing PRNGs. Remember, a digest is a blender, taking recognizable data and mixing it up into an unrecognizable blob. We'll talk more about message digests in Chapter 5.
4. The result of step 3 is a bunch of bits that look random. Take as many of those bits as needed for the KEK and use it with a symmetric-key algorithm to encrypt the session key. When the session key has been encrypted, throw away the KEK and the password. Save the salt.
5. When storing the now encrypted session key, be sure to store the salt along with it. It is necessary to decrypt.

When it comes time to decrypt the data, here's the process.

1. Enter the password.
2. Collect the salt. The same salt used to encrypt is required (that's why you saved it with the encrypted session key).
3. Using the same mixing algorithm used to encrypt, blend the salt and password together. If one or more of the salt, password, or mixing algorithm is different, the result will be a KEK; however, it will be the wrong KEK. If all three elements are the same, the result is the correct KEK.
4. Use this KEK from step 3 along with the appropriate symmetric-key algorithm to decrypt the session key.

You probably have four questions.

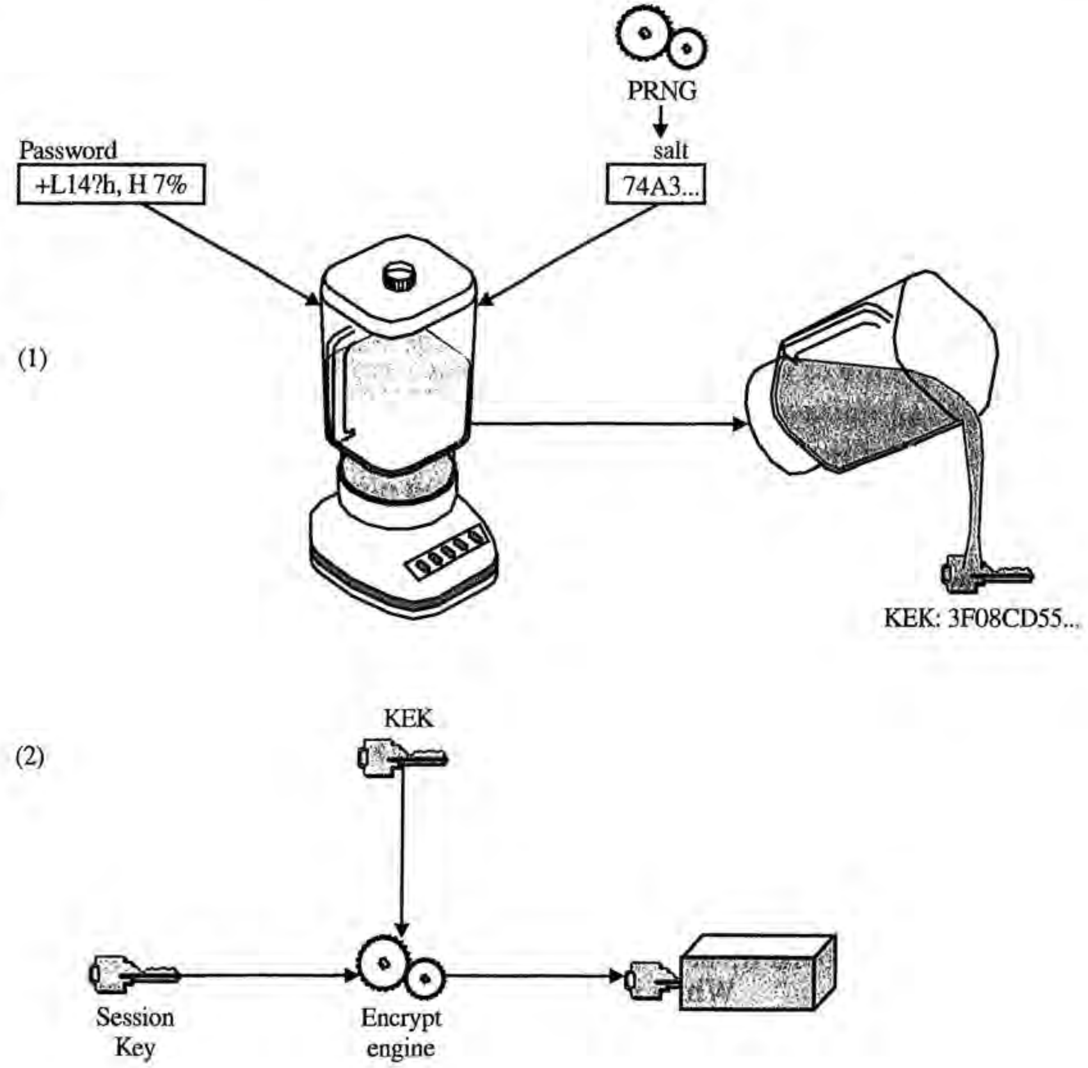


Figure 3-2

In *password-based encryption* (PBE), (1) blend the password and the salt to form a KEK and then (2) use it to encrypt the session key. To decrypt the data, use the same password and salt

Mixing Algorithms and KEK

Why use a mixing algorithm? Why not just use the password as the KEK?

A password does not have much entropy. Recall from Chapter 2 that entropy is the measure of randomness. But a password is made up entirely of keystrokes (characters associated with the keys on a keyboard), which are not sufficiently chaotic. Using a mixing algorithm on the password (and salt) ensures that the KEK looks random.

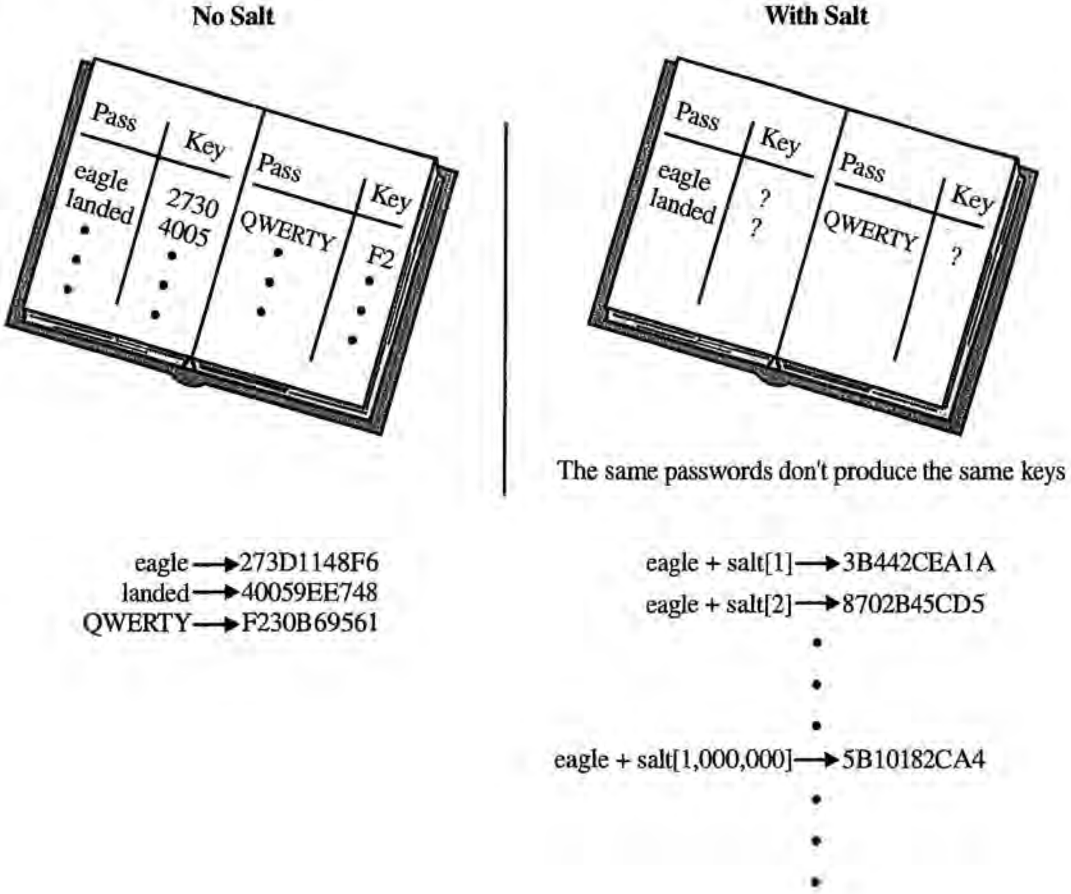
The Necessity of Salt

Why is a salt needed in the first place?

The salt is there to prevent precomputations. If the password were the only thing used to generate the KEK, an attacker could create a dictionary of common passwords and their associated keys. Then a brute force attack would not be necessary; the attacker would try only the precomputed keys (logically enough, this is called a *dictionary attack*). With a salt, the attacker must wait until seeing the salt before finding the KEK any particular password produces (see Figure 3-3).

Figure 3-3

Using a salt foils a dictionary attack



Storing Salt with Ciphertext

If the salt is stored with the ciphertext, then won't the attacker be able to see it? Wouldn't it be safer to keep the salt secret?

As just explained, a salt's only purpose is to prevent precomputations. That's worth repeating: the salt does not add security; it only prevents a dictionary attack. Even though the salt is not secret, it achieves that goal. Besides, if the salt is secret, how is it recovered when needed?

Reasons for Using Two Keys, a Session Key, and KEK

Wouldn't it be easier to simply use PBE to encrypt the bulk data? Why is it necessary to have two keys (the session key and the KEK)?

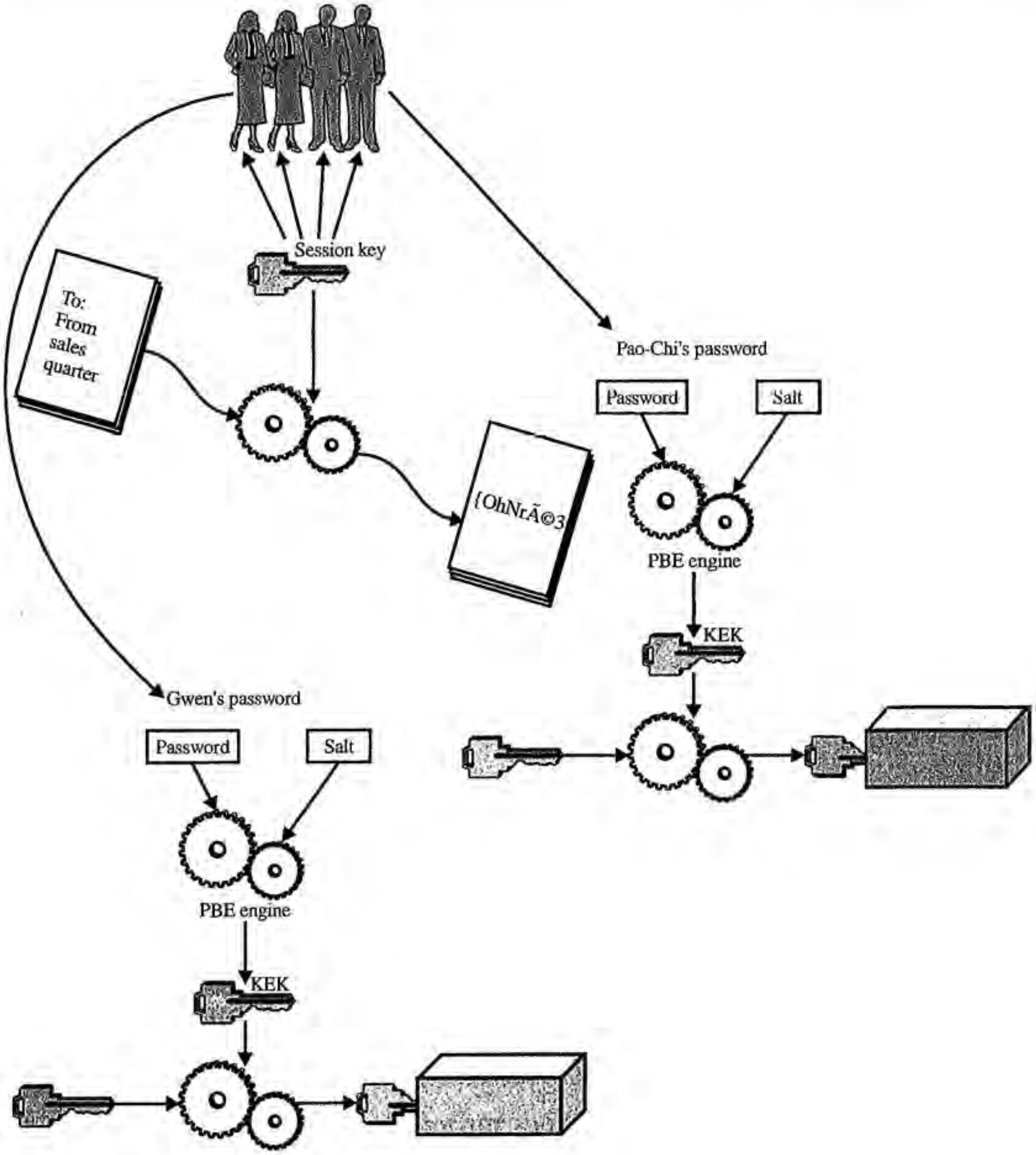
There are a couple of reasons to use a session key and a KEK. First, suppose you need to share the data with other people and you want to keep it stored encrypted. In that case, you generate one session key, and everyone gets a copy of it. Then everyone protects his or her copy of the session key using PBE. So rather than share a password (something everyone would need for decrypting if you had used PBE to encrypt the bulk data), you share the key (see Figure 3-4).

The second reason for using both keys is that it's easier to break a password than to break a key (more on this soon), and attackers might have easier access to the encrypted data than to the encrypted key. For instance, suppose Pao-Chi's data is on the network and the encrypted session key (the value encrypted by PBE using the KEK) is on his own personal computer (or other storage facility). Suppose Ray, an attacker, breaks into the network and steals the encrypted bulk data. To decrypt, Ray would have to break the session key or else perform a second break in (possibly into a more secure location) to find the encrypted session key and then break the password. Alternatively, if Pao-Chi used PBE to protect the data, Ray can recover the information by breaking the password (see Figure 3-5).

Of course, it is possible to use PBE to do the bulk encryption. In this book we don't discuss that option. From a programming point of view, it's not much more difficult to use a session key and then PBE to encrypt the session key, so you might as well because of the reasons given.

Figure 3-4

Using a session key for bulk data and protecting it with PBE means that users don't have to share passwords



Programming Convenience

A PBE program will do its work, even with the wrong password. Suppose the wrong password were entered, the program would have no way of knowing it was an incorrect password. It would simply mix the "bad" value with the salt and produce a KEK. It wouldn't be the correct KEK, but the

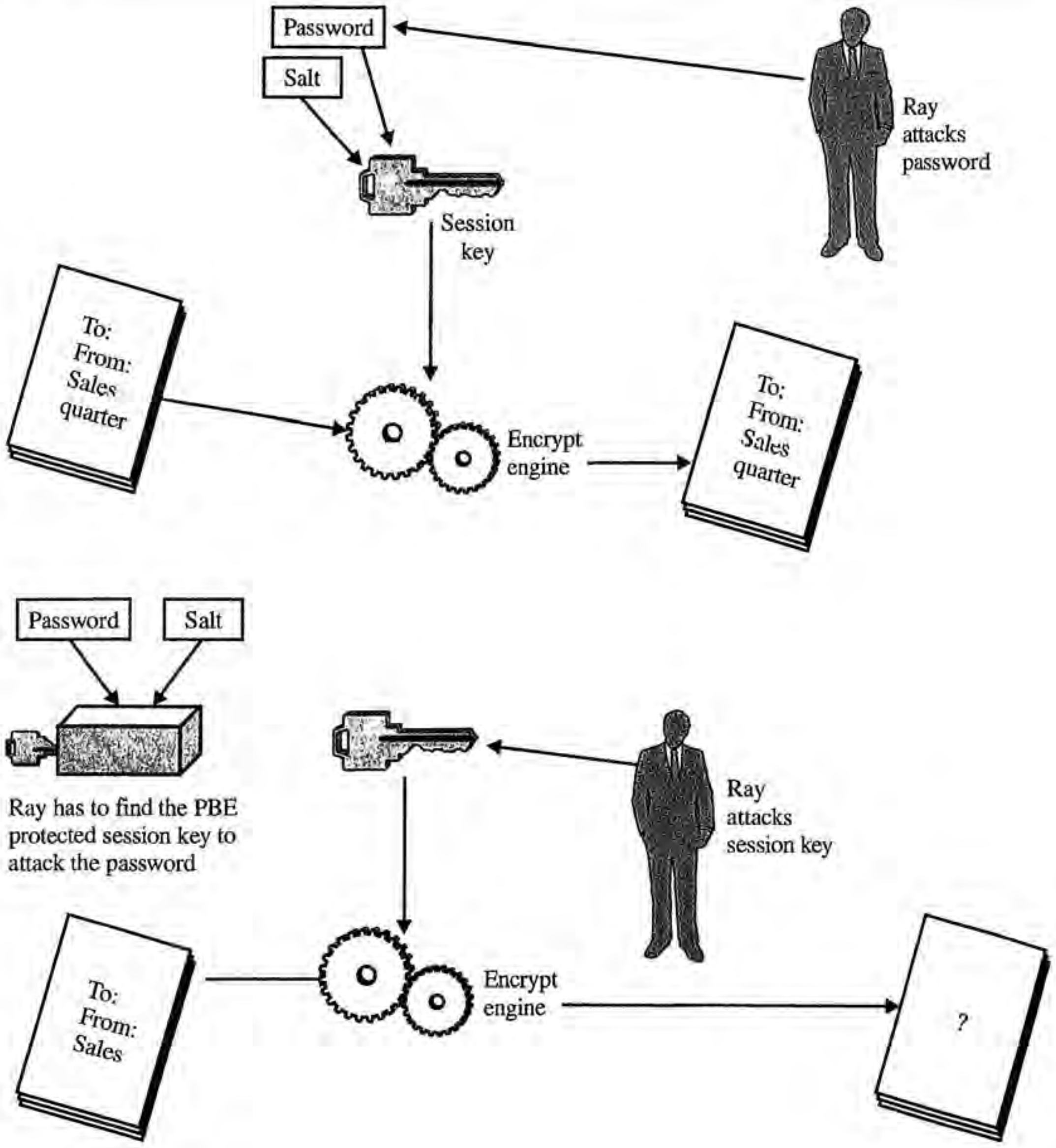


Figure 3-5

If Pao-Chi uses PBE to protect bulk data, Ray can recover it by breaking the password. If Pao-Chi uses PBE to protect the session key, Ray must find the encrypted key

program wouldn't know that; it just blindly follows instructions. It would then use that KEK to decrypt the session key. That would work; some value would come out as a result. It would be the wrong value, but there would be something there. Then the program would use this supposed session key to decrypt the ciphertext. The resulting data would be gibberish, but only then would it be possible to see that something went wrong.

For this reason, it would have been more convenient if, when entering the password, there were some way to know immediately whether it's the correct password or not. That would be better than decrypting the entire bulk data before finding that out.

One solution is to use the KEK to encrypt the session key along with something else, the "something else" being some recognizable value, such as the salt. Then when decrypting, the program checks this recognizable value first. If it's correct, continue using the session key to decrypt the bulk data. If not, the password was wrong and the process should start over.

The overall process looks like this. To encrypt bulk data:

1. Generate a random or pseudo-random session key. Use this key to encrypt the data.
2. Enter the password, generate a salt, and mix the two together to produce the KEK.
3. Encrypt the salt and session key using the KEK. Store the encrypted data with the salt.
4. Store the encrypted session key, which is actually the session key and the salt (see Figure 3-6).

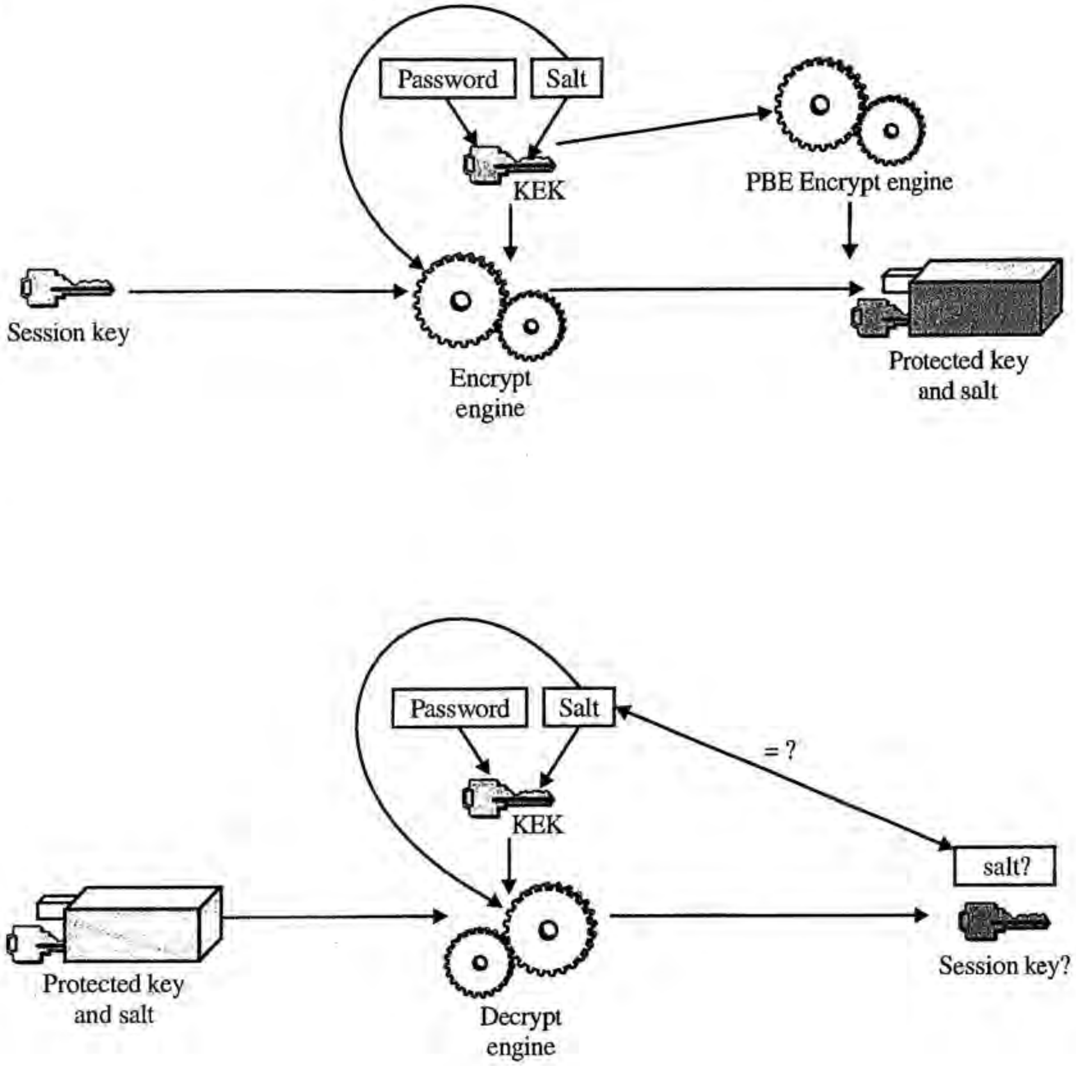
To decrypt the data, follow these steps.

1. Collect the salt and password and mix the two together to produce what is presumably the KEK.
2. Using this KEK, decrypt the session key. The result is really the session key and the salt.
3. Check the decrypted salt. Is it correct?
   a. If it is not correct, don't bother using the generated session key to decrypt the data; it's not the correct value. The user probably entered the wrong password. Go back to step 1.
   b. If it is correct, use the session key to decrypt the data.

Instead of the salt, you can use a number of things as a check. For example, it could be an eight-byte number, the first four bytes being a random value and the second four, that random value plus 1. When decrypting, check the first eight bytes; if the second four bytes is the first four plus 1, it's the correct password. This may be more palatable than the salt, since if the salt is the check, there is now some known plaintext. Presumably, the cipher is immune to a known-plaintext attack, but nonetheless,

Figure 3-6

Use a KEK to encrypt the session key along with a recognizable value such as the salt. Entering the wrong password produces the wrong KEK/salt combination



some people might feel it is more secure without any known plaintext. Of course, it is possible to use the wrong password and get a KEK that decrypts the check into a different eight-byte value that by sheer coincidence passes the test. The chances of this happening are so small, it will probably never happen in a million years.

Another check could be an algorithm identifier. This would be some sequence of bytes that represents the algorithm being used. Or it could be a combination of some of these values. In the real world, you'll probably find that engineers come up with complex procedures that include multiple checks. In these schemes, maybe one check accidentally passes, but not all of them.

Breaking PBE

Our attacker (who we're calling Ray) has two ways to break PBE. First, he could break it like any symmetric-key encryption and use brute-force on the KEK. Second, he could figure out what the password is.

Although the KEK is the result of mixing together the password and salt, Ray doesn't have to bother with those things; he could simply perform a brute-force attack on the KEK, use it to decrypt the session key, and then decrypt the data. This might be plausible if the session key is larger than the KEK. In Chapter 2, though, we saw that if a key is large enough, that's not going to happen. Hence, Ray will probably try the second way, which is to figure out what the password is. Once he has the password, he can reconstruct the key-generating process and have the KEK.

How can Ray figure out what the password is? One way would be to try every possible keystroke combination. This would be another flavor of the brute-force attack. If Pao-Chi entered the password from the keyboard, Ray could try every possible one-character password. Then he would try every two-character combination (AA, AB, AC, AD, . . .), then three-character values, and so on. In this way, eight-character or less passwords (on a keyboard with 96 possible values) would be approximately equivalent to a 52-bit key. Ten-character passwords are equivalent to about 65-bit keys.

Another attack is for Ray to build up a dictionary of likely passwords, such as every word in the English, German, French, and Spanish languages, along with common names, easy-to-type letter combinations, such as "qwertyuiop." He could add to that dictionary lists of common passwords that are available from hacker sites and bulletin boards (if you've thought of a password, someone else probably thought of it also). When confronted with PBE, he runs through the dictionary. For each entry, he mixes it with the salt and generates an alleged KEK. He tries that KEK on the chunk of PB-encrypted data. Did it produce the session key? Because the original PBE probably has a check in it (such as the salt encrypted along with the session key), it's probably easy to determine. If the check passes, that was the correct password and it produced the correct KEK, which in turn will properly decrypt the session key, which will then decrypt the bulk data.

This dictionary attack tries fewer passwords than does the brute force attack. Any password the dictionary attack tries, the brute force attack also tries, but the brute-force attack tries many additional passwords that the dictionary attack does not. As a result, the dictionary attack is faster than the brute force attack.

Of course, if Pao-Chi comes up with a password not in Ray's dictionary, it will never succeed. If Ray is smart, he'll probably start with a dictionary attack and if that fails, move on to a modified brute-force attack.

Slowing Down an Attack on a Password

To check a password, Ray has to mix the salt and password the same way Pao-Chi did. Pao-Chi can slow Ray down by making that a lengthy task. His goal will be to make the process quick enough that it doesn't make his own encryption or decryption process too expensive, but slow enough to be a drain on Ray. He can do this by repeating the mixing over and over.

First, mix the salt and password together. Then take the result of that and run it through the blender again. Then take the result of that and run it through the blender. And on and on, say 1,000 times.

The blender is probably pretty fast, the mixing is almost certainly done with a message digest, and these algorithms are generally very fast, so for Pao-Chi to do 1,000 iterations of the mixing process won't be too time-consuming. In fact entering a password is going to be far more time-consuming than 1,000 mixings. So relatively speaking, for Pao-Chi, the mixing takes up a very small portion of the total time. But Ray is going to have to do 1,000 mixings for every password he tries. That can add up.

Let's say Pao-Chi has an eight-character password. In an earlier section we said that an eight-character password is equivalent to a 52-bit key. But actually, Ray cannot try one password as quickly as one key. If he tries the brute-force attack on a key, here's the process (BFK stands for "brute-force on the key"):

BFK1 Get a candidate key.
BFK2 Do key setup (recall the key table from Chapter 2).
BFK3 Decrypt some ciphertext, yielding some purported plaintext.
BFK4 Check the plaintext.

But for each password Ray checks, on the other hand, here's the process (BFP stands for "brute-force on the password"):

BFP1 Get a candidate password.
BFP2 Perform the mixing to build the candidate key.
BFP3 Do key setup.

BFB4 Decrypt the ciphertext, yielding the purported check and session key.

BFB5 Perform the check.

How long it takes to do one BFK depends on four things. How long it takes to do one BFP depends on those same four things, plus one more. If step BFP2 is as long as the other four steps combined, that's going to double the amount of time to check one password. That's like adding one bit to your password. The eight-character password which was equivalent to a 52-bit key is now more like a 53-bit key.

In our experiments, performing 1,000 iterations (doing step BFP2 1,000 times) is about 136 times slower than the other steps combined (more or less, depending on the encryption algorithm; we used RC4, a very fast algorithm). On one Pentium-based PC, step BFP2 took 4.36 milliseconds, whereas checking one key took 0.032 milliseconds (a millisecond is "one one-thousandth" of a second; Pao-Chi is going to pay this 4 millisecond penalty when he encrypts or decrypts). Although Ray could check 31,000 keys per second, he could check only 230 passwords per second. The eight-character password is now equivalent to a 59-bit key. The 10-character password is more like a 72-bit key.

Incidentally, you may be thinking, "In a lot of places I've used passwords, there's a limit to how many times I can enter the wrong password before the program won't work. So if I try too many wrong passwords even if I later on do enter the correct password, the application won't run. Can't I just make PBE work the same?"

It's possible to write such a program, but the attacker will simply use a different PBE program that mimics the original. That is, Pao-Chi used his program to encrypt. Ray would simply obtain a copy of the ciphertext and run it through another program that looks like Pao-Chi's, except Ray's program puts no limits on the number of passwords allowed.

Good Passwords

In choosing a password, your goal is to choose one that doesn't appear in a dictionary and would thwart a brute-force attack. For example, the following password probably does not appear in a password dictionary:

14G:c*%3<wM*-l6g]_Bnp?~· d86

Editorial: The "Three-Try" Password Limit, A Pain in the Neck

by Steve Burnett

Many programs, especially login programs, place a limit on the number of wrong password tries they will accept before locking up. Usually, the limit is three. Enforcing a limit is a good security measure, but it's very annoying that the limit is so low. Furthermore, a low limit does not add any significant security compared to a larger limit.

Suppose you enter a password and the program denies access. You check and see that you accidentally have the CAPS LOCK on. You fix that and type in a password again. But this one didn't work either. What happened? Did you forget the password? Or did you simply misspell (for instance, how many times have I typed in "teh" for "the" or even "Bunrett" and that's my own name!)? Did you accidentally press a stray key? There's no way to know since you can't see what you typed. You've made two tries and gotten it wrong both times; are you going to try a third time? Probably not, because if you get it wrong, you'll be locked out. So it really isn't it a "three-try" password but a "two-try."

Now what about attackers? If the password is so weak that you need to limit intruders to no more than three tries, it's too weak. The security department should be talking to the employees about using better passwords. What's more, attackers may not even be trying the password through the user interface. Instead, they're probably grabbing information and trying the attack offline.

Given this, why not set the limit of password tries to, say, 10? That would make things easier for the user and most likely wouldn't give attackers any significant assistance. "Three tries and you're out" is just a pain in the neck.

It's a possible password, but attackers probably won't get around to trying it for a very long time. The problem with this password, of course, is that it's not easy to remember, and even if you could remember it (maybe you have a photographic memory), it's easy to mistype.

If you're using PBE, you need a good password. What makes a good password? The following list comes from an RSA Security manual. Other sources might offer other guidelines, but this is a good start.

1. Use at least 10 characters.
2. Mix in uppercase and lowercase letters, numbers, spaces, punctuation, and other symbols.
3. Avoid using a character more than twice.
4. Avoid using actual words.
5. Avoid using personal information, such as the name of a spouse, child, parent, or friend, or your phone number, Social Security number, license plate number, or birthday.
6. Do not write it down. Instead, memorize it.

Number 6 is the hardest if you follow recommendations 1 through 5. In addition, if you have several applications, security experts recommend that you use a different password for each one. What's more, some applications enforce a policy that requires you to change your password periodically.

Given all that, what's the average user to do? So far, there are no easy answers to the password dilemma. Later sections describe some alternatives to passwords, along with ways to use passwords more effectively. Unfortunately, these techniques require new hardware, and for some of them the technology is years away from perfection or public acceptance.

Password Generators

Programs are available that will generate passwords for you. These programs work like PRNGs but produce keystrokes instead of numbers. For example, the program may collect some seed bytes, including your mouse movement and keystrokes. Then it spits out a password that probably looks random. Most programs allow you to specify how long the password will be, whether the password combines uppercase and lowercase letters, or whether it should contain punctuation or other marks. You might get results like this:

tiFXFCZcZ6

K6($xV]!h1

M?a84z9W,g

Technical Note: You Never Know Where Attackers Will Look Next

Do you think that you can choose a key or password that will force a brute force attack to run to completion? For example, if the brute force attack on the password begins with *A,* then *B,* and then so on through the alphabet to *AA, AB,* and so on, you might think it would be clever to choose ZZZZZZZZZZZZ as your password. After all, that's a long way away from the beginning of the list.

Unfortunately, brute force attacks usually don't work that way. First, most brute force attacks use more than one computer, and each computer works with some of the possible key or password space.

Here's how it works. A computer that wants to be part of the cracking process applies to a central "bureaucrat" computer. This central computer keeps track of the keys or passwords that have been searched. It generates a range of keys or passwords for the "worker" computer to check, which then searches all the values in that range. If the worker computer finds the key or password, it reports the good news to the bureaucrat. But if the worker searches its entire allotted range with no success, it goes back to the bureaucrat to get another range.

How is a range determined? Probably not systematically. In other words, the first range is not going to be *A* to *ZZZ,* the second range from *AAAA* to *ZZZZ,* and so on. Instead, the ranges are probably parceled out randomly. The first applicant gets something like EV9A3LGP to FBMA111G, the second applicant gets W6MWC0O to ARH7ZD2F, and so on.

Even if only one computer is involved in the brute force attack, it operates as both a bureaucrat and a worker. As a result, you never know which part of the space will be searched next.

These passwords were generated using the JavaScript Source password generator (see http://javascript.internet.com/).

They are good passwords, but they're harder to memorize. Still, if you want a "random" password, one that will withstand a dictionary attack, a program such as this one might be a good choice.

Make sure that you trust the program you choose. Imagine a malicious password generator programmer. Suppose our attacker Ray creates a program that produces what looks like random passwords. But actually the program is limited to how many it can really create, say 10 million. Now Ray simply looks at who buys the product, and then has a leg up on cracking that customer's passwords.

Hardware-Based Key Storage

We've just examined PBE as a possible way to store cryptographic keys. Another storage place is on a hardware device. Some devices are tiny computers called *tokens*. Others are larger, tamperproof boxes, generally called *crypto accelerators*.

Tokens

A token is not a cell phone or a *personal digital assistant* (PDA) such as Palm, iPaq, and so on, but rather is something even smaller that fits inside your wallet or shirt pocket: a plastic "smart" card, a plastic "key," a small USB port attachment, or even a ring you wear on your finger. (Smart cards and USB port attachments, the most common types of tokens, are discussed in the following two sections.) A token contains a small chip with a processor, an operating system of sorts, and limited input/output, memory, and hard drive storage space. Some tokens are very small or thin, are slow, have very little storage space, and do very little. Others may have more power and can store as much information as a 1970s era PC. Figure 3-7 shows some tokens.

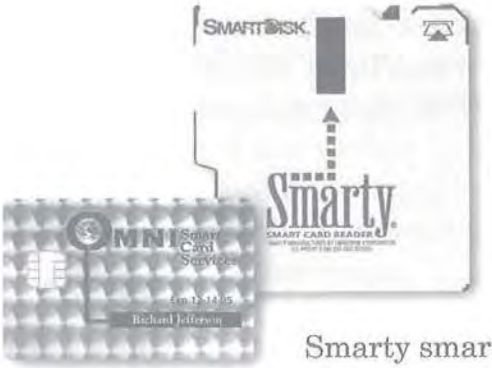
Figure 3-7

Some tokens

Java ring

iKey 2000

Datakey



Smarty smart card and reader

RSA SecurID 3100 smart card

The advantage of using tokens is that the attacker does not have access to them. If our attacker Ray is in Elbonia, he can probably use the internet to access Pao-Chi's computers' hard drives and does not need to be in his office to break in. (As you may know, Elbonia is a fictional country featured in the *Dilbert* comic strip by Scott Adams.) But Pao-Chi's token is not connected to the network (it's in his wallet or on his key chain or finger), so it's not visible. This arrangement thwarts a remote attack. When Pao-Chi *uses* his token, it's connected to his computer, which is ultimately connected to the world, so for a brief while, his secrets are vulnerable. But a few seconds of vulnerability is not as dangerous as the 24 hours a day the network is vulnerable.

Even if Ray obtains Pao-Chi's token, further protections are built-in. Generally, a token performs functions (such as retrieving stored keys) only when a correct password or *personal identification number* (PIN) activates it. Often, a token locks itself if too many incorrect passwords are entered. If someone tries to physically get at the storage space (as in Chapter 1 with data recovery techniques), the token will erase itself—sort of a "scorched earth" policy. This scorched earth thwarts an offline attack on the password.

The problem with tokens is that they need a way to communicate with the computer; once they can communicate with the computer, they can communicate with users through the computer. For example, you communicate with the computer by using the keyboard and mouse. Sound systems communicate using a sound card. A token might use the serial or USB port, or even the floppy drive. Some tokens use a *reader* to one of the ports. It's the reader that communicates with the computer. To use the token, you insert it into the reader, something that's generally easier than inserting it into a port. Of course, this means that you must buy the reader as well as the token and then install it.

Smart Cards

A *smart card* is simply a plastic card, similar to a credit card, that contains a microprocessor. One of the goals of smart card vendors is to replace the current version of the credit card. Just as credit cards with magnetic strips replaced simpler embossed cards, the hope is that smart cards will replace credit cards. But because smart cards contain small computers, they will be able to do more than serve as credit cards.

We'll talk more about smart cards throughout this book, but for now, one of the things you can do with them is to store keys. When you need to use a symmetric key, for example, you transfer it to the computer, which uses it to encrypt or decrypt data. To transfer the key between card and computer, though, you need a smart card reader. Several PC manufacturers have announced that future laptops and keyboards will come with built-in smart card readers.

The *International Organization for Standardization* (ISO) has published several standards outlining the physical characteristics of smart cards, including the dimensions and locations of the contacts, signals and transmission, and more. Virtually all smart cards look alike because they are built to standard. The idea is that all smart cards will be usable with a wide variety of readers. So far, however, many smart cards and readers simply don't work together. Often, to use a particular manufacturer's smart card, you must use that firm's reader. As more PC manufacturers release products with readers built in, this situation should change.

USB Tokens

The Universal Serial Bus port is an industry standard for attaching plug and play devices. Other ports have such functionality (such as PCMCIA), but the USB port is probably the most popular. Since 1998 or 1999, most new PCs and laptops have come with USB ports as standard equipment. If you have a device that connects to your computer through the USB port (such as a camera downloading pictures or a printer), there's no need to attach and reboot. So long as the software to run the device is installed, you simply insert the device and run it. When you're done with one USB device, take it out and insert a new one, or most likely, you can have several attached to the same port.

Several companies have introduced cryptographic tokens that attach to the USB port. Other companies with tokens that are not USB-ready have made adapters to USB ports. These tokens are approximately 2½ by ½ inches in size (about the size of a house key but a little thicker). They have quite a bit more computing power and storage space than smart cards. Hence, they will almost always be much faster, do more work, and store more keys than a smart card.

Tokens as Password Storage Devices

In addition to your keys, tokens can hold passwords. Suppose you have several places to log in: your network account, e-mail, various computer accounts, electronic commerce accounts (such as an account with an online travel agent or bookstore), and so on. For each account, you'd like a different password. In that way, if someone figures out one password (for example, the online travel agent might know your password for that account), he or she won't have them all.

The solution is to use a token to generate big random passwords and store those passwords. When you need to log in to an account, you hook up the token and have it send the password. You don't have to remember the password, so it can be random and very long, perhaps 20 or 30 characters.

You probably have access to the token through a password, so if attackers obtain your token and figure out that password, they've got all your passwords. That is a danger, but using a token does help thwart a remote attack. For example, suppose Ray, the attacker, goes to your online bank account and logs in as you. Although he need not be at your computer to do this—he can be in Elbonia—he does need to enter your password. A long, random password is much more difficult to crack than passwords you might otherwise use for your various accounts because they're easier to remember.

Crypto Accelerators

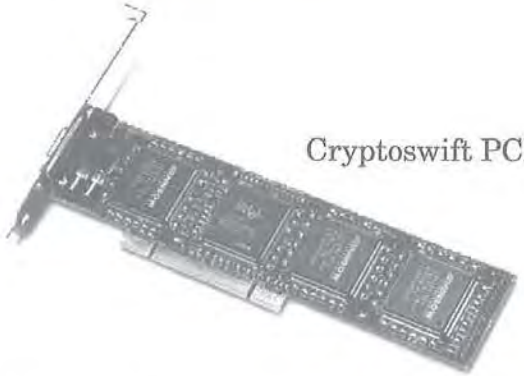
The larger hardware crypto devices are generally called *crypto accelerators* (see Figure 3-8) because they usually have specialized chips that perform cryptographic operations faster than general-purpose microprocessors. Crypto accelerators can also store data more securely than can a regular computer. The problem with, for example, your desktop PC is that the hard drive is visible to the outside world. As you saw in Chapter 1, attackers can probably read your computer's hard drive, and even if you have firewalls around your sensitive information, attackers can use tools, such as data recovery software, to read that data as well. But a crypto accelerator is built so that its storage space is not visible. There is very limited access to it using normal channels, and if attackers try to pry open the cover to physically access the hard drive, the device erases itself. If you store your key on such a box, it's extremely unlikely that someone will be able to extract it.

Many crypto accelerators do not let the key leave the device. With a token, if you want to encrypt 10 megabytes (MB) of data, you must get the key from the token and let your PC do the encrypting. While the key is in memory—and afterward, as you saw in Chapter 1 with memory reconstruction attacks—it is vulnerable. With a crypto accelerator, you send the

Figure 3-8

Some crypto accelerators

nShield key management and acceleration



Cryptoswift PCI E-Commerce Accelerator

Luna CA^3

AXL 300

plaintext to the device, and it encrypts and returns the ciphertext. This arrangement further limits the key's vulnerability.

One problem with crypto accelerators is that they are connected to your computer 24 hours a day. This is in contrast to tokens, which are connected only for a few seconds at a time, limiting their vulnerability. Presumably, the crypto accelerator I/O is secure so that if attackers have remote access to your computer, they still cannot get access to the accelerator. "Presumably," however, may not be adequate security in some situations. That's why most crypto accelerators work in conjunction with tokens—that is, they don't operate without a token inserted.

If you store your keys on the box, you can recover them by presenting the correct token and entering the correct password. For attackers to access your keys, they must somehow obtain your token (another token by the same manufacturer won't work, just as two credit cards don't refer to the same account) and the ability to use that token (usually a password). And, of course, they must have physical contact with the accelerator (to insert the token), again thwarting a remote attack.

Hardware Devices and Random Numbers

Tokens and crypto accelerators usually come with an RNG (see Chapter 2 for details about RNGs and PRNGs). You must be careful, though, because some tokens don't have true RNGs. Rather, they have PRNGs seeded at the factory. Even if your device constantly collects seed material each time it is used—a better approach than a PRNG seeded at the factory—it's still a PRNG.

Biometrics

A hardware device stores your keys securely, but it usually relinquishes them when someone enters a password. Good passwords can be strong, but in real life, not everyone uses good passwords.

Another way to authorize a device to unleash the key is through *biometrics,* which uses your unique physical characteristic to verify your identity. The most well-known biometric is the fingerprint. It's common knowledge that everyone, even an identical twin, has unique fingerprints. If a machine could read fingerprints, it could determine whether the appropriate person is requesting an operation. Such machines exist. (It's macabre, but some of these machines can even tell whether the finger being used is actually attached to the body and whether the body is alive.)

Other biometrics include retina scans, voiceprints, and even DNA. Biometrics companies are attempting to build hardware that can be programmed to identify you by scanning your eye, voice, or DNA and then appropriately release secure information or perform a cryptographic function.

Biometric devices are not currently in widespread use for a couple of reasons. One is the cost of the devices, and the other is their reliability. A number of concerns have been raised. Will the device return an erroneous "positive ID" on someone who isn't the identified subject? Will it always return a positive ID on the subject? What if the subject has cut his or her right thumb—will the fingerprint reader still function? Can it instead use the left thumb? Another finger? For a voiceprint reader, what if the person has a cold—will it still work? And so on. A password works virtually 100 percent of the time. If you enter the wrong password, access is denied. With the correct password, you always get access. With biometrics, there may be some errors.

The technology is advancing, and companies are building better and cheaper readers. Someday, maybe a smart card will contain not only a chip but also a fingerprint reader. Maybe your cell phone will have built-in voice recognition.

Summary

After you've generated a symmetric key and used it to encrypt data, how do you protect the key? One of the most common techniques is password-based encryption. In PBE, you use a password and a salt to build the key encryption key. You then use the KEK to encrypt the session key. Another method of protecting your session key is to store it on a hardware device, such as a token or crypto accelerator.

Real-World Examples

How do companies protect keys in the real world? One class of products for protecting session keys is file encryption applications. These products encrypt the files on your hard drive using symmetric-key cryptography. Protecting bulk data keys can be done in several ways.

Keon Desktop

RSA Security makes a family of products called Keon. One component is Keon Desktop. Among the features of this product is file encryption. You register directories with Keon, and it will encrypt all files in those directories (see Figures 3-9 and 3-10). When you open one of those files, Keon will decrypt it. When you close it, Keon will encrypt it again. That means if the file is on your hard drive, it is encrypted. It is decrypted only when you want to see it.

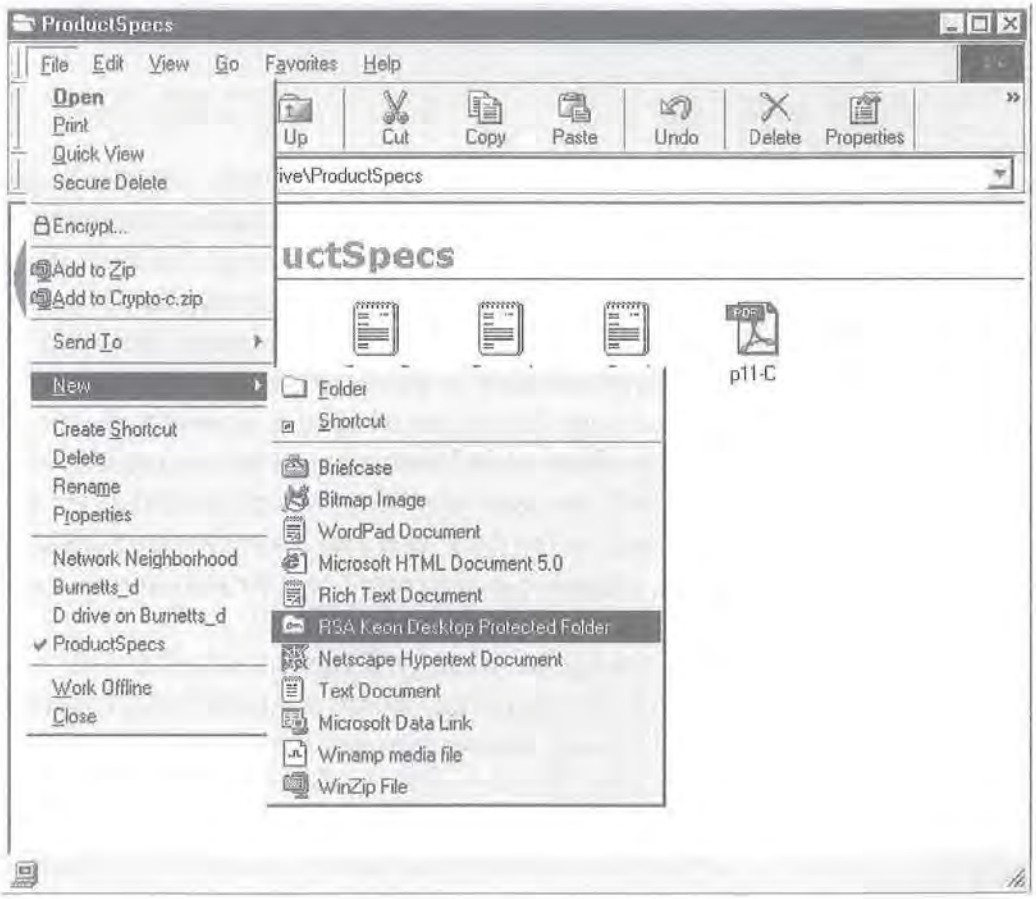


Figure 3-9

Registering a directory with Keon. Once registered, all files in this directory will be automatically encrypted when not in use, and decrypted when accessed

Figure 3-10

After creating a protected directory, choose the algorithm you want to use to encrypt the files. The key menu is for choosing where to store the session key, on a smart card or a virtual smart card

Current Folder: [ProductSpecs]

Name: New Protected Folder

Encryption

Key: [.P458028]

Algorithm: RC5-128

Please note that you will have to use the shift-delete operation if you want to delete encrypted folders using Windows NT.

Keon uses RC5 at 128 bits or DES at 56 bits to encrypt. It uses a PRNG to generate the key. The seed is various machine states and user input. Once the key has been used to encrypt the files, it's necessary to store that key. Keon stores it in the user's Credential Store. If the user has a smart card, Keon will use it as the Credential Store. If not, Keon will create a virtual smart card on the user's hard drive or on a floppy disk or both. The keys on this virtual smart card are protected using PBE.

If you keep your Credential Store on a mobile medium (the smart card or floppy), you can use Keon to encrypt or decrypt files from any computer you work on (as long as it has Keon Desktop installed), whether it is your office computer, home computer (for telecommuting), or a laptop on a business trip.

To read your encrypted file, an attacker will have to either break the encryption algorithm, create a substitute Credential Store (which would entail finding the session key through a brute-force attack) or break your Credential Store to obtain the bulk data key. The first two are highly unlikely, so an attack, if it occurs, will probably be mounted against your Credential Store. If you keep it on a smart card or floppy, the attacker will have to steal it. And then it will still be necessary to either break the smart card or break your password.

Other Products

If you search the Web, you will find dozens or even hundreds of applications out there that offer file encryption. Some are freeware, others are shareware, and some are regular products.

One of the most commonly used file encryption programs is PGP. The letters stand for *Pretty Good Privacy.* PGP was originally a freeware program written by Phil Zimmerman using RSAREF, the cryptographic reference library produced by RSA Data Security. According to the documentation, it has file encryption through PBE (it does not generate a key and protect the key with PBE; it encrypts the file using PBE). It also offers an advanced "enveloping" file encryption that uses a key on your "key ring." Once again, your key ring can be a number of devices, including a PBE-protected file.

CHAPTER 4

The Key Distribution Problem and Public-Key Cryptography

Symmetric-key encryption can keep your secrets safe, but if you need to share secret information with other people, you must also share the keys. How can you securely send keys to other individuals? This chapter describes some solutions, including the revolutionary concept of public-key cryptography.

Chapters 2 and 3 describe how Pao-Chi (the sales rep on the road) can keep secrets by encrypting his data and then safely storing the encrypting key. But suppose he wants to share some of his secrets with other people? For example, let's say Pao-Chi has just met with Satomi, a potential customer, and wants to discuss strategy with Gwen, the VP of sales and Pao-Chi's boss. Normally, Pao-Chi and Gwen could handle the conversation by phone, but they need to send complex documents back and forth, and they figure the best way to do that is through e-mail. Being a little paranoid, they want to ensure the security of this exchange of sensitive data. After all, Pao-Chi will likely be hooking up his laptop to Satomi's phone lines or Internet connection, and who knows what sort of sniffers are attached to her company's wires?

The simple solution is for Pao-Chi to encrypt any files he sends to Gwen. In that way, if Satomi intercepts the message, all she sees is gibberish. The problem is that when the message gets to Gwen, she also sees only gibberish. To decrypt the message, Gwen needs the key. Pao-Chi has the key, but how can he send it to Gwen? He can't send it in another message; if Satomi can intercept the data message, she can also intercept the key message. And if Pao-Chi could find a channel to send the key securely, he could simply send the secret via that route.

The problem facing Pao-Chi and Gwen is known as the *key distribution problem*—namely, how can two or more people securely send keys over unsecure lines? In more general terms, how can people securely send any sensitive information over unsecure lines? Because we can encrypt the data, though, we can reduce the general problem to the smaller problem of securely sending the key. If you have 10MB of sensitive material, you could try to figure out a way to send that bulk data securely, or you could encrypt it using a 128-bit symmetric key and then try to come up with a way to securely send the key. If you solve the key distribution problem, you also solve the bulk data distribution problem (Figure 4-1).

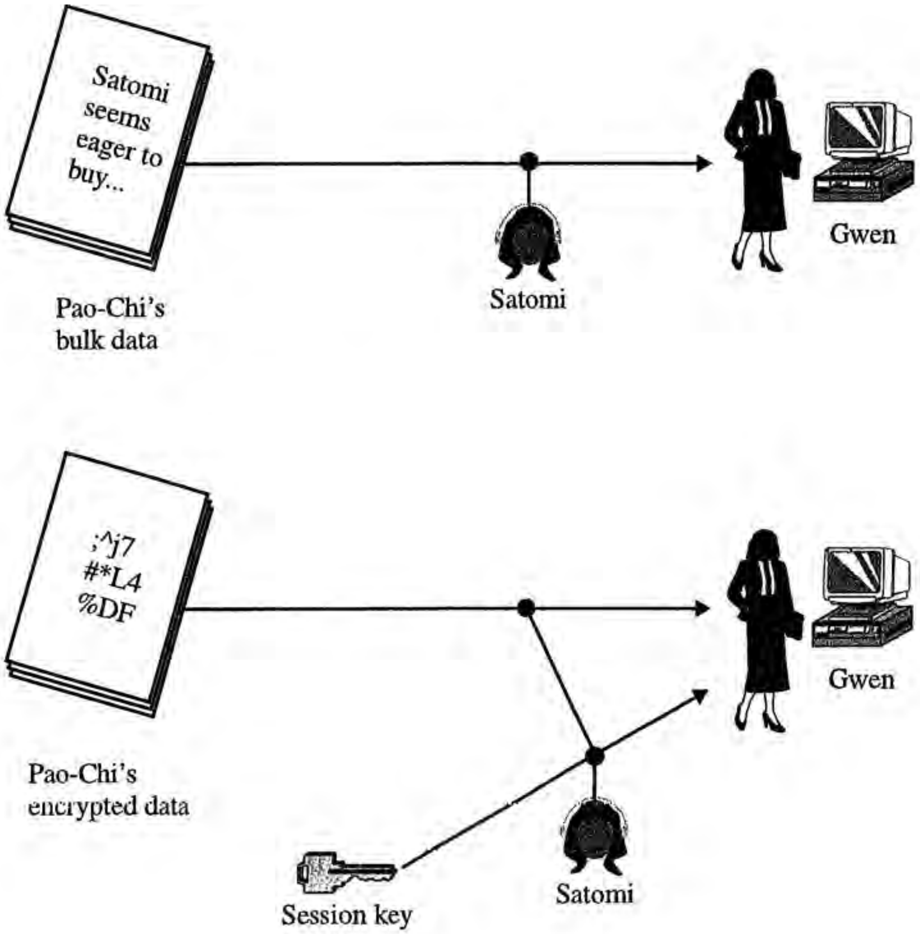


Figure 4-1

The key distribution problem: How can Pao-Chi send Gwen sensitive information, when Satomi might be eavesdropping?

Sharing Keys in Advance

In Chapter 3, you saw how Pao-Chi can encrypt bulk data with a session key and then store that key securely. He can store that key using, for example, PBE or a token. To solve the key distribution problem, Pao-Chi and Gwen can get together in advance to generate a key, and then each of them can store the key. To send secure messages to each other, they use the key to encrypt the data.

So before Pao-Chi leaves on his trip, he stops by Gwen's office with his laptop. He generates a 128-bit key and stores it somehow—maybe using PBE, maybe on a token. He then puts a copy of the key onto a floppy disk and hands Gwen the disk. She inserts the disk into her computer, copies the key, and stores it securely. Now the two parties share a key that they can use whenever they want to send sensitive material. This key, by the way, likely will not be the same key Pao-Chi uses to encrypt the files on his hard drive. If it were, Gwen could read all his sensitive data. If that's not OK with Pao-Chi, he has the option of encrypting his data using a key only he can access.

If the two of them had chosen to exchange the key online, Pao-Chi would not have had to go to Gwen's office in person. But their goal is to send no sensitive data in the clear over unsecure lines, and that includes the company network. Even if the network is secure from outsiders, that doesn't eliminate the possibility of an inside job. Another employee—maybe the system administrator or simply someone who is adept at hacking—might be able to intercept such a key exchange. So the safest way to exchange the encrypting key in advance is to do so in person.

Another possibility is for Pao-Chi to generate the key, encrypt it using PBE, and send the encrypted key to Gwen. Anyone intercepting the message would not be able to decrypt it without the password. Of course, Gwen needs the password, so Pao-Chi can give it to her by phone. In this way, the sensitive data (the password) is never sent over the network lines. But is the phone line secure? Maybe, maybe not. Still, whoever wants to steal the key will have to break into both the network and the phone system. Although this makes the attacker's job more difficult, it still means sending sensitive data over unsecure lines.

Problems With This Scheme

Pao-Chi and Gwen now share a key. This scheme will work; if attackers try to intercept their messages encrypted using that key, the attackers will not be able to recover the information. But this solution does have its problems.

Suppose the parties want to share keys with more than one person. Pao-Chi is not the company's only sales rep, and he may want to securely send information to his sales colleagues as well as people in the engineering, accounting, and shipping departments. To communicate securely with all these people, Pao-Chi will have to visit their offices and perform the key exchange. What's more, Gwen will have to make similar visits (or her colleagues will have to visit her—after all, she is the VP). Everyone will have to exchange keys in person with everyone with whom they share confidential information.

The logistics quickly become burdensome. Some colleagues may have offices in other parts of the country or even in other countries. The company can't send everyone on all the trips required to exchange keys. Maybe the solution would be to gather all the employees at one location and have a giant key exchange party. But what happens when the company hires someone new? Does it have yet another key exchange party? Send the new employee on a worldwide trip to exchange keys?

Furthermore, as more people need to share keys, the number of required meetings grows dramatically. When two people share a key, there's one meeting. When three people share keys, there are two meetings; with four people, six meetings, and so on. In general, n people, must make $1/2(n^2 - n)$ key exchanges. If your company has 10 employees involved in secure data sharing, that's $1/2(100 - 10)$ key exchanges, or $1/2(90) = 45$. For 20 employees, it's 190 meetings. A company with 1,000 employees would need to perform 499,500 key exchanges.

One solution is for everyone in the company to share the same key. You could have a "key master" who gives the key to all employees. The drawback is what happens when someone leaves the company. If the company does not change the key, an unauthorized individual can now decrypt sensitive materials. If, on the other hand, the company changes keys, the key master will have to revisit everyone in the company.

A second problem with the shared secret key is that if attackers crack one message, they crack them all. Because all messages between two people are encrypted with the same key, finding the key for one message means finding the key for all messages. It's not likely that attackers will find the key if the correspondents use a 128-bit key and an algorithm with no weaknesses. On the other hand, if it is possible to easily use a separate key for each message, why not take that extra measure of precaution? Although this is a drawback of the shared key approach, it's trivial compared with the pitfalls of trying to exchange keys in person.

Using a Trusted Third Party

If sharing keys in advance is not an option, Pao-Chi and Gwen can try using a *trusted third party* (TTP). This is a variation on the key master solution. In this scheme, the trusted third party—let's call her Michelle—shares a key with each individual in the company. Actually, the keys are *key-encrypting keys*, or KEKs. Pao-Chi visits Michelle and asks for a KEK. She generates one, stores it securely, and gives a copy to Pao-Chi. The two of them now share a KEK. Gwen also visits Michelle, and the two of them share a different KEK (see Figure 4-2).

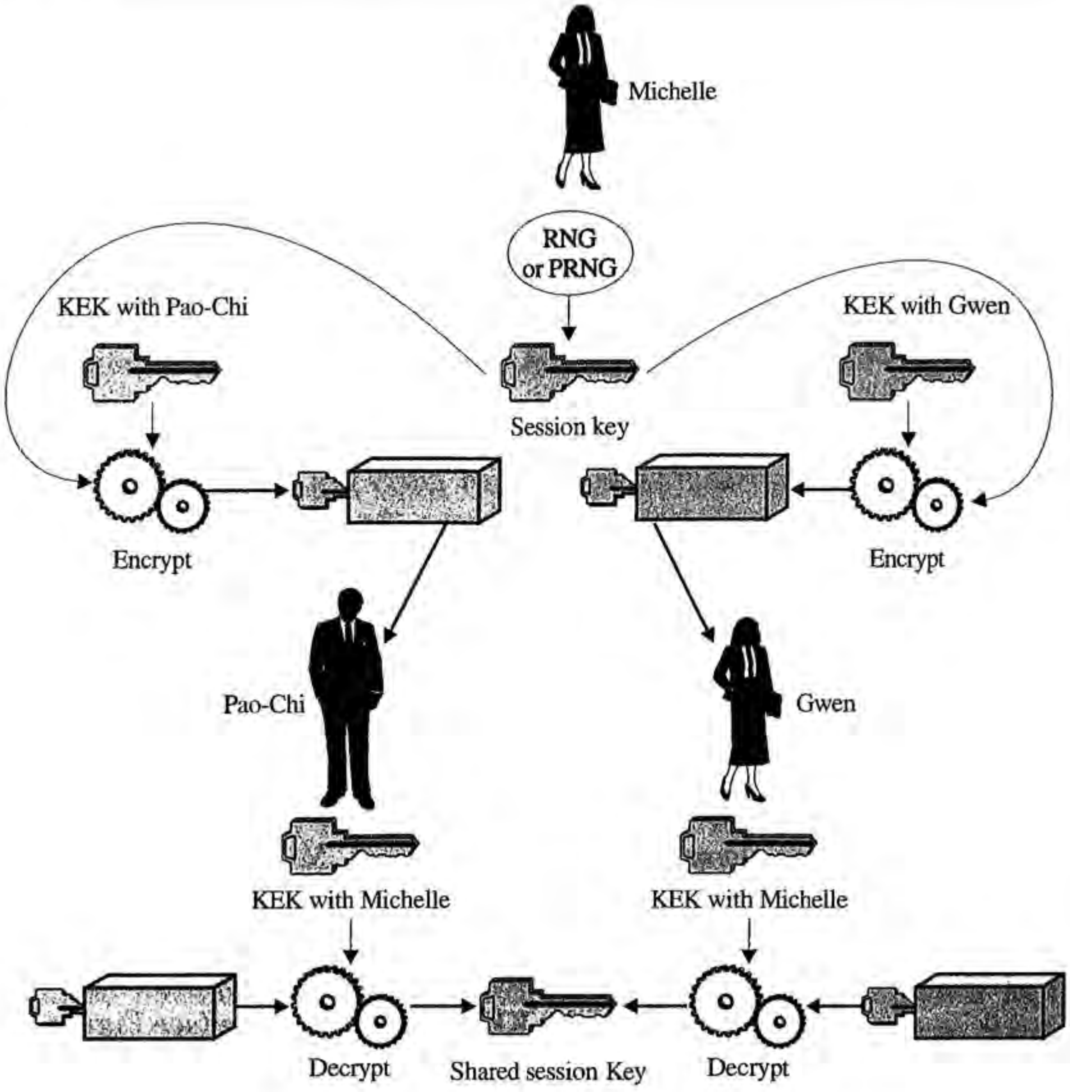
When Pao-Chi wants to communicate with Gwen, he sends a message to Michelle, requesting a session key he can use in his messages with Gwen. To fulfill the request, Michelle generates a new session key and sends it to Pao-Chi. She encrypts the new session key using the KEK she shares with him, so anyone intercepting that message cannot identify this new key. Michelle also sends this same new session key to Gwen, encrypting it using the KEK those two share (see Figure 4-2).

Pao-Chi and Gwen now share a key, and neither had to make a trip to the other's office. Anyone else wanting to share a key with any other employee simply establishes a KEK with Michelle, who distributes the key. In a trusted third party scheme, the correspondents are the first two parties. In our example, Michelle is the third party. Just as important, Michelle must be trusted because she has everyone's keys. When Pao-Chi and Gwen exchange encrypted messages, normally they are the only people who can decrypt them. But now Michelle also has their session key, so she can decrypt their messages. Pao-Chi and Gwen must trust Michelle not to read their sensitive material or release their key to anyone else.

The trusted third party still has to exchange keys with all the employees in person. As you saw in the preceding section, that's a daunting task. To make things easier, you can create a hierarchy of trusted third parties. Everyone goes to a local TTP, each of whom has established a key with every other TTP. For all the TTPs to exchange keys is still a formidable

Figure 4-2

Michelle acts as a trusted third party, distributing keys between Pao-Chi and Gwen



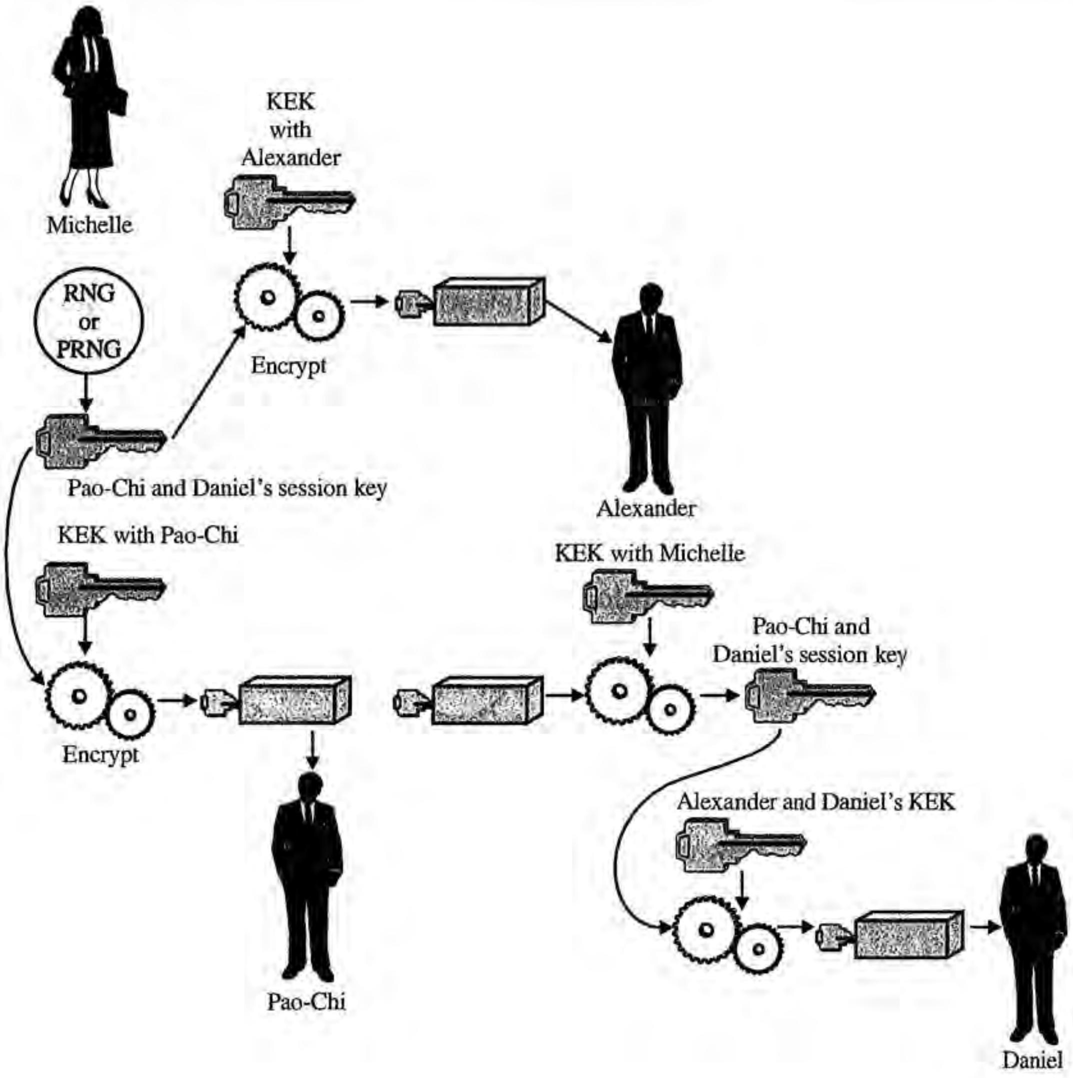
project, but it is more manageable than having a single companywide TTP. If two correspondents are in the same office, they can use the services of a shared TTP. If they are in separate offices, each one communicates with his or her own TTP. Then the two TTPs communicate with each other to bridge the gap (see Figure 4-3).

Problems With This Scheme

The first problem is that the TTP can read all the messages. The whole idea of encrypting messages is to limit their exposure to only the corre-

Figure 4-3

TTP Michelle (San Francisco) shares keys with TTP Alexander (New York), creating a hierarchy that serves Pao-Chi and Daniel in the two cities



spondents. Now a third person has access. If the correspondents can live with that, this scheme will work. Otherwise, they'd better look for another solution.

The second problem is that when the TTP leaves the company it must hire a new TTP and start the process over from the ground up. Otherwise, the outgoing TTP can gain access to all sensitive materials.

An alternative is to contract the job of TTP to an outside company. In this arrangement, the TTP is not an individual but a corporate entity. In this case, you must trust that the company has checks in place that prevent its employees from gaining access to the keys.

Public-Key Cryptography and the Digital Envelope

In the 1970s, researchers invented *asymmetric-key cryptography,* a new way to securely send keys. This scheme uses two different keys. Although they are related to each other—they are partners—they are significantly different. The relationship is mathematical, and what one key encrypts the other key decrypts. In symmetric crypto, the same key is used to encrypt and decrypt (hence the word "symmetric"—the same on both sides); if you use any other key to decrypt, the result is gibberish. But with asymmetric crypto (see Figure 4-4), the key that's used to encrypt the data does not decrypt it; only its partner does (hence the word "asymmetric,"—each side different).

An analogy is the asymmetric lockers often found in airports, train stations, skating rinks, and many other public places. To securely store your belongings, you put them into the locker and lock it by inserting money. Just as your house key locks your front door, the money locks the locker—in a sense, your money is the key. After you lock the door, you receive another key—perhaps an actual key that looks like your house key or car key, or perhaps a piece of paper that contains a number. To reopen the locker, you use the key or enter the number on a key pad (sort of like using a temporary *personal identification number* or PIN).

Suppose thieves want to steal your belongings. To open the locker, they need a key. The key you used to lock it was money. But if the thieves insert more money into the locker, it won't open. They can stuff money into it all day long, and it still won't open. The key that was used to lock the locker will not unlock it. Only the second, different key will unlock the door.

Similarly, it's possible to create a cryptographic algorithm in which one key encrypts data and the other key decrypts it. Another term for this model (the term we use in this book) is *public-key cryptography*. Because both keys are needed to lock and unlock the data, one of them can be made public without jeopardizing security. This key is known as the *public key*. Its partner is called the *private key*. You encrypt data with the public key and decrypt it with the private key. Just as thieves can know what key was used to lock the asymmetric locker—can even have access to that key—and still not be able to open the door, an attacker can have access to a cryptographic public key and still not be able to decrypt the data. Only the private key can be used to decrypt it, and if the owner of

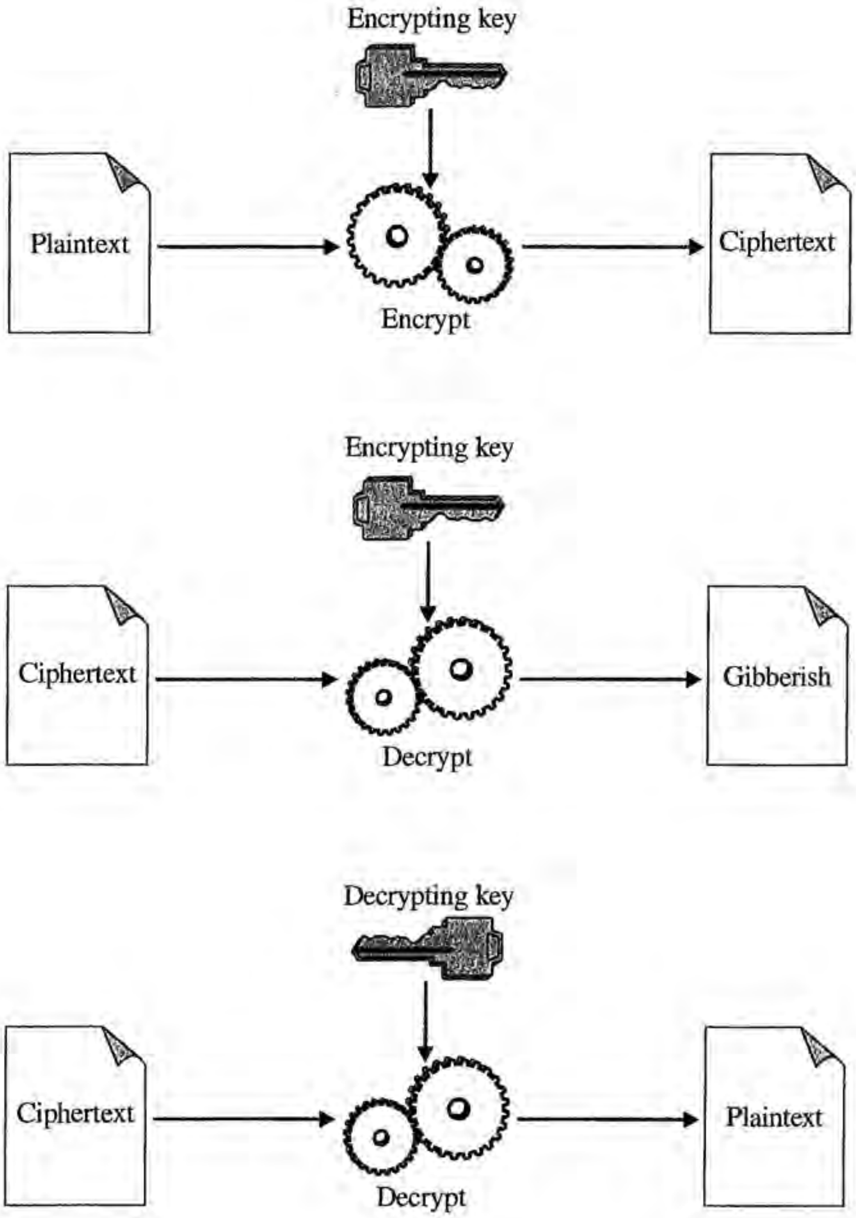


Figure 4-4

In asymmetric crypto, the encrypting key cannot be used to decrypt; you must use its partner

that key keeps it private (as the name implies), plaintext encrypted with the public key will remain secure.

Let's return to our sales rep example. If Gwen has a public and private key pair, she makes the public key publicly available (what else are you going to do with a key called "public"). She is the only one who has access to the private key. Pao-Chi uses a symmetric algorithm with a session key to encrypt his e-mail, and then he uses Gwen's public key to encrypt the session key. Then he sends both the encrypted message and the encrypted session key (see Figure 4-5). This arrangement is similar to password-based encryption, in which the session key is used to encrypt the bulk data, and the KEK (based on the password) is used to encrypt the session key. In PBE, only the owner of the password can recover the session key and consequently decrypt the bulk data. In public-key cryptography, only the owner of the private key can recover the session key and decrypt the bulk data.

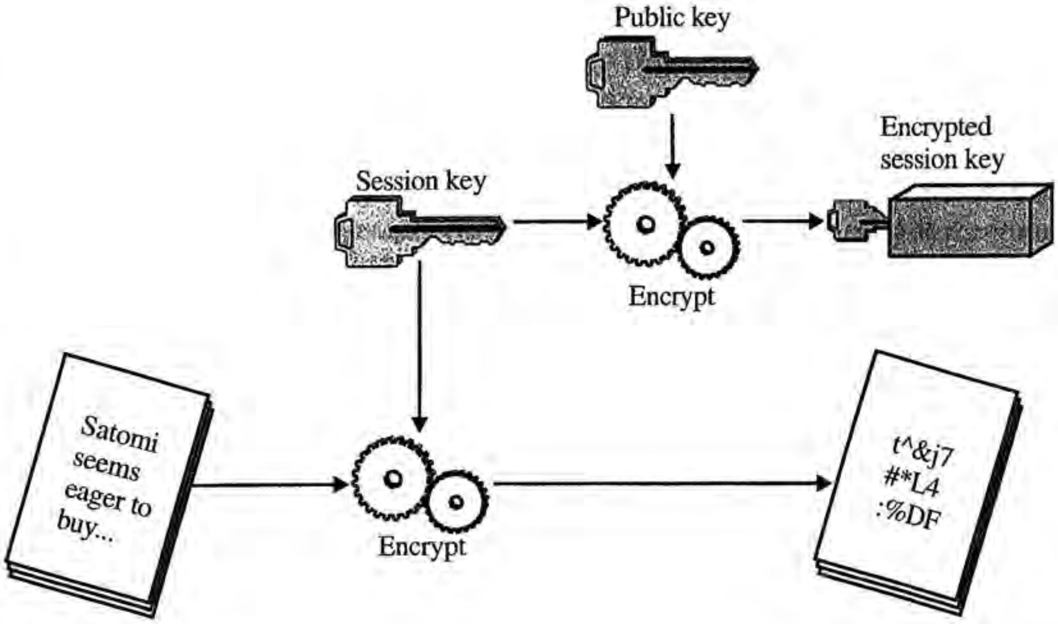


Figure 4-5

You use a session key with a symmetric algorithm to encrypt the bulk data and then encrypt the session key with the recipient's public key

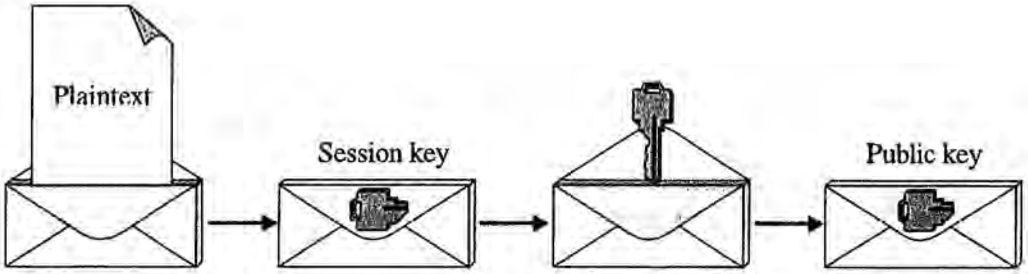
Now you're probably asking, "Why does Pao-Chi use a session key with a symmetric algorithm to encrypt the bulk data and then encrypt the session key with the public key? Why doesn't he simply encrypt the bulk data with the public key?" The answer has to do with performance: Public-key algorithms are slow, whereas symmetric-key crypto can encrypt bulk data very quickly. Depending on the platform, some symmetric algorithms can operate at speeds of 10MB, 20MB, 50MB (or even more) per second. In contrast, a public-key algorithm operates at probably 20KB to 200KB per second, depending on the algorithm, platform, and other factors. That's too slow for processing bulk data, but encrypting 128 bits (the probable size of a symmetric key) would not take much time. So if Pao-Chi's e-mail (the plaintext) is a few megabytes, it's more efficient to use this combination of symmetric-key and public-key crypto.

You may ask, "Why not simply develop a public-key algorithm that can encrypt as fast as the symmetric algorithms?" You're welcome to try.

This process of encrypting bulk data using symmetric-key crypto, and encrypting the symmetric key with a public-key algorithm, is called a *digital envelope*. The idea is that the symmetric key is wrapping the data in an envelope of encryption, and the public key is wrapping the symmetric key in an envelope (see Figure 4-6).

Figure 4-6

A digital envelope. The session key wraps the bulk data in an envelope of encryption, and the public key wraps the session key in another envelope



Notice the huge advantage of this method compared with a shared secret (discussed in the section "Sharing Keys in Advance"). With a shared secret scheme, Pao-Chi and Gwen have a key they use each time they communicate. Each of them must have separate session keys to use when communicating with anyone else. And they must keep all these keys secure. Using a digital envelope, Pao-Chi and Gwen still have to keep a separate key for each individual, but this time it's a public key, which doesn't need to be protected. Furthermore, they probably don't need to store the public keys themselves; directories of public keys are readily available. We talk about these directories in Chapter 6. For now, it's sufficient to know that you can leave the task of managing all those public keys to someone else.

Security Issues

Suppose Pao-Chi sends an e-mail to Gwen using a digital envelope, and Satomi indeed intercepts the message. Will Satomi be able to read it? The bulk data was encrypted using a symmetric algorithm, so she needs the session key. To decrypt the data she could try a brute force attack, but if the key is 128 bits, that would take billions or even trillions of millennia (as you saw in Chapter 2). But because the session key is right there, part of the message itself, it seems she doesn't need to try this attack—except the session key is also encrypted. To decrypt the session key, she needs the partner to the public key that was used to encrypt it because that's the only key that will decrypt it. That's the private key, but only Gwen has that.

Maybe Satomi can break the public-key algorithm or perform a brute force attack to find the private key. Recall that there were two ways to recover messages encrypted using a symmetric-key crypto: break the algorithm or find the key using brute force. The same is true for public-key crypto. If Satomi can figure out what the private key is by breaking the algorithm or using brute force, she can decrypt the session key and use it to decrypt the bulk data.

To determine the private key, Satomi must finds a 160-bit to 510-bit (or possibly higher) number. If a brute force attack on a 128-bit value (the symmetric key) is outside the realm of feasibility, then so is such an attack on a 160-bit number. So a brute force attack on the 160-bit or 510-bit number is not a realistic option.

What about the algorithm? Can a public-key algorithm be broken? It turns out that all public-key algorithms can be broken by determining what the private key is, based on the public key. Remember that the public and private keys are partners, that they're related, and that this relationship is mathematical. Math computations can be used to derive the private key from the public key.

Luckily, these math computations are time-consuming. As with symmetric-key crypto, the longer the public key, the longer it will take to derive the private key from it. If the keys are long enough, solving the math problem would take as much time as a brute force attack on a 96-bit to 128-bit key. In the section titled "Key Sizes," we talk about key sizes for public-key algorithms.

Breaking a Public-Key Algorithm

In Chapter 2, we say that you should use only symmetric algorithms with no weaknesses that the fastest way to break them should be a brute force attack. Why, then, are we now telling you to use public-key algorithms that can be broken? For these algorithms, the brute force attack is not the fastest attack. Why the change of heart?

The answer is simple: No one has been able to develop a public-key algorithm that has no weaknesses. For all public-key algorithms, there are techniques that will break them faster than brute force. Think of these techniques as shortcuts. But most users are willing to live with the shortcuts for two reasons. First, cryptographers have performed a tremendous amount of research quantifying the time required by the shortcuts. Even though an algorithm is susceptible to an attack faster than brute force, the research shows it still takes a long time. For most people, that amount of time is sufficient security. Second, people are willing to use algorithms that suffer from shortcuts because these algorithms are the best way to solve the key distribution problem.

For people who don't trust public-key cryptography, the only recourse is to use a shared secret scheme for key distribution. Otherwise, until someone comes up with a public-key algorithm with no shortcuts, we'll have to live with them.

Actually, though, having the shortcuts is not too bad. Using brute force, an attacker might get lucky and find the key in one of the first few tries, theoretically reducing the time of a successful attack to almost zero. In contrast, cryptographers know how long they can expect it will take to break a public-key algorithm using a shortcut. These attacks usually must run their entire course before coming up with the answer, almost never hitting on a lucky early answer, so researchers have established a more concrete minimum attack time.

Some History of Public-Key Cryptography

In the mid-1970s, Stanford University graduate student Whitfield Diffie and professor Martin Hellman investigated cryptography in general and the key distribution problem in particular. The two came up with a scheme whereby two people could create a shared secret key by exchanging public information. They could communicate over public lines, sending information back and forth in a form readable by eavesdroppers, at the same time generating a secret value not made public. The two correspondents would then be able to use that secret value as a symmetric session key (discussed in more detail soon). The name given to this scheme is Diffie-Hellman, or DH.

DH solves a problem-sharing a key—but it's not encryption. That does not make it unusable; in fact, DH is in use to this day. But it was not the "ultimate" algorithm, one that could be used for encryption. Diffie and Hellman published their result in 1976. That paper outlined the idea of public-key cryptography (one key encrypts, the other decrypts), pointed out that the authors did not yet have such an algorithm, and described what they had so far.

Ron Rivest, a professor at MIT, was intrigued by Diffie and Hellman's idea of public-key cryptography and decided to create the ultimate algorithm. He recruited two colleagues—Adi Shamir and Len Adleman to work on the problem. In 1977, the trio developed an algorithm that could indeed encrypt data. They published the algorithm in 1978, and it became known as RSA, the initials of its inventors.

In 1985, working independently, two men—Neal Koblitz of the University of Washington and Victor Miller of IBM's Watson Research Center—proposed that an obscure branch of math called elliptic curves could be used to perform public-key cryptography. By the late 1990s, this class of algorithms had begun to gain momentum.

Since 1977 (and 1985), many researchers have invented many public-key algorithms. To this day, however, the most commonly used public-key algorithm for solving the key distribution problem is RSA. In second place is DH, followed by elliptic curves. We talk about these algorithms in the following sections.

How Public-Key Cryptography Works

It's easy to imagine symmetric-key crypto. Using the key, you follow a step-by-step procedure to scramble the outgoing data. To decrypt it, you perform the steps in reverse. If the last thing the encryptor did was to rotate a word, the first thing the decryptor does is to rotate the ciphertext word in the other direction by the same amount (see Figure 4-7). If the key used to encrypt the data is the key used to decrypt it, the rotation number will be the same. (If the key is wrong, there is a chance that particular rotation may still be correct, but almost all the rest of the operations down the line, maybe an XOR here or an AND there, will be wrong.)

But with public-key cryptography, such a procedure won't work. You can't simply reverse the steps. Why not? The quick answer has to do with math. Whereas symmetric-key crypto simply operates on the data as bits

Who Invented Public-Key Cryptography?

Because they published the first papers on the subject, Whitfield Diffie and Martin Hellman, along with Ron Rivest, Adi Shamir, and Len Adleman, are generally credited with inventing public-key cryptography in the mid 1970s. Another researcher, Ralph Merkle, also deserves credit for his pioneering work.

Yet British and U.S. information security organizations claim that they developed these techniques in the 1960s and 1970s. Did they?

The Code Book, Simon Singh's history of crypto, gives ample evidence that James Ellis of the *British Communications Electronic Security Group* (CESG) proposed the idea of asymmetric encryption in the 1960s. Apparently, he was inspired after reading an anonymous paper written at Bell Labs during World War II. Ellis had difficulty finding an algorithm that would work. In 1973, mathematician Clifford Cocks joined the CESG. Ellis described the concept to him, and within a few minutes Cocks had devised a solution that was essentially the algorithm known today as RSA. In 1974, Malcolm Williamson, another Ellis colleague, described yet another algorithm, this one similar to the one we call Diffie-Hellman. Because this work was secret (the CESG is a secret organization, called by some people a spy group), it was never published, and the authors did not receive credit until years later.

The U.S. *National Security Agency* (NSA) also claims to have invented public-key crypto in the 1960s. Whitfield Diffie has remarked that part of his inspiration for public-key crypto was hearing about the secure phone system at the NSA. Although Diffie did not know how the NSA had solved the key distribution problem, he explains that because he knew it was possible, he figured he could come up with the solution. The NSA system—which, it was later learned, used public-key crypto—was up and running by the mid-1970s, perhaps indicating that years of study preceded deployment. In addition to the NSA phone system, a document with the exciting title "National Security Action Memorandum 160" outlines a proposal for installing "permissive links" onto nuclear weapons. Apparently, this memo was submitted to President John F. Kennedy; it

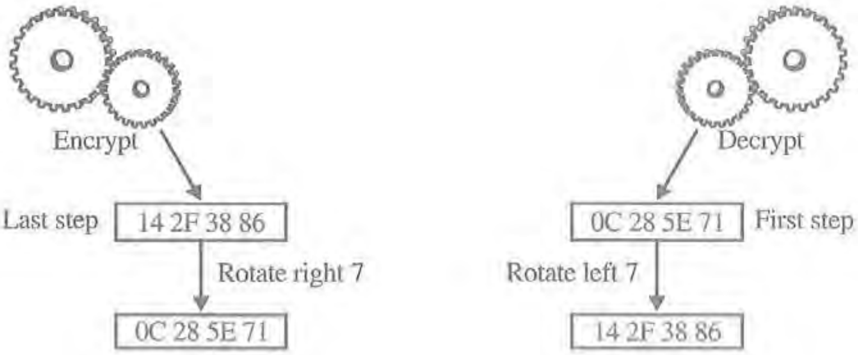
continued

bears his signature. Along with NSAM 160 is the "Weisner Memorandum," which includes more details about permissive links. It can be inferred that the authors proposed equipping nuclear arms with cryptographic switches. Bombs could be activated only with the correct codes, with a form of public-key crypto guaranteeing correct codes (two principles referred to as authentication and nonrepudiation; see Chapter 5).

What about the former Soviet Union or the People's Republic of China? Did these nations have public-key algorithms before 1976? Or how about Hungary or Japan—or any other government? If they did, they're not saying.

Figure 4-7

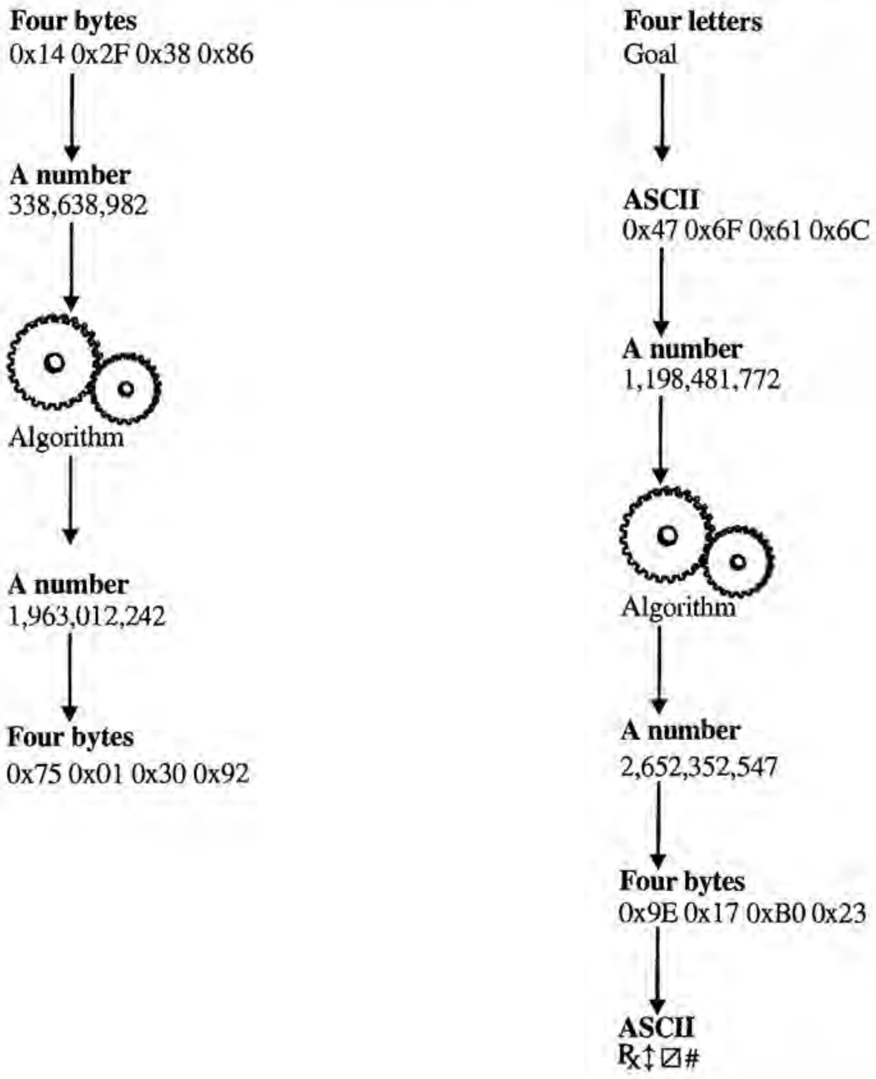
In symmetric-key crypto, generally the last thing done in encrypting is the first thing done (in reverse) in decrypting



and manipulates them using computer operations, public-key crypto operates on the data as numbers and plays with the numbers (see Figure 4-8). And the math is one-way: It's easy in one direction but not in the other direction. In fact, the foundation of any good public-key algorithm is a *one-way function,* the class of math problems on which public-key crypto is built. Actually, public-key one-way functions are more accurately described as one-way with a trap door. To the rest of the world the functions are one-way, but the private key operates as a trap door that allows the owner to recover the original data (see Figure 4-9). There are true one-way functions, and we talk about some of them in Chapter 5.

Figure 4-8

Public-key crypto treats all data as numbers and performs mathematical operations on them



In this book, we don't describe the full details of the math behind the various algorithms; you can find that in the RSA Labs FAQ on the accompanying CD. But in the following sections we talk about the three most widely used algorithms for solving the key distribution problem: RSA, DH, and ECDH (Elliptic Curve Diffie-Hellman). We tell you the names of the one-way functions and outline the problems.

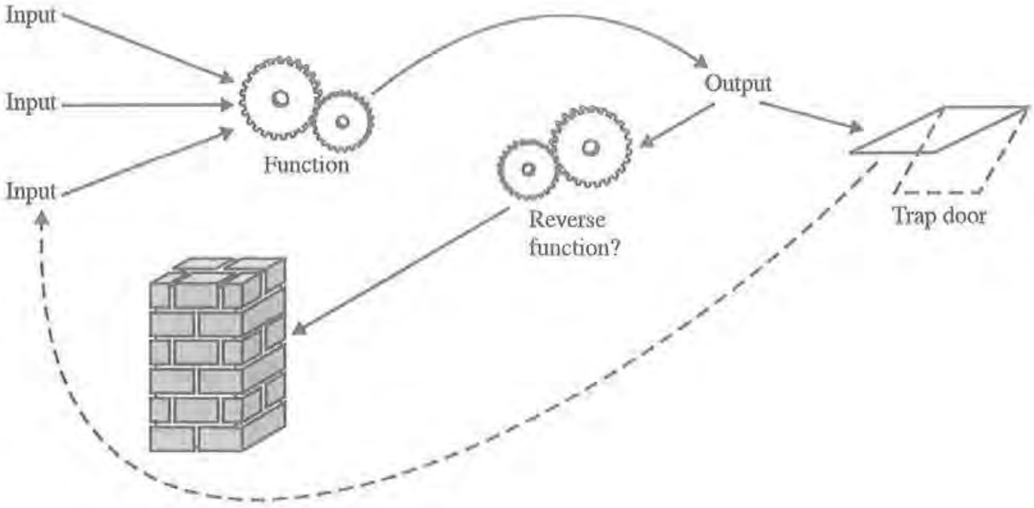


Figure 4-9

A one-way function with a trap door. Performing operations in one direction is easy, but reversing the steps is difficult unless you know the secret trap door

The RSA Algorithm

The RSA algorithm encrypts data. If you feed your plaintext to the algorithm along with the public key, you get ciphertext as a result. With the digital envelope, the plaintext is the session key. It's certainly possible to use RSA to encrypt data other than a session key, but RSA is not as fast as the symmetric algorithms. For example, RC4 (probably the fastest symmetric algorithm in wide use today) will encrypt data at a rate 700 times faster than 1,024-bit RSA (1,024 bits is the most commonly used RSA key size). RC5 (one of the fastest block ciphers) is about 500 times faster.

NOTE:

Incidentally, the R in RC4 and RC5 is the same R as in RSA.

So the best way to use RSA is to create a digital envelope. For example, Pao-Chi can generate a random or pseudo-random 128-bit RC4 key, use it to encrypt his e-mail message to Gwen, and then use Gwen's RSA public key to encrypt the RC4 key. Encrypting the RC4 key (16 bytes) will take only a few milliseconds on most platforms. Pao-Chi's message to Gwen consists of two parts: the encrypted session key and the encrypted bulk data (see Figure 4-10). Gwen separates the two components, uses her RSA private key to decrypt the session key, and then uses that decrypted RC4 key to decrypt the bulk data.

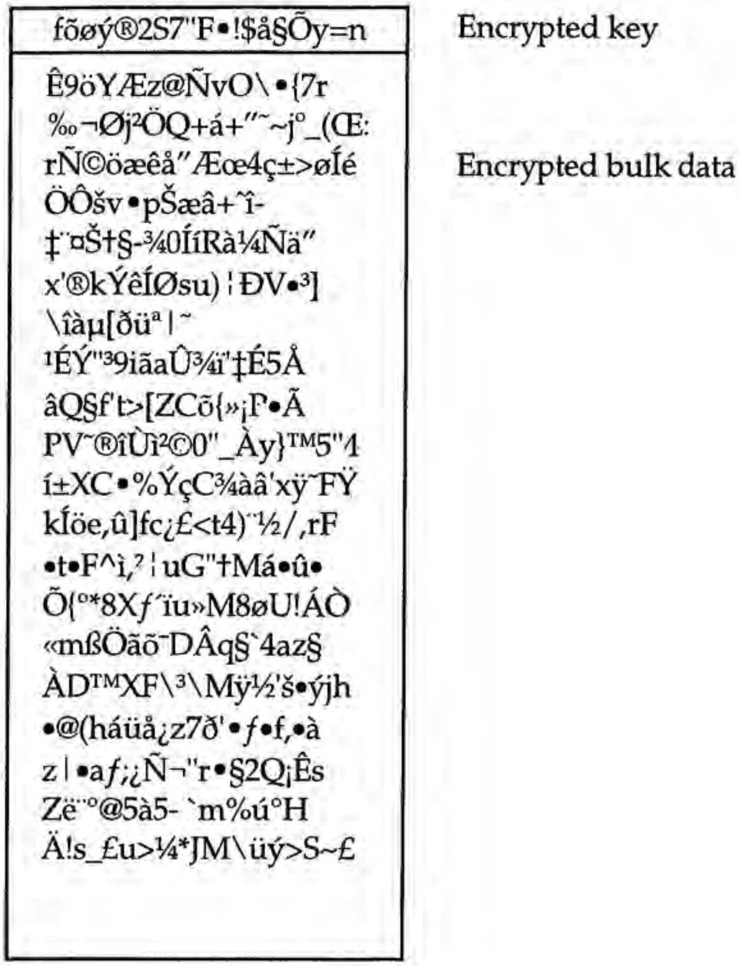


Figure 4-10

In Pao-Chi's message, the encrypted session key comes first and the encrypted bulk data follows

An RSA public key consists of two numbers: a modulus and a public exponent. The private key is made up of the same modulus and a private exponent (see Figure 4-11). The modulus, incidentally, is the product of two very large prime numbers. (A prime number, or prime, cannot be evenly divided; for example, 3, 5, 7, 13, and 17 are primes.) In the cryptographic literature, these numbers are usually given the romantic names n, e, and d, where n is the modulus, e is the public exponent, and d is the private exponent. Equally poetic are the names for the two primes that make up the modulus: p and q.

When you generate an RSA key pair (or rather, when the program you're running generates an RSA key pair), you decide on a public exponent e, find two large primes p and q that work with the e you've chosen, multiply p and q to get the modulus n, and finally compute your private

Figure 4-11

A 1,024-bit RSA key pair. The number *n* is the modulus, *e* is the public exponent, and *d* is the private exponent

Public Key:

n =

```
c6 2c 3f 8c fe c2 95 d1 d9 11 55 ae 94 62 1d b4
f3 0d f2 22 ea a1 62 01 13 22 95 89 3c 0f 89 9f
5e f3 01 2c e8 45 3f d9 2f 99 90 37 4e fa 35 89
0b cf e4 83 cf 9e f7 28 92 a8 89 2b 0b 0b e8 f1
ec 00 f1 e9 30 6f ae 32 16 29 0c 64 71 48 b9 f6
d7 e5 73 db b0 4b be ab d8 a3 83 3f 34 1e 0d 03
d0 70 51 f1 40 df 11 f3 6c 29 6e 7d 5a a6 dc b1
c8 d8 13 1f 57 14 a0 ff 4e d7 de a9 ef 4a 9c b7
```

e =

```
03
```

Public Key:

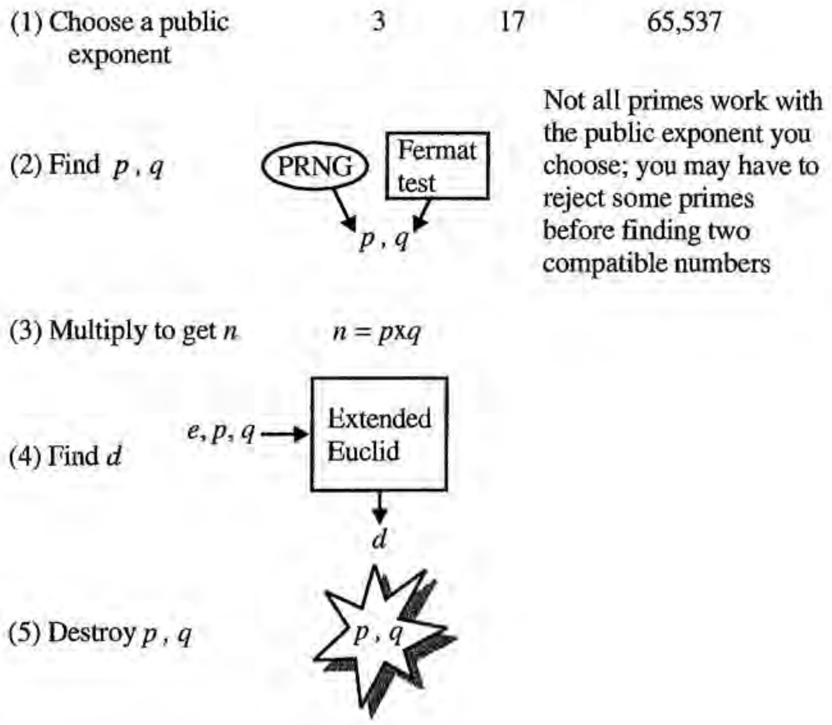
(Use the same *n* as the public key)

d =

```
84 1d 7f b3 54 81 b9 36 90 b6 39 1f 0d 96 be 78
a2 09 4c 17 47 16 41 56 0c c1 b9 06 28 0a 5b bf
94 a2 00 c8 9a d8 d5 3b 75 11 0a cf 89 fc 23 b0
b2 8a 98 57 df bf 4f 70 61 c5 b0 c7 5c b2 9b 4a
c5 56 70 ff 91 e0 c9 e2 67 25 4e f7 d0 a5 f8 73
f5 ec 07 83 73 24 06 76 ed d8 1e e7 d2 f3 6c 3b
af 1c 0b 3e ba 33 e3 34 08 24 f3 b9 51 20 68 0d
ee a4 e3 e7 42 71 90 a6 20 5e 2e dc 2b 4c c0 db
```

exponent *d* based on *e*, *p*, and *q*. Then you throw away *p* and *q* (see Figure 4-12). Incidentally, finding large primes is easy using the Fermat test (in the 1600s, Pierre de Fermat discovered interesting things about numbers, one of which led to a test of primality). Furthermore, researchers have shown in the Prime Number Theorem that there are more primes of 512 bits or fewer than there are atoms in the known universe. This means that we'll never "run out" of primes, and the probability that two people will pick the same prime are so small that we can safely assume it will never happen.

Suppose that Satomi, our attacker, wants to determine Gwen's private key. If Satomi knows the key, she can open Pao-Chi's digital envelope. She must figure out *n* and *d*. Because the public key is, well, public, she knows



Not all primes work with the public exponent you choose; you may have to reject some primes before finding two compatible numbers

Figure 4-12

Generating an RSA public and private key pair

n because it's part of the public key. So really, all she has to do is figure out d. It turns out that d is simply the inverse of e modulo $\phi(n)$. Satomi knows what e is, so all she has to do is find $\phi(n)$ and perform a modular inverse function. That's very easy to do using the Extended Euclidean Algorithm.

NOTE:

Here's an interesting bit of history. Euclid published his algorithm in about 400 BCE, but researchers have concluded that he didn't invent it. It's believed that the algorithm had been around for about 200 years before Euclid presented it. Who was the true inventor? No one knows, but there is a lesson to be learned from this anonymous mathematician: If you get a good idea, publish!

By the way, $\phi(n)$ is known as Euler's phi-function (ϕ is the Greek letter *phi*, pronounced "fee"). Leonhard Euler (pronounced "Oiler") was an 18th-century mathematician who noticed some interesting things about numbers. For example, if n is the product of those two primes p and q, then $\phi(n)$ is $(p - 1)(q - 1)$. That's "the quantity p minus 1 times the quantity q minus 1" (see the FAQ on the accompanying CD for more details).

So Satomi's problem, which began as "find *d*" and was reduced to "find $\phi(n)$," has now been further reduced to "find *p* and *q*." She knows *n* and knows that $p \times q = n$, so all she has to do is factor *n*, which is the hard problem at the foundation of the RSA algorithm.

In other words, in RSA, the one-way function is multiplication. That's right, multiplication. You're probably thinking, "That's not one-way. To reverse multiplication, all you have to do is divide." That's true—if you know what to divide by. But if someone multiplies two numbers and tells you the result, can you determine the original two numbers? That's known as *factoring*, and it happens to be difficult.

Suppose *n* is 35. What are *p* and *q*? That's easy—they're 5 and 7 because 5 × 7 = 35. The numbers 5 and 7 are the prime factors of 35. When you break 35 into its prime factors, you're factoring.

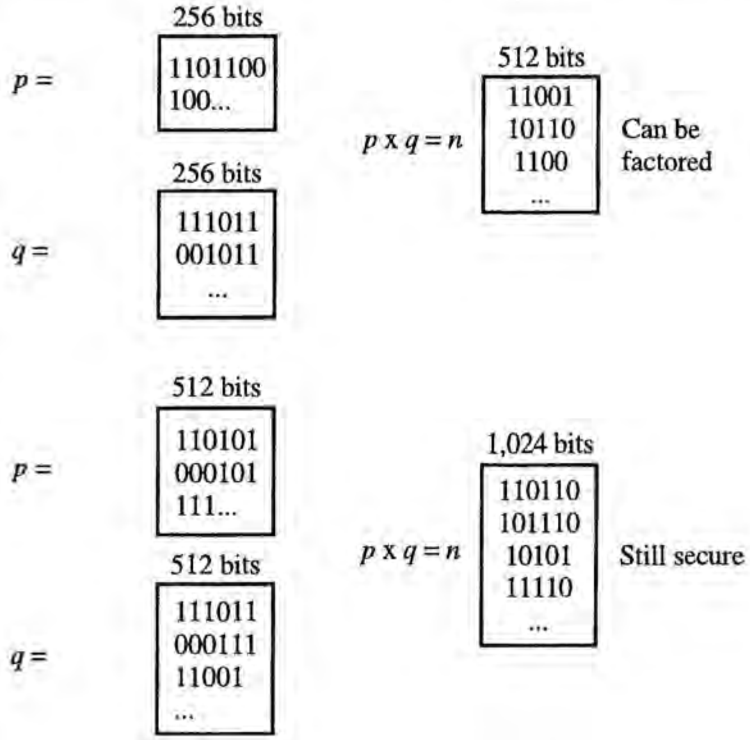
Now suppose *n* is 893. Factor that. (The answer is given in the next paragraph.) If you factored 893, you probably discovered that it was a little more time-consuming than factoring 35. The longer the number, the more time it takes to factor it. Researchers have written computer programs to factor numbers. For those programs, factoring 893 would be trivial. But just as with humans, it takes these programs longer to factor bigger numbers. You can pick a number so big that the amount of time it would take to factor, even for the fastest computers, would be prohibitive.

Remember Satomi's problem? If she finds *p* and *q*, she can compute $\phi(n)$. With $\phi(n)$ and *e*, she can determine *d*. When she has *d*, she can open Pao-Chi's digital envelope. Because $p \times q = n$ and because she knows what *n* is (remember, that's part of the public key), all she has to do is factor *n*—and that's how factoring can break RSA. (The answer from the preceding paragraph is 19 and 47.) Because the modulus (that's *n*) is the number Satomi needs to factor, we'll say that the size of the modulus is the size of the RSA key. Hence, an RSA key that uses a modulus of 1,024 bits is a 1,024-bit key.

No one has been able to factor big numbers in a reasonable amount of time. How big is big? Currently, the most commonly used RSA key size is 1,024 bits. The record for factoring (as of December 2000) is 512 bits. In that case, *p* and *q* were each 256 bits long. It took a team using 292 off-the-shelf computers a little more than five months to do the job. With a brute force attack, each time you add a bit to the key size, you double the time it takes to break. But with the technique used by the current factoring champions, each time you add a bit to the number, you don't quite double the time to factor. Each added bit makes the program run about 1.035 to 1.036 times longer. So if a 512-bit key is broken in five months, a 1,024-bit key can be broken in about 3 to 30 million years (see Figure 4-13).

Figure 4-13

In a popular 1,024-bit RSA key, the modulus is 1,024 bits, built by multiplying two 512-bit primes



You may wonder why the modulus has to be the product of two primes. Why can't the modulus itself be a prime number? The reason is that for a prime number p, $\phi(p)$ is $(p - 1)$. Because your modulus is public, if the modulus were p, a prime number, any attacker would be able to find $\phi(p)$; it's simple subtraction. Armed with $\phi(p)$, an attacker can easily find d.

Incidentally, Satomi has a couple of brute force opportunities. First, she could try to find d by trying every value it could possibly be. Fortunately, d is a number as big as the modulus. For a 1,024-bit RSA private key, d is 1,024 bits long (maybe a bit or two smaller). No, brute force on d is not an option. A second possibility is to find p or q. Satomi could get a number b (call it b for brute force candidate) and then compute $n \div b$ (n divided by b). If that doesn't work (b does not divide n evenly; there is a remainder), she tries another b. She keeps trying until she finds a b that works (one that divides n evenly). That b will be one of the factors of n. And the answer to $n \div b$ is the other factor. Satomi would then have p and q. But the factors of n are half the size of the modulus (see "Technical Note: MultiPrime RSA"). For a 1,024-bit RSA key, p and q are 512 bits each. So Satomi would be trying a brute force attack on a 512-bit number, and

Technical Note: MultiPrime RSA

Faster performance is always a goal of programmers, so anything that would speed up the RSA algorithm would be welcome. The first speed improvement came in 1982 from Belgian researchers Jean-Jacques Quisquater and C. Couvreur. They showed that it's possible to make private key operations (opening a digital envelope) faster if you keep the *p* and *q* around, by using what is known as the *Chinese Remainder Theorem* (CRT). This theorem dates to the fourth century and originated in, as the name implies, China. It's a result of research into how to count columns and columns of soldiers more quickly.

Remember that an RSA private key is made up of the two numbers *n* and *d*, where *n* is built by multiplying two primes, *p* and *q*. When you have your *d*, you throw away *p* and *q*. According to the theorem, if you don't throw away your *p* and *q*, and if, while generating your key pair, you make a few other calculations and save a few more values, the private key operations you perform can run almost three times faster. The fundamental reason is that *p* and *q* are smaller than *n* (there's more to it than that, but at its foundation, that is the reason). Because *p* and *q* must be kept private, this technique will not help public key operations. But, as you'll see in the section "Performance," RSA public key operations are already rather fast. Recently, people have been looking into using three or more primes to make up *n*. Here's why.

When you multiply two numbers, if you add the sizes of those two numbers you get the size of the result. For example, if you multiply a 512-bit number by a 512-bit number, you get a 1,024-bit number because 512 + 512 = 1,024 (it could end up being 1,023 bits, but let's not quibble). Actually, you could multiply a 612-bit number by a 412-bit number to get a 1,024-bit result, but for security reasons, it's better to have the numbers the same size or very close. Virtually all programs that generate RSA key pairs find two 512-bit primes and multiply them to make *n*.

If you want a 1,024-bit number as a result of multiplying three smaller numbers, how big should they be? One possibility is 341, 341, and 342 bits. If *p* and *q* are each 512 bits, and if private key operations are faster because they are smaller than *n* (which is

continued

1,024 bits), will operations improve even more if p, q, and r (let's call our third prime r) are smaller still?

The answer is yes. The more primes that make up the modulus, the faster the private key operations run. It's all because of the Chinese Remainder Theorem.

The problem is that the more primes that make up the modulus, the easier it is to factor. More precisely, if "too many" primes make up the modulus, it's easier. How many is too many? That depends on the size of the modulus. The bigger the modulus, the safer it is to use more primes. Using three primes to build a 1,024-bit modulus will not help an attacker; it will take just as long to factor as does a two-prime number. But should you use four primes to generate a 1,024-bit modulus? That may be too dangerous. If your modulus is 2,048 bits, four primes is safe, but five might not be.

Actually, that issue is still under contention. How many primes is it safe to use at various sizes of moduli? Although there is disagreement in some areas, it is widely believed that using three primes is safe for a 1,024-bit modulus. Research continues on the topic.

So if you hear about MultiPrime RSA, you'll know that it has to do with making private key operations faster by using more than two primes to build a modulus.

that's out of the question. Actually, because p and q are primes, they are odd, so the least significant bit is set; and because they are 512 bits long, the most significant bit is also set, so Satomi would know at least 2 of the 512 bits. So it's not brute force on 512 bits but rather on 510—but that's not much better.

The DH Algorithm

The Diffie-Hellman algorithm is not used for encryption, so how can it solve the key distribution problem? After all, don't you have to encrypt the session key to create a digital envelope?

With DH, you don't generate a symmetric session key and distribute it using public-key technology; instead, you use public-key technology to generate the symmetric session key. Each corresponding party possesses a secret value and a public value. If you combine a private value with the other public value, each individual will generate the same secret value (see Figure 4-14).

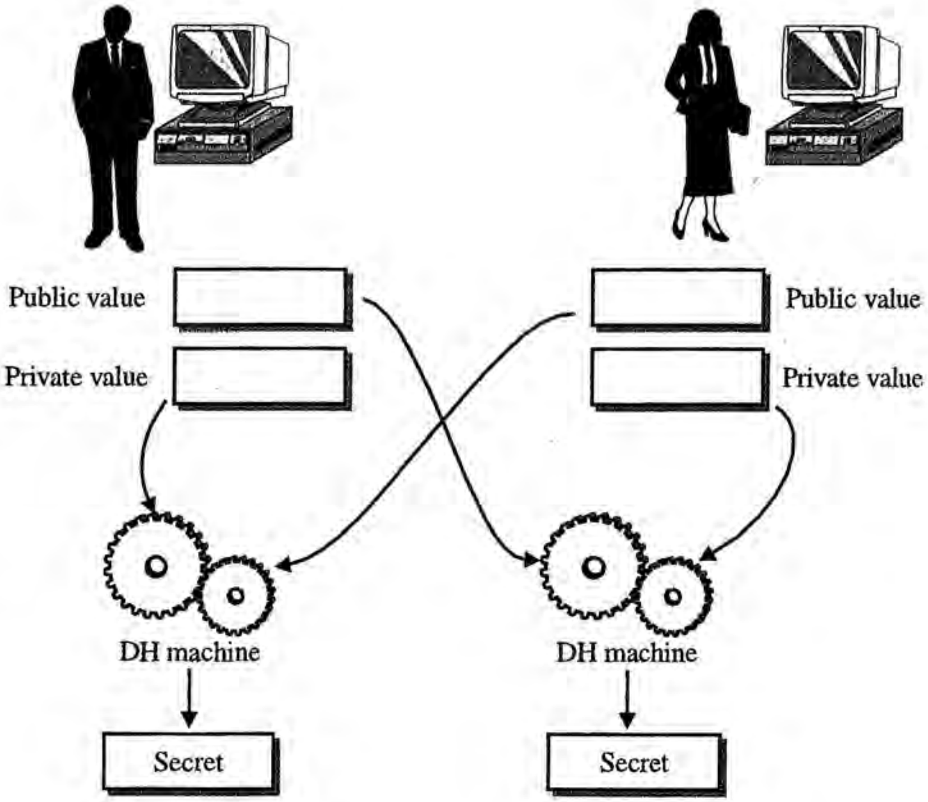


Figure 4-14

With Diffie-Hellman, you combine your private value with the other party's public value to create a secret. The other party combines his or her private value with your public value and creates the same secret

Here's how Pao-Chi and Gwen would make it work. Gwen has a DH key pair; the public key is (obviously) publicly available, and she keeps her private key someplace where only she has access. Inside Gwen's public key is enough information for Pao-Chi to generate his own temporary DH key pair. Now both of them have a DH key pair (see Figure 4-15). For each of the key pairs, the public and private keys are related. But Pao-Chi's and Gwen's key pairs themselves are also related. Pao-Chi uses his private key and Gwen's public key together to generate a number, called a *secret value.*

To encrypt the bulk data, Pao-Chi needs a session key. Instead of using an RNG or PRNG to generate the key, he uses the secret value result from the DH computations. For Gwen to read the message, though, she needs

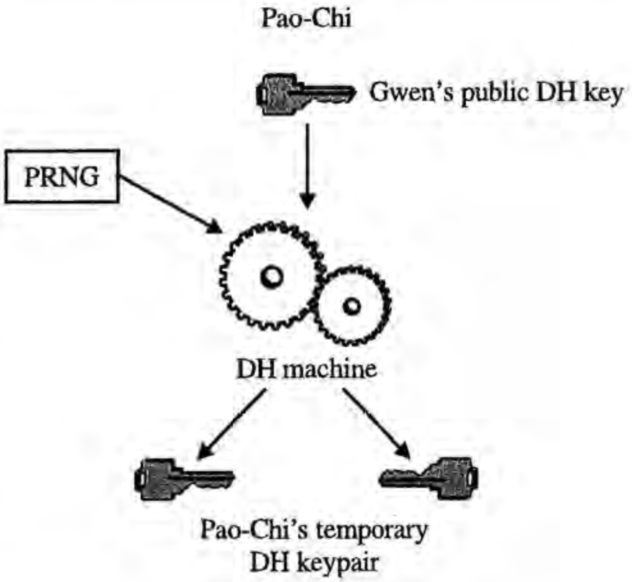


Figure 4-15

Pao-Chi generates a temporary DH key pair using the information from Gwen's public key. Now both parties have related key pairs, and each can create the same secret

the session key. Since Pao-Chi used the DH secret value as his session key, that means Gwen needs the secret value. She can generate the secret value using her private key and Pao-Chi's temporary public key, which he sends along with the message (see Figure 4-16).

Figure 4-16

Pao-Chi's message has his public value first followed by the encrypted bulk data

<table>
<tr><td>VÚvãõœBÕNñÝm_LÂ£</td><td>Pao-Chi's temporary DH public key</td></tr>
<tr><td>ØCª…K¯BÖ•¸ZÖ=¤
C•°=Mþiðªq¢A
'É‰'RÄ!ä"˜øáS×íXPÚj¡ý
ò~5OÔÐ¦/)zyÆN¨Ù•4
"•£ÙÂôÜ|çlWÛÁRéŸC
^22ë-Á'‡!_Ôÿ½i%_÷$÷,ù
ˆaZww"#Èb¥h}WåÜZà¾
‡Ê†À¨éåÅ94Ç' Ãj—X
û9•3÷ÎO"‡¹Yì•3Æcµ8Y
ýÍ,@³þµ"W<ñëk.bî0Ù
d[0ynG¯k'@Æÿ,!±Ø¥þu
•$?½VTØ\)•À¼zo"¿œ/
Ý½H›ô×øþs?ÿÆå<óõÉ
"l¶î#†n7©tza›
%° *ðÍflüëcDEhø -äÐ¥#
ë®L—7ûXÄÊ ‡9,ê¢«í</td><td>Encrypted bulk data</td></tr>
</table>

The Diffie-Hellman algorithm does not encrypt data; instead, it generates a secret. Two parties can generate the same secret and then use it to build a session key for use in a symmetric algorithm. This procedure is called *key agreement*. Two parties are agreeing on a key to use. Another name found in the literature is *key exchange*. That description is not as accurate, but some people use it. It means that two parties perform an exchange, the result of which is a shared key.

But if Pao-Chi and Gwen can generate the secret, why can't Satomi? Satomi knows Gwen's public key and, if she's eavesdropping, Pao-Chi's temporary public key. If she puts those two keys together, what does she have? Nothing useful. The secret appears only when combining a public and a private value (each from a different person). Satomi needs one of the private keys—not both, just one.

A DH public key consists of a generator, a modulus, and public value. The private key is the same modulus along with a private value. As with RSA, cryptographers exercise their creativity to give these numbers more melodious names: g, p, y, and x. The generator is g, the modulus is p, the public value is y, and the private value is x (see Figure 4-17). Here, p is a prime number; note that it's not the product of two or more prime numbers but rather is itself a prime. You generate a key pair by finding the prime p first, then a generator g that works well with your p, and then a random or pseudo-random x. If you combine those numbers using modular exponentiation (see Figure 4-18), you get y.

$$y = g^x \bmod p$$

We have said that there is a way to break all public-key algorithms. That includes DH. Satomi can break DH by deriving one of the private keys from its public partner. Because Satomi needs only one of the private keys, she'll probably go after Gwen's, which has been out there longer (remember, Pao-Chi generates his temporary private key only when he sends the message). Gwen's public key consists of y, g, and p. All Satomi has to do is find x. In the preceding equation, Satomi knows all the values except one. High school algebra describes this as "one equation in one unknown." That's solvable, right?

Yes, it's solvable. It's known as the *discrete log* problem (finally, a more interesting name), and computer programs will solve it. But the longer the p, the more time the computer programs will take—in fact, the same time as it would take to factor. As it happens, the factoring problem and the discrete log problem are related. It's commonly believed that if you solve one you solve them both. So in use, p should be 1,024 bits long.

Figure 4-17

A 1,024-bit DH key pair. The number p is the modulus, g is the generator, y is the public value, and x is the private value

Public Key:

$p =$

```
ab db 22 1a 70 1f 14 7d 84 7d 18 4d fe f2 3e 2a
b1 24 00 05 0e 71 a0 38 d9 cc 3b fa 6d 07 ac 8b
a4 fd 96 75 70 a0 a9 36 a5 0c 03 04 74 4d 48 df
9f 8f 80 3c 69 68 35 ee d0 da 14 9c a5 78 f8 72
0d f6 79 a7 03 24 10 32 65 36 69 e5 0c 21 72 d5
c3 af bd ba 4f 3b b4 c5 67 ff e2 db 0f 4f 80 80
a5 a1 2b 1f 69 4a 3e 87 c1 2d 51 cb 5a 80 13 b2
b1 f0 bd 3a 7f 0b cd 87 9c 62 75 c5 e6 45 2d 75
```

$g =$

```
8e af 41 2d ae f4 89 e7 77 4d af f9 cd d8 8d 32
46 c1 3f ca b8 a6 16 04 c8 84 51 19 a8 f9 67 87
f1 13 5c 5c 3c 38 9e 20 e0 93 dd 01 ea 7a 1e 16
e8 96 b5 6c f1 60 a8 eb 76 c2 c2 42 f5 d8 66 99
ef cd 0a a9 dd 42 33 2f a5 bb f3 73 e1 9c 62 61
e7 47 fc 14 da 8f d2 42 4e d7 e1 57 48 70 d0 c5
6c dd 4e e4 2f 5b 92 d4 96 d3 2e e3 ed 1d d2 3a
b0 54 b6 3c a1 f1 e0 7f ea ad 68 b2 dd 02 f8 b8
```

$y =$

```
29 0e a7 68 b8 72 d6 a3 2b 19 9c 46 62 a8 ab 06
9a 11 08 d5 17 08 ef 06 c7 15 2c 09 82 37 01 e6
62 76 30 0e 60 ea 00 5f 69 31 2d c1 36 f0 0d 16
13 1b fa b6 55 26 6d 93 bd 16 73 77 18 4b 7a b3
d4 37 44 c3 0d 9f a4 33 0c f6 ef d8 89 8d 6d 62
fa f8 db 7e d4 0b b2 e4 a4 03 2c e2 d7 34 cf c3
df fd 62 73 f4 e1 6d 6a 60 8f 01 03 a7 51 21 26
ef ad e1 19 e1 2a d5 6a 74 eb 42 99 f6 0c 50 46
```

Private Key:

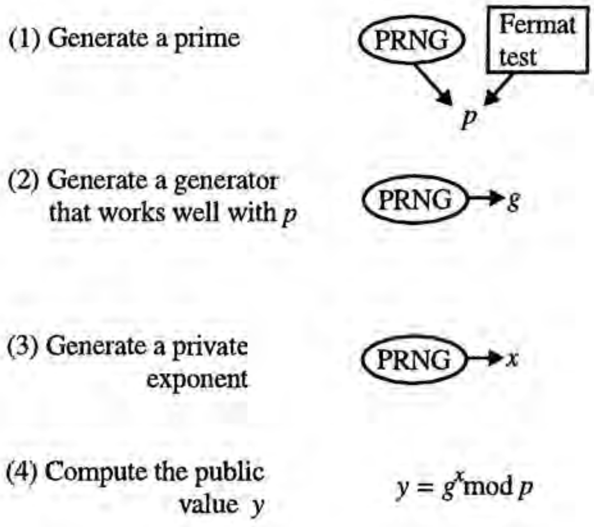
(Use the same p as the public key)

$x =$

```
b7 f0 a0 92 0d 87 27 6b 02 47 d7 cb 98 a2 09 02
13 15 aa 39
```

Figure 4-18

Generating a DH public and private key pair



With RSA, you find two 512-bit primes and multiply them to get a 1,024-bit modulus. With DH, you find one 1,024-bit prime and use it as the modulus.

NOTE:

"Discrete log" doesn't refer to a felled tree that's good at keeping secrets (that would be a "discreet log"). The word "discrete" means that we're working with the math of integers only—no fractions or decimal points—and the word "log" is short for "logarithm."

With RSA, you can't use a single prime as the modulus; you must multiply two primes. But with DH, you use a single prime as the modulus. Why is it that single-prime RSA can be broken but single-prime DH cannot? The answer is that the two algorithms do different things. RSA encrypts data, whereas DH performs key agreement. With RSA, you use a value called d that is dependent on $\phi(n)$. With DH, you don't use d, and you don't mess around with $\phi(n)$.

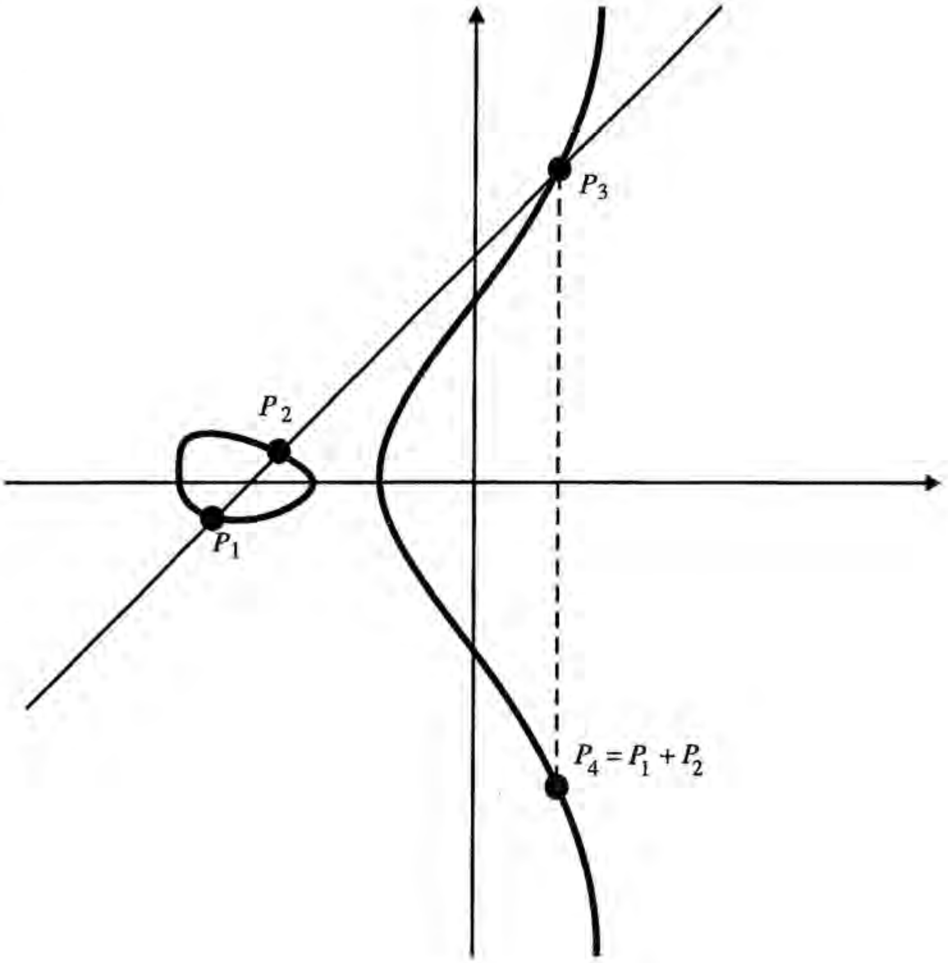
So Satomi will need a few million years to break Gwen's private key by going the discrete log route. What about brute force—would that work? The private key is really just x, a random or pseudo-random number that can be as long as Gwen wants it to be. If she wants it to be 160 bits, she can make it 160 bits. Then Satomi won't be able to mount a brute force attack on it. Gwen could make x even longer, but the longer it is, the longer it will take her to perform her calculations. So for performance reasons, she wants it as short as possible, and for security reasons, she wants it as long as possible. Today, 160 bits is probably the most common size of x.

The ECDH Algorithm

The first thing to know about Elliptic Curve Diffie-Hellman is what an *elliptic curve* (EC) is, and that's shown in Figure 4-19. This curve is not the only form an EC can take, but it's a common one. Actually, it's not even a cryptographic EC, but when cryptographers talk about EC, they generally show a picture similar to Figure 4-19.

Figure 4-19

An elliptic curve. This also shows EC addition

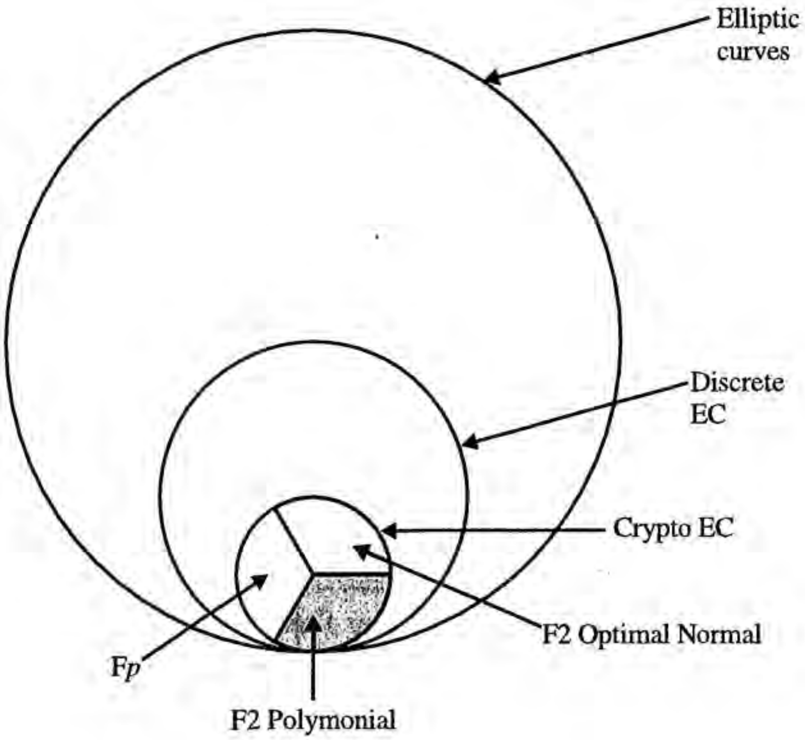


Elliptic curves date to the 1800s. They are actually a form of the Weierstrass equation (a "smooth" Weierstrass equation, to be a little more precise). Karl Weierstrass was a 19th-century mathematician who did pioneering work on number theory. Elliptic curves played a role in the proof of Fermat's Last Theorem and are also involved in factoring.

Cryptographers use only a few of the many flavors of ECs. The curves used by cryptographers fall into two main categories, generally called "odd" and "even." Another way to categorize the types of curves used in crypto is F*p*, F2 Polynomial, and F2 Optimal Normal (see Figure 4-20). These latter categories can be broken down to even more classes of curves.

Figure 4-20

Classes of elliptic curves used by cryptographers



A cryptographic EC is discrete (only integers; no fractions or decimal points). All numbers fall within a certain range. The bigger the range, the more secure the curve; the smaller the range, the faster the computations.

An elliptic curve has *points*; a point is an *x,y*-coordinate. For example, in Figure 4-19, the point labeled *P*3 could also be described as (3,8). The *x*-coordinate is 3, so you start at the origin and go to the right 3 units (the unit—inches, millimeters, or something else—depends on the scale). Then you use the y-coordinate to go up 8 units. The point $P2$ could be $(-6, 1)$: left 6 units (the negative in -6 means left) and up 1 unit. As the figure shows, you can add points on an EC. Notice that it's not an intuitive sense of "adding." You find two points you want to add, draw a line through them, and see where that line intersects the EC. That point is not the solution; the negative of that point is the solution. Why isn't $P3$ the sum of $P1$ and $P2$? Here's why. If you added $P1$ and $P2$ and got $P3$, then what would $P3 - P2$ be? It would be $P1$. But what would $P3 + P2$ be? It would also be $P1$. You can't have $P3 + P2 = P3 - P2$ (unless $P2$ were zero, and it's not). So there's a different set of rules for addition.

The graphical form of elliptic curves (the curve itself, the points, the addition rules, and more) can be described with mathematical equations. You don't deal with pictures; instead you deal only with numbers and equations. And if you're dealing with only numbers and equations, you can write computer programs to do the work. If you have programs that manipulate numbers, maybe you can get crypto. All you need now is a one-way function (with a trap door).

The one-way function is called *scalar multiplication*: You add a point to itself some number of times. We have a point, generally called $P0$ (that's a capital P and a zero; the point is "P – zero"). Add it to itself: $P0 + P0$. Figure 4-19 shows the addition of two distinct points, but there is a way, via another strange rule, to add a point to itself. The special thing about elliptic curves is that if you add a point on the curve to another (or the same) point on the curve, the result is also a point on the curve. If you have an elliptic curve and a point or two on that curve, when you add a point following the special rules you will get another point on that curve—guaranteed. If you have a curve and one or two points on that curve, and the result of adding is not on the curve, it is not an elliptic curve.

So the answer to $P0 + P0$ is another point; let's call it $P1$. Now add $P0$ to that result; let's call it $P2$. $P1 + P0 = P2$. What you've actually done is to find $P0 + P0 + P0$. Another way of saying that is $3 \times P0$. You're multiplying 3, a *scalar* (the mathematical term for a single number), by $P0$, a point (a point cannot be described using a single number; you need two numbers: the x-coordinate and the y-coordinate). You could compute any such scalar multiplication. What's $120 \times P0$? Why, that's $P0$ added to itself 120 times. What's $d \times P0$? That's $P0$ added to itself d times. The result of any scalar multiplication is another point on the curve.

There are shortcuts. If you want to find $120 \times P0$, you don't actually have to do 120 additions; instead, you can use a multiplication program. We just wanted to show you how scalar multiplication is defined.

We said that scalar multiplication is a one-way function. Here's how it works. Suppose you find an elliptic curve (that's not hard to do) and a point on that curve. Cryptographers have again demonstrated their lyrical side by calling the curve E and the point P. You now generate a random or pseudo-random scalar called d. Now you multiply, finding $d \times P$. The answer is some point on the curve; let's call it Q. Now you show the world your curve and those two points; E, P, and Q are publicly available, so the challenge is to find d. That is, if $dP = Q$ inside E, and if you know E, P, and Q, your task is to find d. As with Diffie-Hellman, you have one equation in one unknown.

This is known as the elliptic curve discrete log problem, and, as long as the curve is big enough, no one has found a way to solve it in a reasonable amount of time. Recall that in cryptography, elliptic curves are defined over a specific range. The technical term for this range is *field*. In the three kinds of curves we've mentioned—Fp, F2 polynomial, and F2 optimal normal—the F stands for "field." The p in Fp stands for "prime number." That's a lowercase p, not to be confused with the uppercase P used as the point in the description of the EC discrete log problem (cryptographers sure know how to choose names, don't they?). The 2 in F2 is indeed 2. Actually, it would be more accurate to say F2^m.

If you want to work with an Fp curve, you find a big prime p, and all your calculations will use integers from 0 to $p - 1$. If you want to work in F2^m, choose a size m and all your calculations will use integers from 0 to $2^m - 1$. For more security, you should use a bigger range. But the bigger the range, the slower your computations will be. The most common size is 160 bits to 170 bits.

Here's how Pao-Chi and Gwen would use elliptic curve cryptography (ECC). Gwen generates an EC called E. She finds a point, P, on that curve. Then she generates a random or pseudo-random scalar d and finds $Q = d \times P$. Her public key is E, P, and Q (see Figure 4-21). Her private key is the same curve E coupled with the random or pseudo-random d, which is most likely the same size as the range of the curve.

To send Gwen a message, Pao-Chi gets her public key. It contains enough information for Pao-Chi to generate his own temporary ECDH key pair. Now both correspondents have an ECDH key pair. For each of the key pairs, the public and private keys are related. But Pao-Chi's and Gwen's key pairs themselves are related as well. Pao-Chi uses his private key and Gwen's public key together to generate a secret point on the curve. He uses that secret value somehow as a session key. Because a point is a pair of numbers x and y, the two correspondents will have to decide in advance which bits from those numbers to use as the key. The most common ECDH applications use x), so they just throw away the y (see Figure 4-22).

Figure 4-21

A 160-bit F2 EC key pair. The numbers under *E* describe an elliptic curve (composed of 2^m field, *a*, *b*, order, and cofactor), and *P* and *Q* are two points on the curve related by *d*, a scalar

Public key

E =

2^m field =

```
01 00 00 00 00 00 20 00 00 00 00 00 00 00 00 00
00 00 00 00 07
```

a =

```
00
```

b =

```
0d a9 e3 58 04 7f 39 7a 9d 7a 01 e4 60 67 80 37
e2 38 44 de
```

order –

```
0d a9 e3 58 04 7f 39 7a 9d 7a 01 e4 60 67 80 37
e2 38 44 de
```

cofactor =

```
04
```

P =

```
x-coordinate:
52 2f 38 09 b9 4e dc 39 23 f5 23 60 0e 3b 0b 59
7e cd c8 35
y-coordinate:
3c b3 ff 5d 20 40 c5 38 11 4b 73 fa 82 74 f3 b7
92 26 6a e5
```

Q =

```
x-coordinate:
ea f6 59 3c 0d 9d e1 de 4b 91 f9 95 e5 26 09 a6
93 23 92 8d
y-coordinate:
df 84 76 34 5a b5 69 3b ba 91 d2 f8 f5 38 6e 07
68 39 f4 49
```

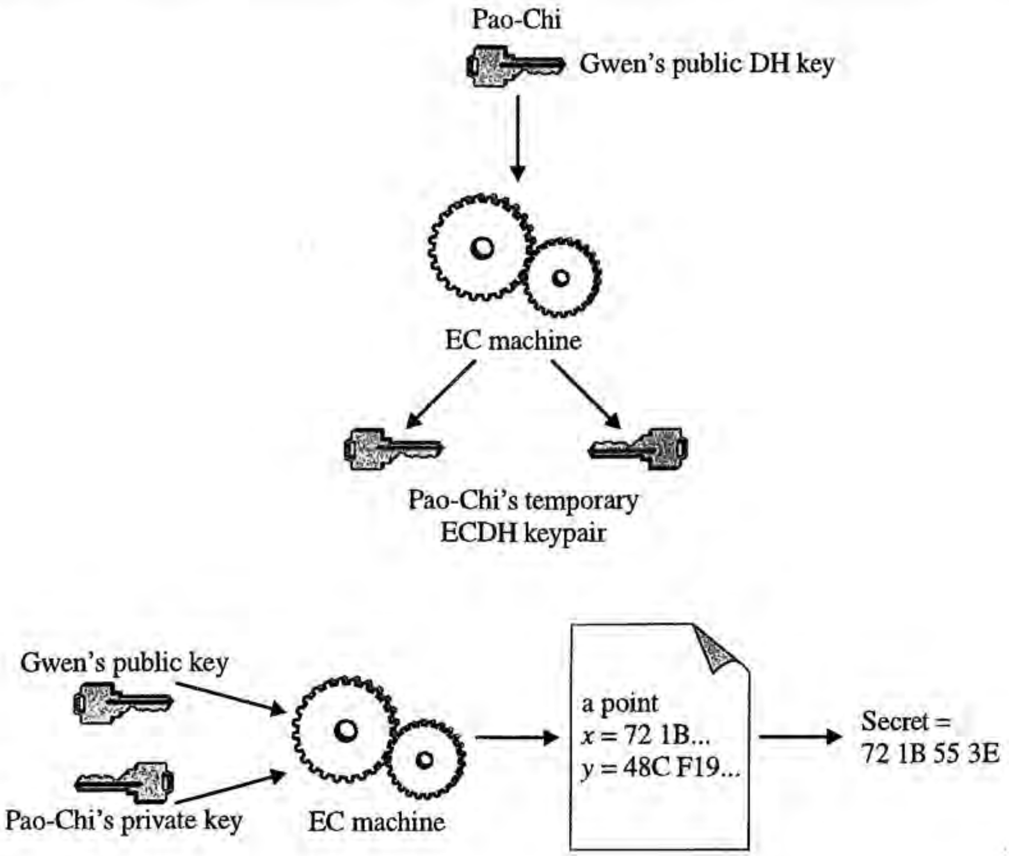
Private key:
(use the same *E* as the public key)

d =

```
32 83 65 87 cc e7 f6 1c 50 1a 72 7d 75 e8 16 d3
bc b2 cb 4e
```

Figure 4-22

Pao-Chi combines his temporary private key with Gwen's public key to get a secret point



To read the message, Gwen needs the session key. She gets it by combining her private key with Pao-Chi's temporary public key (he sends his temporary public key along with the encrypted message).

This sounds just like Diffie-Hellman. In that scheme, two people combine public and private keys in a special way to generate a shared secret. In this scheme, the same thing is happening. The difference is the underlying math, and that explains the name Elliptic Curve Diffie-Hellman.

To read Pao-Chi's intercepted message, Satomi needs one of the private keys, knowing both of the public keys will not do the trick. To break Gwen's private key (probably Satomi's first choice), Satomi must figure out d. That would require her to solve the EC discrete log problem, something that would take a few million years, so Satomi might try a brute force attack. The problem is that d is the same size as the underlying field. Gwen probably chose a 160-bit or 170-bit EC, meaning that d is also 160 bits to 170 bits, so brute force won't work either.

Remember that RSA and DH were based on related problems, and that's why the key sizes are the same. But with ECC, you use a different key size because the underlying problems are different. And solving the EC discrete log problem is harder than solving the factoring or discrete log problem.

By the way, it's possible to use ECC to do encryption. However, in the real world, it's not used very much for security and performance reasons. Recall that as you increase the key size, you slow down the computations. And for ECES (elliptic curve encryption scheme) or ECRSA to achieve the level of security of regular RSA, you must use bigger keys. The keys need to be so big that you take too big a hit in performance.

Comparing the Algorithms

The three algorithms we've discussed can be used to solve the key distribution problem. Which one is the best? There's probably no answer to that question because each has its advantages and disadvantages. A more appropriate question might be, "Which algorithm works best in which situation?" When you're evaluating each approach, it's a good idea to look at five areas: security, key size, performance, transmission size, and interoperability.

Security

Is one of the algorithms more secure than the others? There's no truly objective answer. It depends on what you think is important.

ECC is based on the EC discrete log problem, which is "harder"; does this mean it's more secure than RSA, which is based on factoring, or DH, which is based on the discrete log problem? Not necessarily.

Thousands of mathematicians have been studying the factoring problem for many years (most intently since 1978). Some of them think that if a solution could have been found, it would have been found by now. On the other hand, it took about 300 years to come up with a proof of Fermat's Last Theorem, so maybe the ultimate factoring solution simply has not yet been found. Considering the enormous bank of research available to build on, finding a solution may become easier over time.

ECC is newer and less well understood. Far fewer researchers have been attacking it, and for a shorter time. Some people think that more time and effort are needed to develop a better sense of security. Furthermore, despite the "lag" in research, some classes of curves have been found to be susceptible to cryptanalysis. Of the many flavors of elliptic curves, not all of them are used in crypto. For some flavors, it was known early that they contained more weaknesses than others and that there were ways to break them faster than security requirements allowed. Such curves have never been proposed for use in crypto. Other flavors that were proposed for such use were later shown to possess weaknesses. All the weaknesses found so far lie in the F2 area. At this point, it's believed that no application has ever been deployed in the real world with a weak EC. But because some curves have fallen, some cryptographers are not confident in F2 ECC, and others do not trust any curve at all—*Fp* or F2.

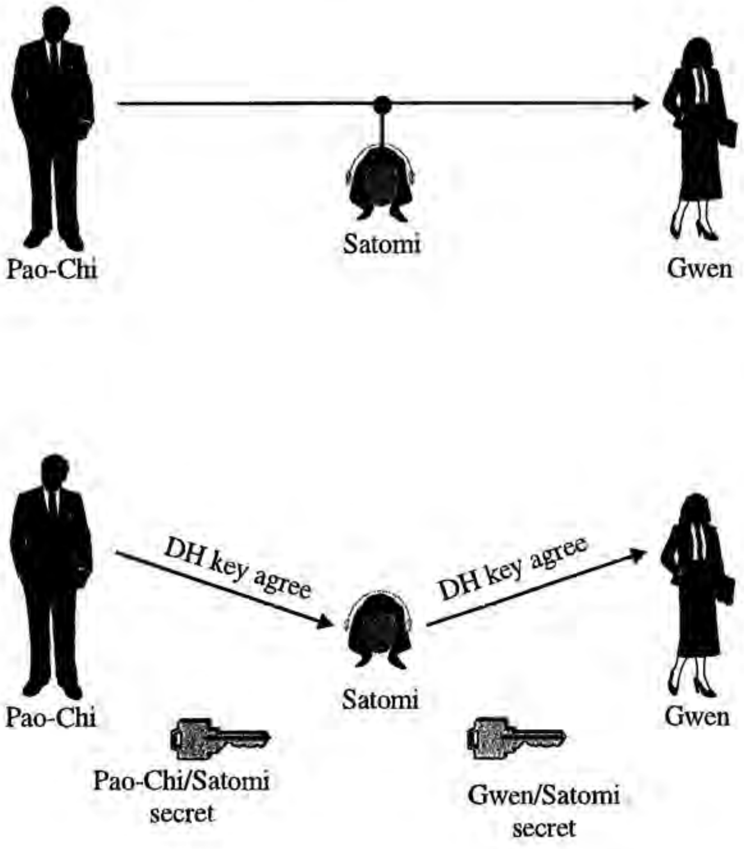
Some people prefer RSA because DH and ECDH are susceptible to the man-in-the-middle attack. In our sales rep example, the potential attacker is a woman in the middle, Satomi. She could intercept all messages between Pao-Chi and Gwen, establishing DH or ECDH keys with each of them. Pao-Chi would think he's computing a shared secret key with Gwen but would really be computing one with Satomi. Similarly, Gwen would compute a shared secret key with Satomi, thinking she was talking with Pao-Chi. Then if Pao-Chi sent a message to Gwen, only Satomi would be able to decrypt it. She would decrypt it, store the message, reencrypt it with the key she established with Gwen, and send it on (see Figure 4-23). The man-in-the-middle attack is easily thwarted by using authentication along with the key exchange (Chapter 5 discusses authentication), and most protocols include authentication anyway. So for some people, this attack is no real disadvantage.

Another issue is each correspondent's ability to contribute to the key. With RSA, only the initiator of the contact has any say in what the session key will be. With DH or ECDH, both parties contribute to generating the session key. Each correspondent performs some operations and sends the result to the other; the final secret depends on each individual's contribution. For some people, this arrangement sounds better than trusting someone else entirely to generate a good key. For others, it's not a great feature. After all, they argue, another party who would do a bad job of generating a session key probably wouldn't do any better with the key exchange.

So, the choice of algorithm is a matter of your own feeling of security. At this time, no honest cryptographer can make a definitive statement about which algorithm is more secure.

Figure 4-23

The man-in-the-middle attack



Key Sizes

The bigger the key, the greater the level of security and the slower any public-key algorithm will run. You want the algorithm to run as fast as possible but maintain a particular level of security. The question is, how low can you go before you jeopardize security? The conventional wisdom is that a 1,024-bit RSA or DH key is equivalent in security to a 160-bit ECC key. There is a little contention on that issue, but research continues. In this book, when making comparisons, we look at 1,024-bit RSA or DH, and 160-bit ECC. With RSA, the modulus is made up of three primes; with DH, the private value is 160 bits.

In April 2000, RSA Labs published a paper that analyzed how long it would take to break the RSA algorithm at various key sizes if an attacker had $10 million to throw at the problem. Table 4-1 summarizes the research; the symmetric key and ECC key columns are there for comparison. With ECC, you could probably get the same results with smaller key

Table 4-1

Time to Break Keys of Various Sizes with $10 Million to Spend

Symmetric Key (Size in Bits)	ECC Key (Size in Bits)	RSA Key (Size in Bits)	Time to Break	Number of Machines	Amount of Memory
56	112	430	Less than 5 minutes	105	Trivial
80	160	760	600 months	4,300	4GB
96	192	1,020	3 million years	114	170GB
128	256	1,620	10^{16} years	0.16	120TB

sizes. However, the assumption in the report is that the public key algorithm should use a key size at least twice as long as the symmetric key (regardless of performance) for security reasons.

The table says that with $10 million, an attacker could buy 105 specially made computers to crack a 56-bit symmetric key, a 112-bit ECC key, or a 430-bit RSA key in a few minutes. Actually, that $10 million would probably buy more than 105 machines, but 105 is all it would take. With the same amount of money, at the next key level the attacker could buy 4,300 machines specially built to solve the problem; at the next key level, 114, and at the next level, 0.16.

Why does the money buy fewer machines as the key size increases? The reason is that the amount of required memory increases. The base computer is the same, but to break bigger keys, the attacker needs more memory (120 terabytes, or about 120 trillion bytes, in the case of a 1,620-bit RSA key), and buying memory would eat up the budget. In fact, the attacker will probably need more than $10 million to break a 1,620-bit RSA key because that amount of money would only buy 0.16, or about 1/6, of a machine.

Performance

If no algorithm wins on security, you might think that you should choose the fastest one. But there is no simple answer there. Comparing the performance of the public key operations (initiating the contact, or creating the digital envelope) shows that RSA is significantly faster than ECC, which in turn is faster than DH. For the private key operations (receiving the contact or opening the digital envelope), ECC is somewhat faster than DH, and both are faster than RSA.

For many machines, though, the difference in performance is negligible. The two times might be 0.5 milliseconds and 9 milliseconds. Even though one algorithm may be 18 times faster, there's no discernible difference between times that are that fast. But if the processor performing the action is a slow device, such as a smart card, a Palm device, or other handheld device, the difference might be 0.5 seconds versus 9 seconds. Or maybe one of the correspondents is a server that must make many connections, maybe several per second. Then the comparison might be 111 per second versus 2,000 per second.

Another factor with ECC is whether you use acceleration tables to speed the private key operations. If you do, you must store extra values in addition to your key. Those extra values amount to about 20,000 bytes. If the device is a server, that's no problem—but will a smart card or handheld device have that kind of storage space?

So the most suitable algorithm depends on which is more important—public-key or private-key operations—in your application. Table 4-2 lists estimates from RSA Security Engineering on the relative performance of the two algorithms. The baseline is an RSA public-key operation, which is 1 unit. As shown in the table, if a particular computer can create an RSA digital envelope in 1 millisecond, it would take that same computer 13 milliseconds to open it. Or it would take that same computer 18 milliseconds to initiate an ECDH exchange and 2 milliseconds to receive one using acceleration tables.

Table 4-2

Estimated Relative Performance of the Public-Key Algorithms

	RSA	DH	ECC	ECC with Acceleration
Public key (initiate contact)	1	32	18	
Private key (receive message)	13	16	6	2
Combined	14	48	24	20

Transmission Size

What if the amount of money it costs or the time it takes to transmit bits across the wire (or in the air) is significant? It turns out that the algorithms differ in the size of the transmission. With RSA and DH, transmission size is the same as the key size. With ECC, you send twice the key size. So using a 1,024-bit RSA or DH key pair means that each time you send a digital envelope, you're adding 1,024 bits to the message. With a 160-bit ECC key, you're adding 320 bits.

Interoperability

With symmetric-key crypto, if you want to make sure that someone else can decrypt your ciphertext, you should use DES, Triple DES, or AES. Any correspondents who have crypto will have those algorithms. You may want to use RC4 or RC5 because they're faster, but to ensure interoperability, you might choose the algorithm you know everyone has.

Can the same be said in the public-key world? For the most part, yes. RSA is almost ubiquitous and has become the de facto standard. If you send an RSA digital envelope, the recipient will almost certainly be able to read it, whether or not your correspondent uses the same application you do. With DH, there's a good chance that the other party will have the necessary code, but it's not as widespread. ECC is even less prevalent than DH. Most applications using ECC today are closed, meaning that they talk only to themselves. The vast majority of those are in the United States. You will find very little ECC used in Europe.

Another problem with ECC and interoperability is that the flavors of curves (F*p* and F2) are not interoperable. If you have code that does F*p* and your correspondent has code that does F2, you can't talk to each other. In the future, the interoperability issue may go away for ECC if more people adopt it and the world settles on a single class. But until that time, your best bet is to use RSA.

Protecting Private Keys

Throughout this chapter, we emphasize the importance of keeping a private key private. How do you do that? The quick answer is that most of the techniques mentioned in Chapter 3 for protecting session keys apply to private keys.

For example, suppose you want a key pair. You'll most likely run a program that generates it for you. You make the public key available to the world, and you store the private key on your computer. Of course, simply storing data on your computer is not safe, so you'll probably store it encrypted, using password-based encryption. When you run the program that uses the private key (for example, when you receive some encrypted e-mail), it loads the data. You enter your password, the program uses it to decrypt the key, and now you can open the envelope.

You can also store the private key on a smart card or other token. The card will generate the key pair and return the public key for you to distribute, but it probably won't allow the private key to leave the device. To open an envelope, you give the token the encrypted session key (if you're using RSA) or the sender's temporary public key (if you're using DH or ECDH). The token performs the private key operation and returns the session key to you. For servers, crypto accelerators might be used. They behave the same way as tokens except that they're much faster.

Using the Digital Envelope for Key Recovery

If you lose your car key, you can often call a dealer in the area who can make a new one. If you lose your house key, you can call a locksmith who can create a new one. If you lose a cryptographic key, there's no one to call. It's gone. That's why many companies implement a key recovery plan.

When Pao-Chi generates a symmetric key to encrypt his files or generates a public/private key pair to be used for key distribution, he stores the symmetric and private keys in such a way that only he can recover them. If he has a key recovery plan, though, he also creates copies of the keys and stores them in such a way that someone else can recover them. In addition, it is possible to store them so that it takes more than one person to recover the keys. In that way, no one single individual can surreptitiously recover the keys and examine Pao-Chi's secret information.

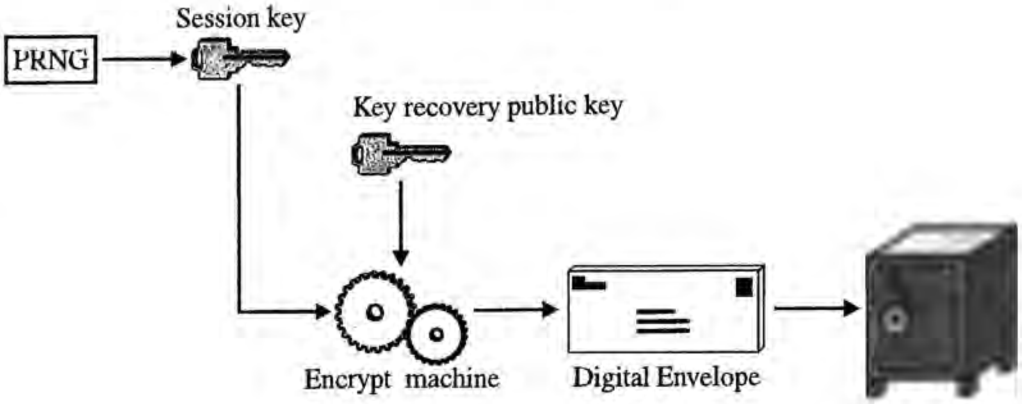
The most common form of key recovery is the RSA digital envelope. Pao-Chi has a software program that encrypts his files. It generates a symmetric session key and uses that key to encrypt each file. He then stores that key securely, possibly using PBE or a token. At the time the session key is generated, he also encrypts it using the key recovery RSA public key (see Figure 4-24). This arrangement is essentially a digital envelope. If Pao-Chi loses his key, the owner of the key recovery private key can open the digital envelope and retrieve Pao-Chi's encrypting session key.

There are three basic entities that can act as a *key recovery agent*:

- A trusted third party
- A group of trustees, each holding a portion of the key
- A group of trustees using a threshold scheme

Figure 4-24

Pao-Chi encrypts his session key with the key recovery public key, storing that digital envelope for emergencies



Key Recovery via a Trusted Third Party

Earlier in this chapter in the section titled "Using a Trusted Third Party," you met Michelle, a TTP who creates session keys for Gwen and Pao-Chi. Now Michelle is going to be their key recovery agent. Michelle generates her RSA key pair and distributes the public key to each individual who will participate in the key recovery program. Pao-Chi's software, for example, can have that public key built-in. When he generates his keys (the session key or public/private key pair), he encrypts them with this public key. He could send this digital envelope to Michelle, but he probably prefers to keep it himself. In that way, Michelle cannot open the envelope without his knowledge. Michelle is a trusted third party, but Pao-Chi's trust in her has some limit. Hence, he will probably store the digital envelope on a floppy disk and keep the disk in his locked desk drawer. Then if Pao-Chi forgets a password, loses his smart card, has a hard drive failure, and so on, and needs to recover a key, he takes the digital envelope to Michelle. She opens it using her RSA private key and gives Pao-Chi the output, namely his key. After he uses the key, Pao-Chi again protects the key.

The Difference Between Key Recovery and Key Escrow

Many elements of cryptography go by different names. There's "symmetric-key" crypto, which is also known as "secret-key" crypto. "Asymmetric-key" crypto also goes by the name of "public-key" crypto, and the terms "message digest" and "hash" (see Chapter 5) are often interchangeable. Now we come to an area of crypto-key recovery and key escrow—in which two terms appear to describe the same thing but are actually significantly different.

Key recovery and key escrow are not the same thing. *Key recovery* is a method that's implemented to restore keys that get lost. *Key escrow* is the practice of giving keys to a third party so that the third party can read sensitive material on demand. "Key escrow" is almost always used to describe a way for governments to obtain keys in order to collect evidence for investigations.

Consider the analogy of your house key. With key recovery, if you lose your key, you hire a locksmith to create a new one. With key escrow, the day you buy the house, you surrender a copy of the key to the police so that they can enter your house when they want to, possibly without your knowledge.

This book is not concerned with the political or practical implications of key escrow. It is our intention only to point out the difference between the two terms. The actual techniques used to implement key recovery and proposed key escrow plans are often the same. So for the rest of this chapter, we describe key recovery schemes.

The advantage of this system is that recovering the key is easy. The disadvantage is that Michelle has access to all the keys. It is possible for her to recover keys without anyone's knowledge. Another disadvantage is that Pao-Chi must depend on Michelle. What does he do when she is away on vacation? What does the company do if she leaves for another job? In that case, the company will have to get a new TTP, generate a new key recovery key pair, distribute the new public key, and have everyone create new digital envelopes with all their keys.

Key Recovery via a Group of Trustees

Some companies and individuals do not like the idea of one person having access to all keys. In such situations, a better scheme is to break the key into parts and distribute them among several individuals. Suppose those individuals are the company's TTPs—Michelle and Alexander—and Gwen, the VP of sales. Now Pao-Chi's software comes preloaded with three public keys. Each of his keys is broken into three parts, and three digital envelopes are created. For example, Pao-Chi has a 128-bit symmetric key that he uses to encrypt the files on his hard drive; this key is separated into three blocks of five bytes, five bytes, and six bytes. Michelle's public key protects five of the bytes, Alexander's protects another five, and Gwen's protects the last six. Now if Pao-Chi needs to recover his key, all three trustees must gather to reconstruct the data.

The advantage here is that no one individual can recover keys secretly. For keys to be recovered surreptitiously, all three trustees would have to agree to subvert the system, a scenario less likely to occur than if only one individual possessed the ability to recover keys.

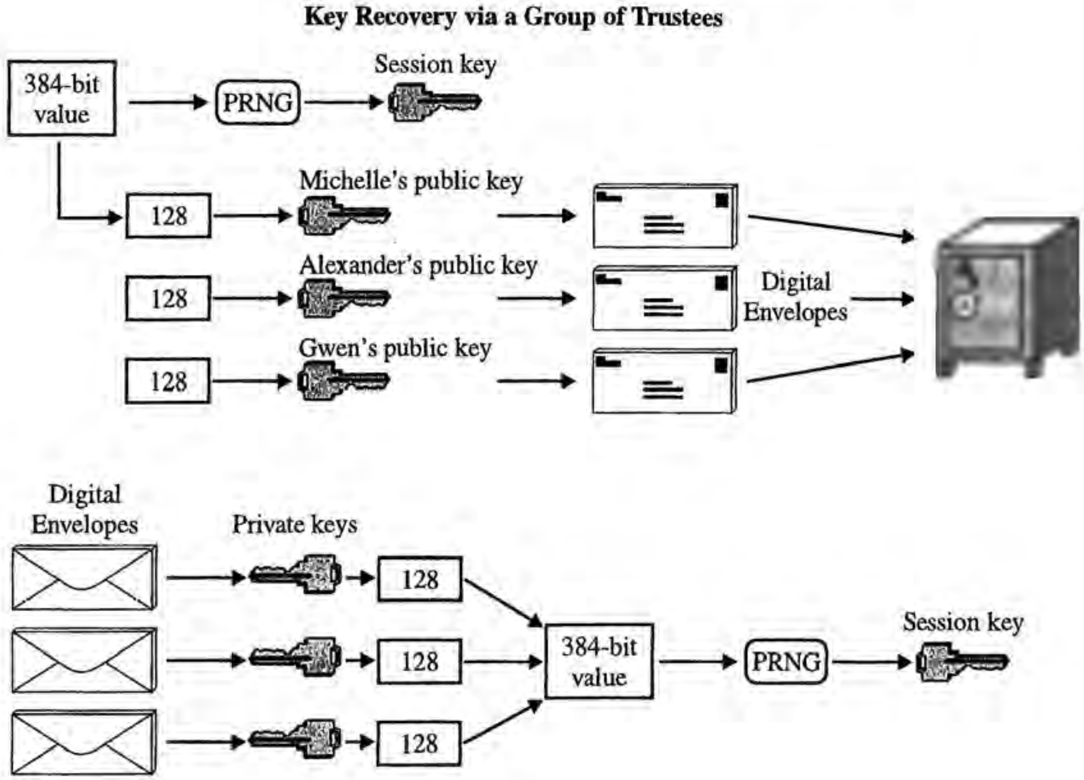
The scheme as described here has a problem. Because each trustee has a portion of the key, it would be possible for an individual to recover the known portion and then perform a brute force attack on the rest. Gwen has the largest portion—six bytes (48 bits)—so her task would be equivalent to breaking an 80-bit key. Such an attack is not likely, but it would be better if that avenue were closed.

One way around this problem is to create a 384-bit value and split that into three 128-bit components. Each trustee knows 128 bits but is missing 256 bits of the total value. The 384-bit value is actually used to derive the key. That is, Pao-Chi generates a 384-bit value and uses it as a seed for a PRNG. The PRNG produces the session key. Each trustee gets a portion of the 384-bit value. To recover the key, you must put all three of the trustees' components together and re-create the PRNG (see Figure 4-25).

This splitting of the secret into multiple digital envelopes has the advantage of preventing one individual from wielding too much power. But it has the disadvantage of being more difficult to implement and also carries all the disadvantages of the TTP approach: If one trustee is on vacation, the key is still lost. Furthermore, if one trustee leaves the company, the key recovery process must start over from scratch, new public/private key pairs have to be generated and public keys distributed, and all employees must create new digital envelopes.

Figure 4-25

Pao-Chi creates a 128-bit session key using a 384-bit seed value and splits the 384-bit value into three portions, encrypting each portion with one trustee's public key. Recovering the session key means recovering the 384-bit value and recreating the PRNG



Key Recovery via Threshold Schemes

Probably the most common key recovery method involves *threshold schemes*, also called *secret sharing* or *secret splitting*. A secret, such as a key, is split into several shares, some number of which must be combined to recover the secret. For example, a secret can be split into 6 shares, any 3 of which can be combined to reproduce the value. Or the secret can be split among 10 shares, any 4 of which can recover the item, or 12 shares with a threshold of 11, or 5 shares with a threshold of 5, or 100 shares with a threshold of 2. Almost any reasonable share and recovery count is possible (as long as the threshold is less than or equal to the share count). For key recovery, the secret is an RSA private key.

If Pao-Chi's company implements a threshold scheme, it might work like this. The company decides how many shares there will be, how many are needed to implement key recovery, and who the trustees will be. Suppose the policy is for six trustees and three shares needed. The trustees are a system or network administrator, the HR director, and representatives from several departments. Say the sys admin is Warren, the HR director is Maria, Gwen represents sales and marketing, the shipping department sends Daniel, Julia comes from engineering, and Michelle is the key recovery administrator.

To start the process, all the trustees gather to generate and collect shares. First, an RSA key pair is generated. Then the threshold program splits the private key into six shares, with each trustee getting one share (see Figure 4-26). The program generating the shares takes as input the private key, the number of shares (six), and the threshold count (three) and produces as output six shares. It's up to the trustees to protect their shares, although the company probably has a policy that defines the procedure. They can simply use PBE on the shares and store them on floppy disks, or they can store them on smart cards or other tokens. After the shares are generated and distributed, the public key is distributed and the private key is destroyed.

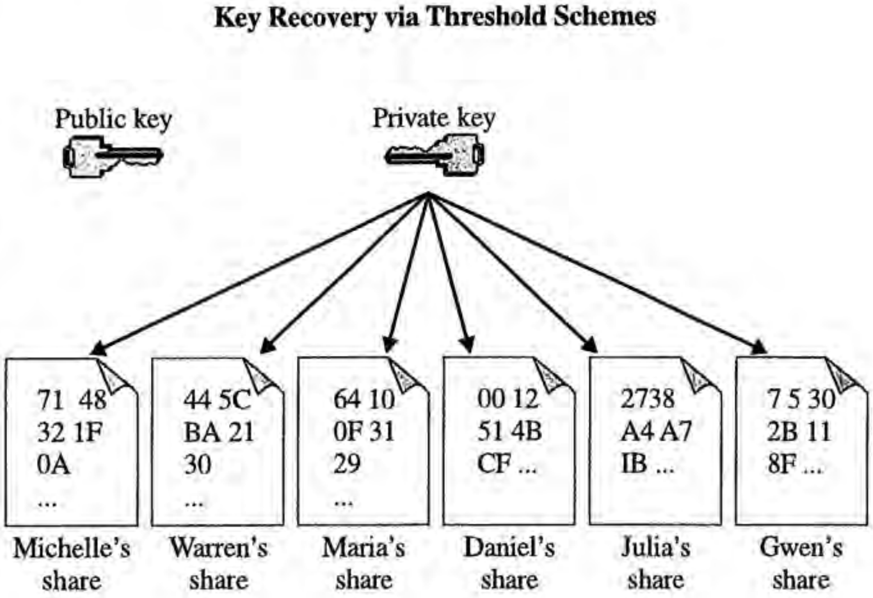


Figure 4-26

An RSA key pair is generated, and each trustee gets one share of the private key, which is then destroyed

Now employees can copy their keys (symmetric encryption keys, key exchange or digital enveloping keys) and encrypt them using the key recovery public key.

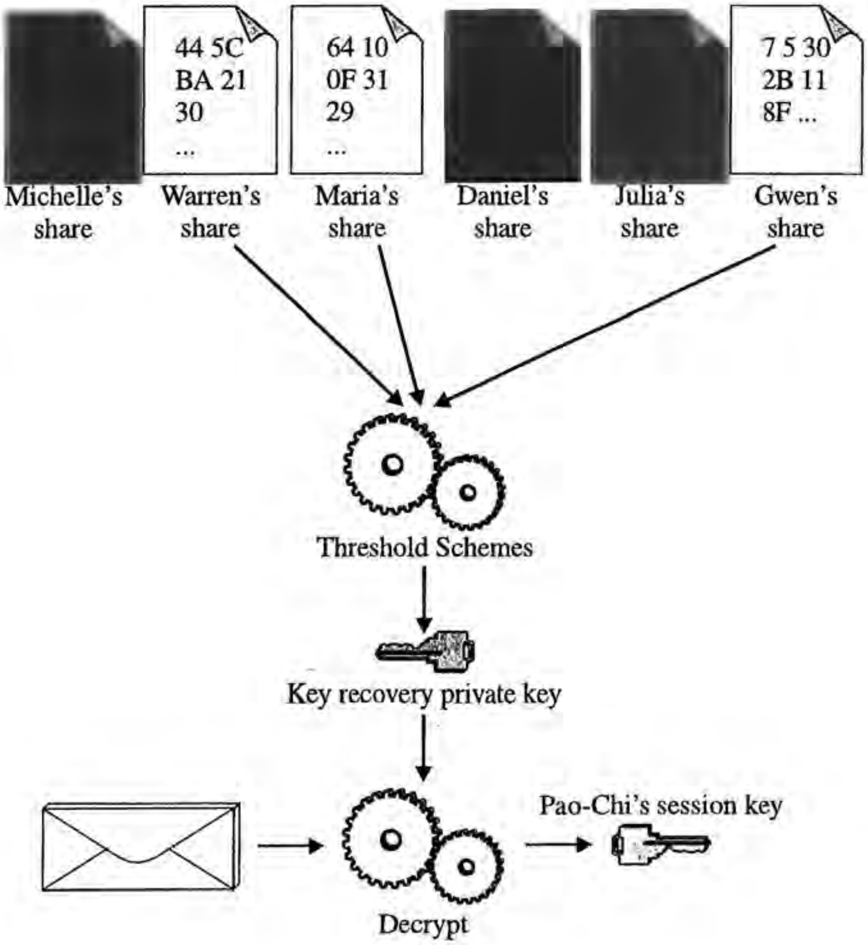
Suppose Pao-Chi encrypts sensitive files on his hard drive and keeps the key on a token. Furthermore, suppose he participates in the key recovery and has created a digital envelope of his session key using the key recovery public key. He keeps that digital envelope on a floppy in his desk drawer. Now suppose he loses his token. How can he recover his data?

To recover the data, Pao-Chi takes the floppy containing the digital envelope to Michelle, the key recovery administrator. If Michelle is out that day, he could take it to Warren, the system administrator, or Gwen, the VP of sales, or any of the other trustees. The trustee he visits must then find two other trustees. The combination of trustees might be Warren, Daniel and Julia, or Maria, Daniel, and Julia. Maybe it would be Warren, Maria, and Gwen, or if Michelle were there that day it could be Michelle, Gwen and Daniel. It doesn't matter; the scheme needs three trustees.

The three trustees give their shares to the program running the threshold algorithm, and the program combines them to produce the secret, which in this case is an RSA private key. Now that the private key is reconstructed, Pao-Chi's digital envelope can be opened. The result is the session key he needs to decrypt the data on his hard drive (see Figure 4-27).

Figure 4-27

Three trustees combine their shares to reproduce the key recovery private key



The threshold scheme has many advantages over the key recovery programs described earlier, and it eliminates some of the disadvantages. First, no one person can recover keys; it takes a group acting together. Anyone attempting to be dishonest must find some co-conspirators. Second, if one of the trustees is unavailable, it's still possible to perform the operation. Third, if one of the trustees leaves the company, the secret is still safe, and there's no need to restart the key recovery process from the beginning.

A disadvantage is that if one trustee leaves the company, his or her share is still valid. By itself, this share can't do anything, but if a threshold number of people leave the company, this group of unauthorized people would have the power to recover the company's secrets. For example, suppose that Warren, Maria, and Julia leave the company, either all at once or over a period of time. They might form their own company, start working for another firm, or work for different companies. If the three of them decide to steal their former employer's secrets, they could re-create the key recovery private key.

Of course, that private key won't do them any good without the digital envelopes protecting the session keys of all the employees. So if they want to steal secrets, they still have to find the floppy disks or tokens storing the encrypted session keys. But a company that wants to eliminate such an attack would generate a new key pair and restart the key recovery program from scratch. Fortunately, with a threshold scheme, this step is not necessary every time a trustee leaves but only when several of them leave.

How a Threshold Scheme Works

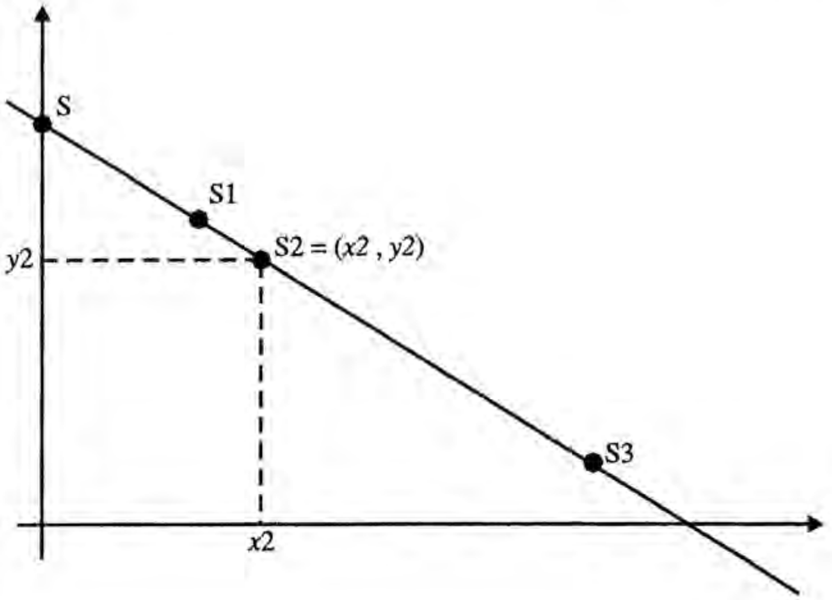
One of the first threshold algorithms was developed in 1979 by Adi Shamir (the *S* in RSA). It's probably the easiest to understand.

Consider the case of a key recovery scheme that uses three shares with a threshold of two—that is, three shares are created, any two of which can recover the secret. You can think of the secret as a point on an (x, y) graph. Any point on the graph can be represented by two numbers: the x-coordinate and the y-coordinate. In Figure 4-28, the secret is the point (0, S). For the Shamir algorithm, the secret is always a point on the y-axis. So let's consider the secret a number, call it S, and then use the point (0, S).

Now you generate a random or pseudo-random line that runs through that point. Next, you find three random or pseudo-random points on that line. In Figure 4-28, these points—the shares—are labeled S1, S2, and S3.

Figure 4-28

The Shamir threshold scheme



To recover the secret, you take two of the points and find the line that runs through them. You might recall from high school algebra that any two points uniquely define a line. With the line just created, you next determine where it crosses the *y*-axis. That's the secret. It doesn't matter which points are used: S1 and S2, or S1 and S3, or S2 and S3. Each pair of points generates the same line. If your scheme uses more than three shares, you simply find additional random or pseudo-random points on the line. To create a line, however, you need at least two points. One point is not enough because an infinite number of lines can run through any single point. Which one is the correct line? It's impossible to tell, and that's why one share alone won't recover the secret.

If you use a threshold of three, instead of a line, the algorithm generates a parabola (a curve of degree 2) that intersects the *y*-axis at the secret. Any three points on a parabola uniquely define it, so any three shares (points on the parabola) can re-create the curve. With the curve, if you find the point where it intersects the *y*-axis, you find the secret. For any threshold count, then, you simply generate a random curve of the appropriate degree (the degree of the curve will be 1 less than the threshold count) that intersects the *y*-axis at the secret. Each share will be a random point on that curve. Of course, a program executing the Shamir algorithm will not do this graphically; instead, it will do all the work using math equations.

Summary

To solve the key distribution problem, you can use public-key cryptography. With the RSA algorithm, the data encrypted by the public key can be decrypted only by the private key. To securely transmit the session key, you can use a digital envelope. With Diffie-Hellman or Elliptic Curve Diffie-Hellman, you can use public-key technology to generate a shared secret. Only the correspondents can create this secret value, which can then be used as a session key.

Each of the three algorithms has its advantages and disadvantages, so it's not really possible to say that one or the other is better. But any one algorithm may be better suited for a specific application.

It's possible to lose cryptographic keys by forgetting a PBE password, losing the token where they're stored, and so on. In addition, a company may want to be able to recover material encrypted by an employee who, for example, has left the firm. For these reasons, many organizations implement a key recovery plan. Generally, key recovery involves the use of an RSA digital envelope, encrypting keys with a recovery agent's public key. The key recovery agent might be an individual or a group of trustees. Threshold schemes offer an attractive means of implementing key recovery with checks and balances. With a threshold algorithm (also known as secret sharing or secret splitting), a secret such as an RSA private key is split into a number of shares. To recover the secret, a minimum number of shares must be collected. This method prevents one individual from obtaining keys surreptitiously, while making it possible to reconstruct the keys even if one or more trustees is absent.

Real-World Example

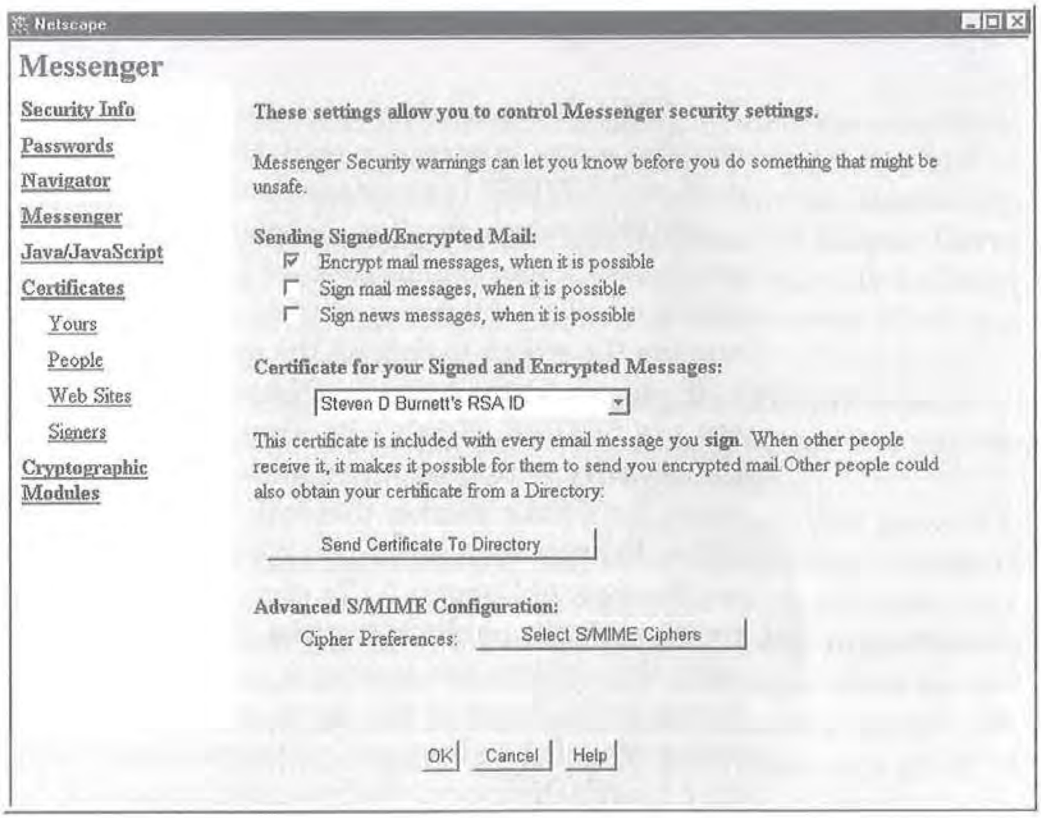
The S/MIME (Secure/Multipurpose Internet Mail Extensions) standard specifies a way to encrypt e-mail. MIME is a widely adopted e-mail standard, and S/MIME is an extension that adds encryption.

S/MIME solves the key distribution problem by using RSA digital envelopes. If your e-mail package is S/MIME-enabled, you can create a digital envelope. All you need to do is get your correspondent's public key and flip the switch to encrypt the message.

If you send e-mail through Netscape Communicator, for example, you can use S/MIME. Here's how. First, launch the Netscape browser. Click the Security button and then click Messenger (along the left-hand column). You'll get a window that looks like the one in Figure 4-29. Click the option Encrypt Mail Messages, When It Is Possible. (The signing options are the topic of Chapter 5.) To encrypt a message, you need to select your correspondent's public key, which you'll find inside a certificate. If you don't already have the certificate, you can search for it in a directory (see Figure 4-30). To get to this menu, click Security Info. Under Certificates (along the left-hand column in the resulting window), click People. Then click Search Directory. After you select the public key, any e-mail you send to that individual will be encrypted using a digital envelope.

If you use Microsoft Outlook 98, click Tools, then Options, and then the Security tab. You'll see a window that looks like the one in Figure 4-31. As with the Communicator program, there is an option to encrypt outgoing messages. Again, you'll need the other party's public key to do that.

Chapter 6 talks about certificates and their directories. For now, you can see that applications today are using public key cryptography to solve the key distribution problem.



Messenger

These settings allow you to control Messenger security settings.

Messenger Security warnings can let you know before you do something that might be unsafe.

Sending Signed/Encrypted Mail:

☑ Encrypt mail messages, when it is possible
☐ Sign mail messages, when it is possible
☐ Sign news messages, when it is possible

Certificate for your Signed and Encrypted Messages:

Steven D Burnett's RSA ID

This certificate is included with every email message you **sign**. When other people receive it, it makes it possible for them to send you encrypted mail. Other people could also obtain your certificate from a Directory.

Advanced S/MIME Configuration:

Cipher Preferences:

Figure 4-29

Netscape Communicator's menu for encrypting e-mail using S/MIME

Communicator will search network Directories for the Security Certificates that are used to send other people encrypted mail messages.

Enter the exact E-mail addresses of the people you are looking for and press Search.

Directory: Netcenter Member Directory

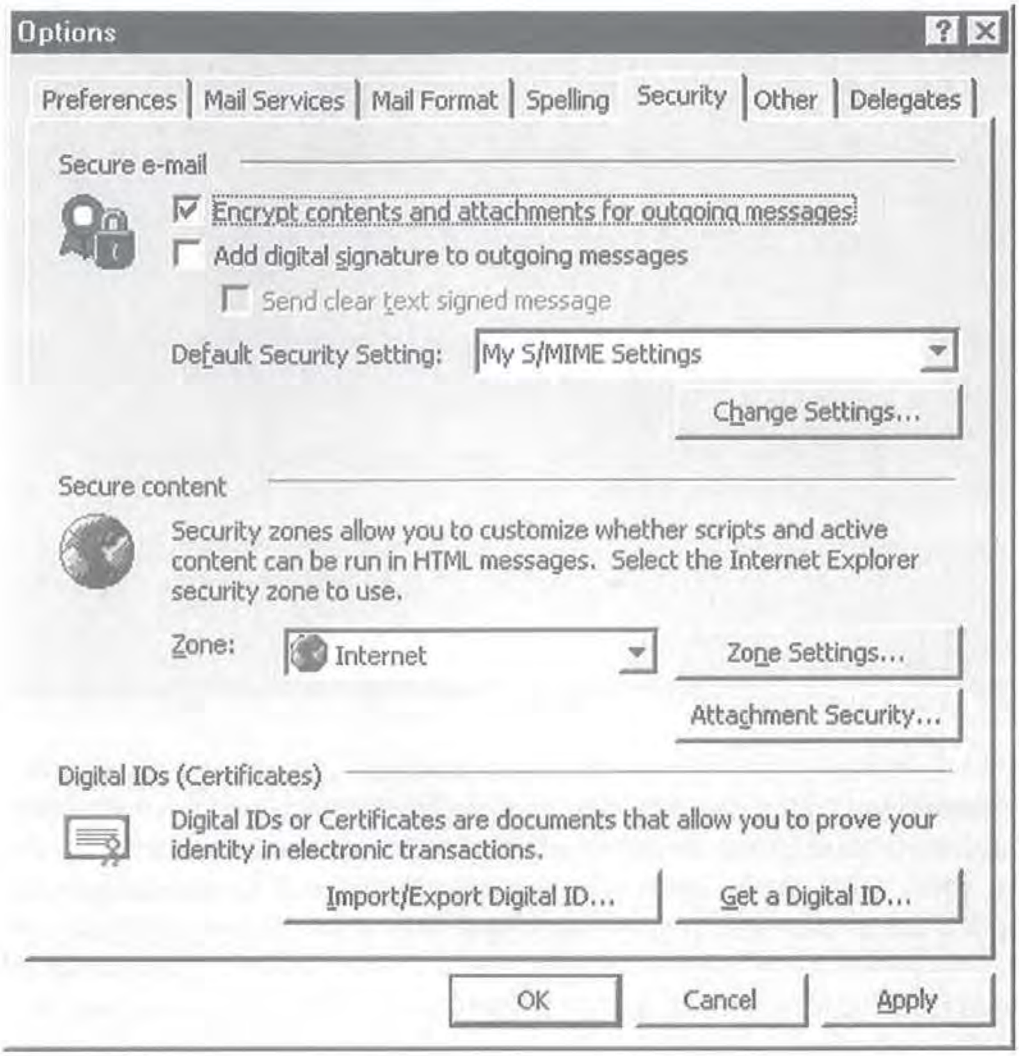
E-mail Addresses: .com

Figure 4-30

A Netscape Communicator menu for finding a public key to use when creating the digital envelope

Figure 4-31

The S/MIME menu in Microsoft Outlook 98



Secure e-mail

☑ Encrypt contents and attachments for outgoing messages

☐ Add digital signature to outgoing messages

☐ Send clear text signed message

Default Security Setting: My S/MIME Settings

Secure content

Security zones allow you to customize whether scripts and active content can be run in HTML messages. Select the Internet Explorer security zone to use.

Zone: Internet

Digital IDs (Certificates)

Digital IDs or Certificates are documents that allow you to prove your identity in electronic transactions.

CHAPTER 5

The Digital Signature

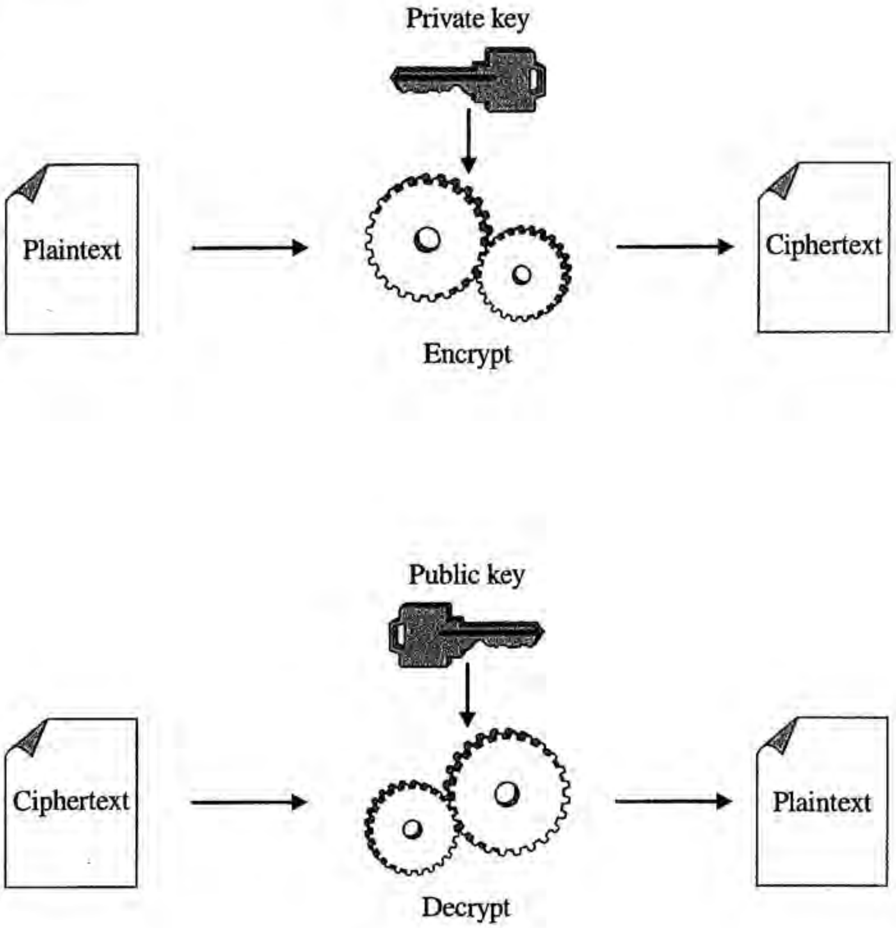
Public-key cryptography helps to solve the key distribution problem. It also addresses two other cryptography issues: authentication and nonrepudiation. Authentication allows someone in the electronic world to confirm data and identities, and nonrepudiation prevents people from going back on their electronic word. One way to implement these features is to use a digital signature.

When you use the RSA algorithm, it means that anything encrypted with the public key can be decrypted only with the private key. What would happen if you encrypted plaintext with a private key? Is that possible? And if so, which key would you use to decrypt? It turns out that RSA works from private to public as well as public to private. So you can encrypt data using the private key, and in that case, only the public key can be used to decrypt the data (see Figure 5-1).

You may ask, "What good is that?" After all, if you encrypt data with your private key, anyone can read it because your public key, which is publicly available, can be used to decrypt it. It's true that using RSA in this direction does not let you keep secrets, but it is a way to vouch for the contents of a message. If a public key properly decrypts data, then it must have been encrypted with the private key. In the crypto community, this technique is conventionally called a *digital signature*. If we didn't "all" agree to call it a digital signature, it wouldn't be, it would be just an interesting exercise in math and computer science. But the crypto community

Figure 5-1

If you encrypt plaintext with an RSA private key, you can use the public key to decrypt it



called it such, the rest of the computer community (hardware and software vendors) have agreed to this nomenclature, and governments are starting to come on board. At the state and national level, laws are being passed that declare a digital signature as a legally binding way to sign documents. This means that anything you encrypt with your private key is a digital signature. So you shouldn't go around encrypting things with your private key unless you're willing to vouch for them.

The Uniqueness of a Digital Signature

Suppose Pao-Chi sells four printing presses to Satomi and must now communicate the sale to the home office. He sends a message to Daniel in the shipping office:

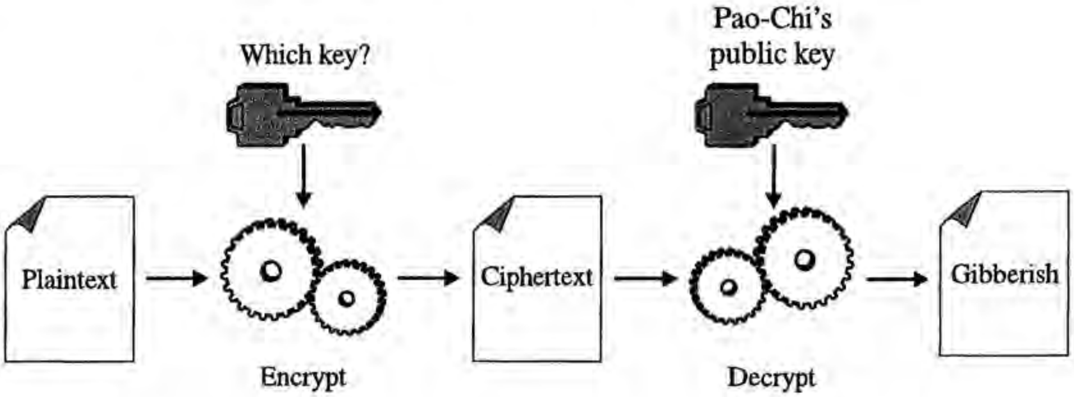
```
Daniel, I sold 4 presses to Satomi. Ship immediately.
```

Pao-Chi can send this e-mail using a digital envelope (see Chapter 4), and only Daniel can read it. But how can Daniel know that this message really came from Pao-Chi and not someone posing as him? For all Daniel knows, Satomi sent that message, maybe she's trying to get four printing presses shipped to her for free. In the paper world, you can look at the signature on a document. Generally, everyone has a unique way of writing his or her name, a way that is supposed to be hard to forge. If Pao-Chi and Daniel have corresponded by paper in the past, Daniel can probably spot the difference between Pao-Chi's signature and a fake, but with e-mail, there's no such signature.

Pao-Chi could encrypt the plaintext (his e-mail) using his RSA private key, producing ciphertext. Daniel could then use Pao-Chi's public key on the ciphertext. If the result of that decryption were gibberish, Daniel would know it was not encrypted using Pao-Chi's private key and would figure Pao-Chi did not send it (see Figure 5-2). Sure, it's possible that the message came from Pao-Chi and that he actually encrypted it using some key other than his private key. But why would he do that? What would he accomplish? No—he's trying to prove to Daniel that he did indeed send the e-mail and that the contents have not been altered along the way. Daniel can safely conclude that Pao-Chi did not send that message.

Figure 5-2

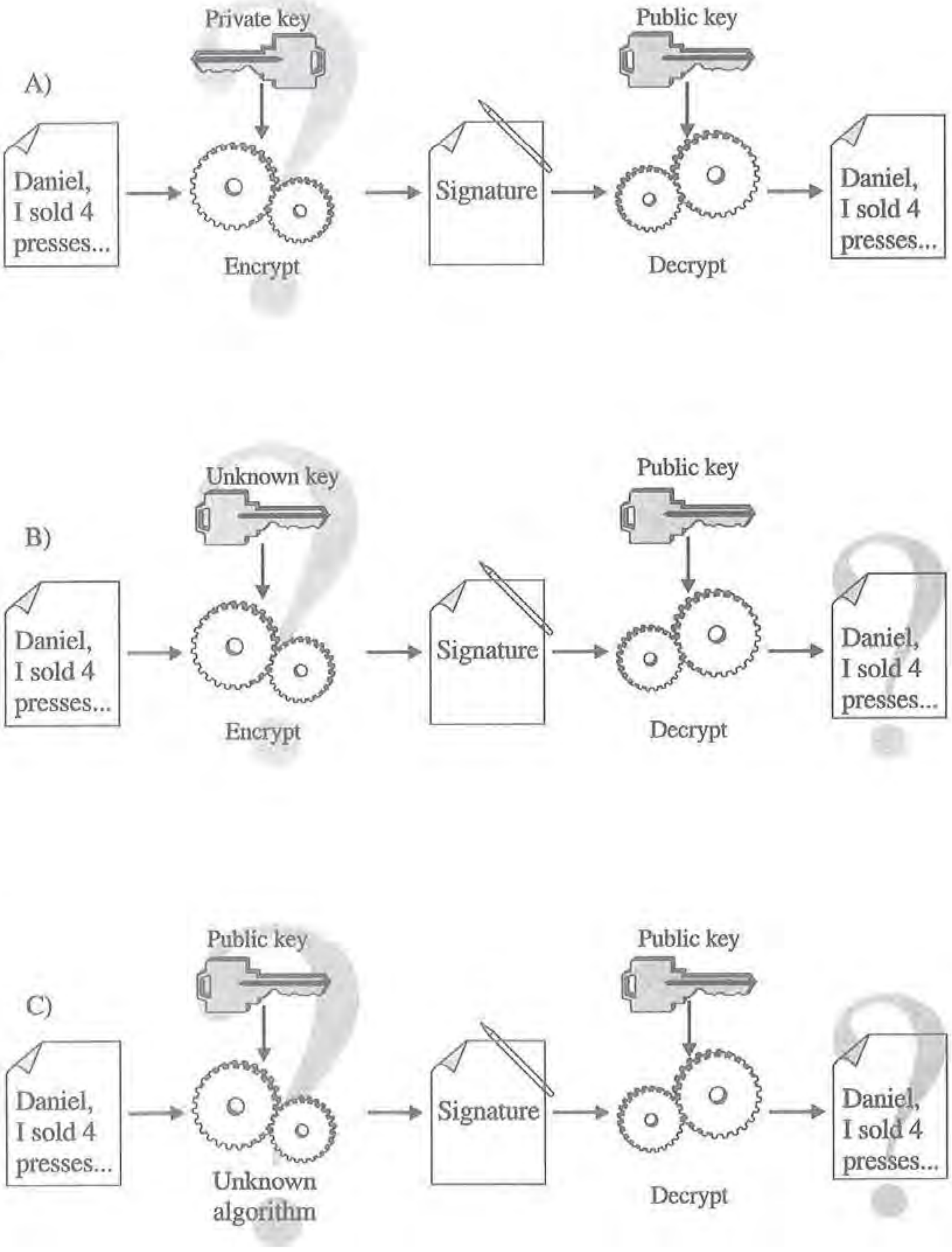
If Pao-Chi's public key produces gibberish, it means the ciphertext was not encrypted with his private key



If, on the other hand, using Pao-Chi's public key produces a reasonable message, it must be that his private key was used to encrypt the plaintext. Is it possible that someone other than Pao-Chi produced a chunk of data that looks like ciphertext and, when "decrypted" with Pao-Chi's public key, produces a reasonable message (see Figure 5-3)? As far as we know, no one has yet been able to do that. So we say there is only one way to produce

Figure 5-3

(A) Pao-Chi's digital signature is encrypted using his private key and verified by decrypting with his public key. (B) If the plaintext is encrypted using a different key, can the resulting ciphertext be decrypted with Pao-Chi's public key? (C) Is it possible to perform some operation on the plaintext, possibly using Pao-Chi's public key as a guide, and produce correct ciphertext?



the ciphertext: Start with the plaintext, and encrypt it with the private key. Because the message was encrypted using Pao-Chi's private key and because we're assuming that Pao-Chi is the only person with access to his private key, it must have come from him. Because it must have come from him, we can call the ciphertext a digital signature. A signature is a way of vouching for the contents of a message—of saying, "Yes, I'm the one who wrote it." In addition, a digital signature lets you check that the data has not been altered.

Digital signatures depend on two fundamental assumptions: first, that the private key is safe and only the owner of the key has access to it, and second, that the only way to produce a digital signature is to use the private key. The first assumption has no technical answer except that keys must be protected (for details, see Chapter 3). But the second assumption can be examined from a mathematical point of view. Is it possible to show that a signature is unique?

Figure 5-3a shows the path that data takes to become a digital signature and to be verified. Is it possible to send data on another path that ends up at the same place? An attacker might want to start with the plaintext, encrypt it with a key other than the true private key, and still produce the correct ciphertext (Figure 5-3b). Or maybe the attacker would try to perform some other operation on the plaintext (not regular RSA encryption), possibly using the public key as a guide, and still produce the correct ciphertext (Figure 5-3c). If that were possible, a digital signature would not be unique. If it were not unique, it would not be possible to claim that the owner of the private key is vouching for the plaintext.

The best that cryptographers can say is that no one knows of any such successful attack. The literature contains phrases such as "computationally infeasible," "it is believed to be true," and "for some classes of signatures, it is possible to prove certain security properties." But no one has completely proven signature uniqueness for any signature scheme. Researchers have spent countless hours trying to come up with alternative paths to break uniqueness, and no one has yet come close.

Message Digests

Because public-key crypto is slow (see Chapter 4), it's not a good idea to encrypt the entire plaintext. Imagine creating an e-mail message, encrypting it using the sender's private key, then encrypting the result with a session key (so that eavesdroppers cannot read it), and then encrypting the session key with the recipient's public key. Such a procedure wouldn't be very efficient, and performance would suffer. So instead of encrypting the entire plaintext with the private key, the best method is to encrypt a representative of the data.

The representative of data in cryptography is a *message digest,* a concept we've mentioned in earlier chapters without defining in detail. We said we would talk about it later, and this is finally the time to describe the details. So for the moment, we're going to take a detour from digital signatures to explain message digests.

Probably the best way to begin a description of what a message digest is would be to give two examples. Here are two messages and their associated SHA-1 digests (SHA-1 is generally pronounced "shaw one").

```
message 1:
   Daniel, I sold 4 presses to Satomi. Ship immediately.
SHA-1 digest:
     46 73 a5 85 89 ba 86 58 44 ac 5b e8 48 7a cd 12
     63 f8 c1 5a

message 2:
   Daniel, I sold 5 presses to Satomi. Ship immediately.
SHA-1 digest:
     2c db 78 38 87 7e d3 1e 29 18 49 a0 61 b7 41 81
     3c b6 90 7a
```

The first thing you notice about these digest samples is that even though the messages are 53 bytes long (each character, including spaces and punctuation marks, is 1 byte), the digests are only 20 bytes. The word "digest" means to condense or to reduce and sure enough, we've taken a 53-character message and condensed it to 20 bytes. No matter what you give to SHA-1, the result will be 20 bytes. Is your data 10,000 characters? The result of SHA-1 will be 20 bytes. Do you have a 200MB message? SHA-1 will produce a 20-byte digest. Even if your message is smaller than 20 bytes, the result of SHA-1 will be 20 bytes.

The second thing to notice about the digests is that they "look random." The bytes appear to be gibberish—a bunch of bits thrown together haphazardly. In fact, you could test the results of digests for randomness (recall that discussion in Chapter 2). Tests of randomness need plenty of input, so you could digest lots of different things, string them all together, and see what the tests say. It turns out that the product of message digests passes tests of randomness. Of course, a digest is not truly random. If you digest the same thing twice using the same algorithm, even on two different computers using two different software packages (assuming they've both implemented the algorithm correctly), you'll always get the same result. So the output of a message digest algorithm is pseudo-random. This is why message digests are often the foundation of PRNGs and PBE.

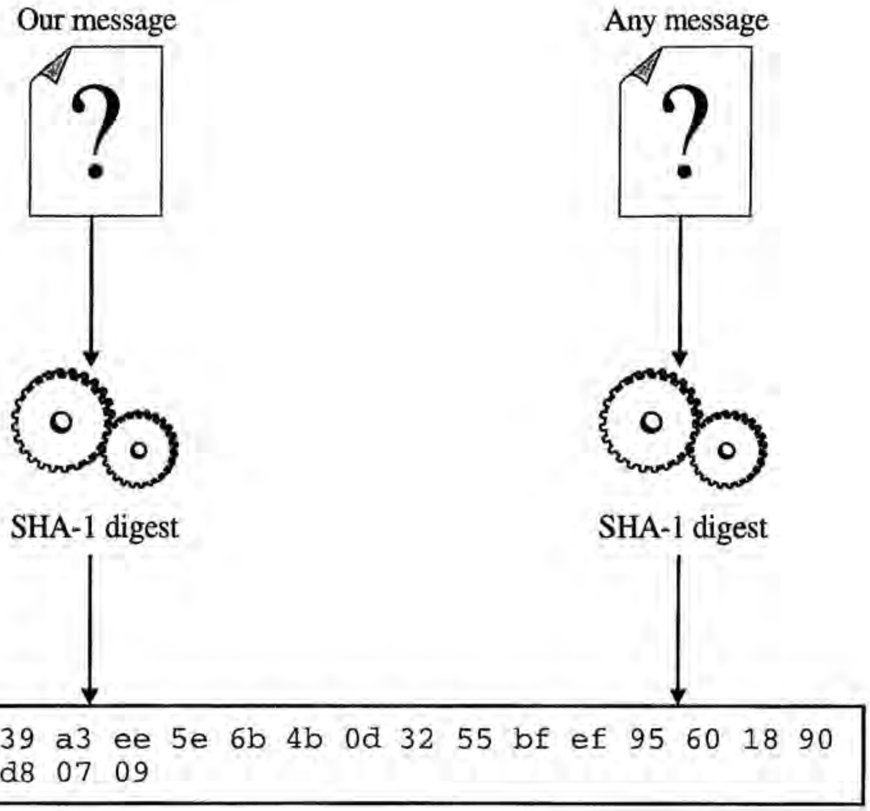
The third thing about the digests is that even though our sample message 2 is almost identical to message 1 (there's really only a 1-bit difference between the two), the digests are dramatically different. That's a quality of a good digest algorithm: If you change the input, you change the output. Two messages that are very similar will produce two digests that are not even close.

So what is a message digest? It's an algorithm that takes any length of input and mixes the input to produce a fixed-length, pseudo-random output. Another word you'll often see used for message digest is *hash*. In fact, the algorithm name SHA-1 stands for *Secure Hash Algorithm*. (The original SHA was shown to be weak, so the designers improved it and called the updated version SHA-1 or SHA1.) The word "hash" can mean a jumble or hodgepodge, which aptly describes the result of a message digest.

Other properties of good digest algorithms aren't as easy to see. First, you can't reconstruct the message from the digest. Here's a suggestion. Have a friend create a message, digest it, and give you the result. Now try to figure out the message. If your friend used a good digest algorithm, that won't be possible. Sure, you could do a brute force attack by trying every possible message, digesting it, and seeing whether it matches. If you did that, you would eventually find it. But your friend's message is one of a virtually infinite number of possible messages. In Chapter 2, you saw how long it would take to find a 128-bit value; imagine how long it would take to find a message that could be of any possible length? For good algorithms, no one has yet been able to figure out the message from only the digest. In other words, it's a one-way function. Remember that Chapter 4 talked about one-way functions with trap doors. A message digest has no trap door.

Another property of a good digest algorithm is that you can't find any message that produces a particular digest. You've seen that you can't find *the* message your friend used to produce the digest, but can you find *any* message that will produce the value? No one has yet come up with a method that can find a message that will produce a given digest.

The last property is that you can't find two messages that produce the same digest. Here, you're not looking for a particular digest but rather two messages that produce the same result, whatever that result may be. Again, with good algorithms, no one has yet been able to do that. The brute force attack would be to digest a message, save the message and result in a table, digest another message, compare it to the first one, and save the result in the table, and then digest another message, compare it to all previously saved values, and so on. Figure 5-4 illustrates these properties with a challenge: Find the message, or any message, and produce the given digest.



```
da 39 a3 ee 5e 6b 4b 0d 32 55 bf ef 95 60 18 90
af d8 07 09
```

Figure 5-4

Can you find the message we used to produce this digest (or any message that will produce it)? If so, you will have found a collision in SHA-1

NOTE:

By the way, you probably already know this, but for the sake of completeness, let's say it. A "message" is not necessarily a communication between two people. Any data you give to a digest algorithm is a message, even if it's not in human-readable form. Each byte of input is simply a byte of input, whether or not the byte is an ASCII character.

Collisions

When an algorithm violates one of the last two properties discussed in the preceding section, the result is a *collision*, the technical term to describe a situation in which two messages produce the same digest. A collision occurs when a second message produces the same digest as a previous message, or when two messages—any two messages—produce the same digest whatever that digest is. If two messages collide, they meet at the digest.

Although the number of possible messages is virtually infinite, the number of possible digests is finite. With SHA-1, the number of possible digests is 2^{160}. Clearly, there will be many messages that produce any one digest. To show that, let's use the time-honored mathematical tool known as the *pigeonhole principle.* Suppose you had a cabinet of pigeonholes (see Figure 5-5). Each pigeonhole corresponds to a digest. The zeroth pigeonhole is for the digest `00 00 . . . 00`, the first is for `00 00 . . . 01`, and so on, until you reach the last pigeonhole, the place for `FF FF . . . FF`.

Now you start digesting messages. After you digest a message, place the message into the pigeonhole of the digest it produces. For example, the digest of the 1-byte message `00` is

```
5b a9 3c 9d b0 cf f9 3f 52 b5 21 d7 42 0e 43 f6
ed a2 78 4f
```

So you place message `00` into pigeonhole `5B A9 . . . 4F`. The digest of message `01` is

```
bf 8b 45 30 d8 d2 46 dd 74 ac 53 a1 34 71 bb a1
79 41 df f7
```

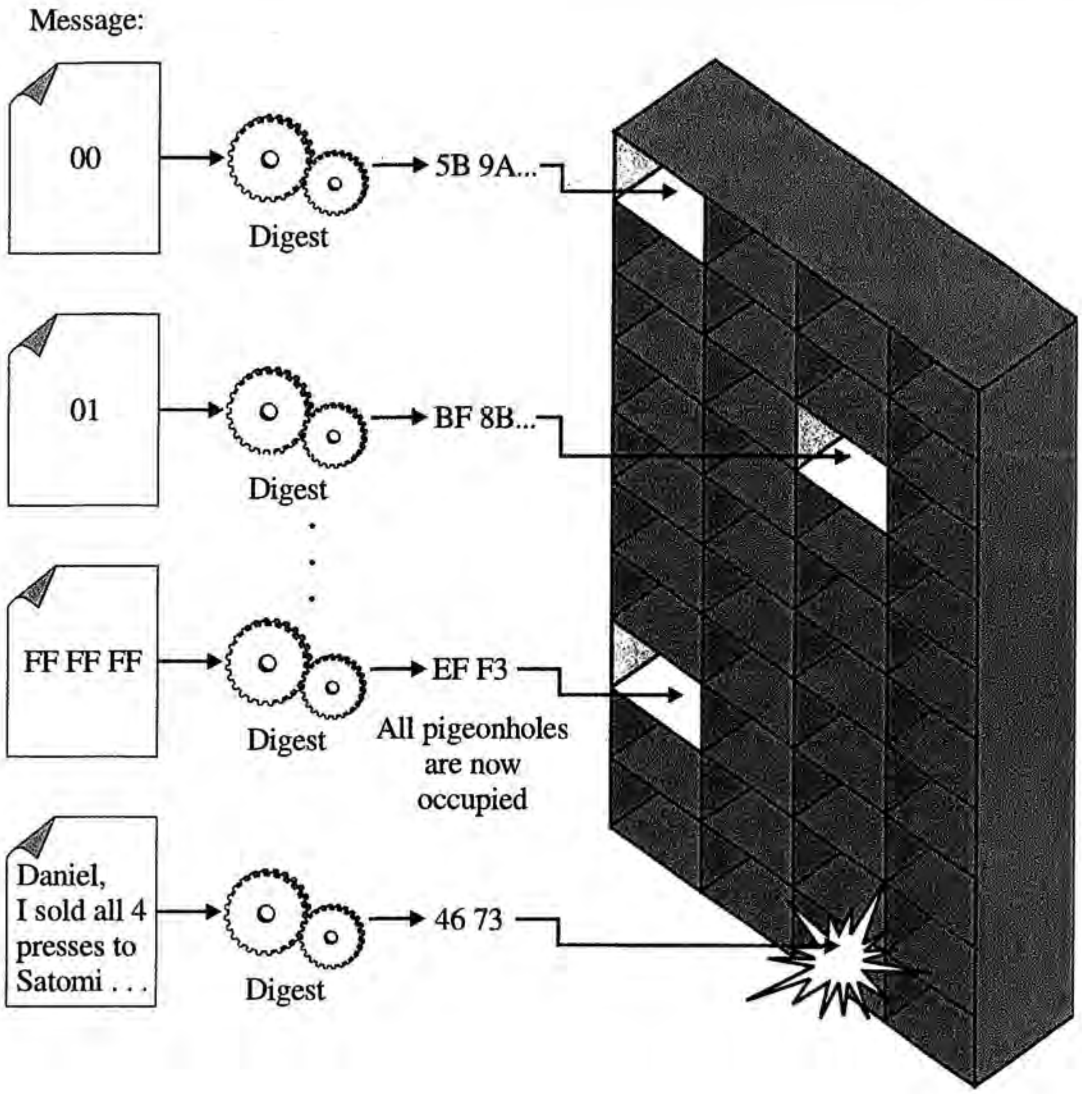
Message `01` goes into pigeonhole `BF 8B . . . F7`.

Suppose you keep digesting messages, the next message being the preceding message plus 1. The sequence of messages is `00, 01, 02, . . ., FF, 01 00, 01 01, . . ., FF FF, 01 00 00`, and so on. Suppose you did this for 2^{160} messages. The last message in the sequence would be

```
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF
```

Figure 5-5

The pigeonhole principle says that sooner or later some messages will collide in the same digest



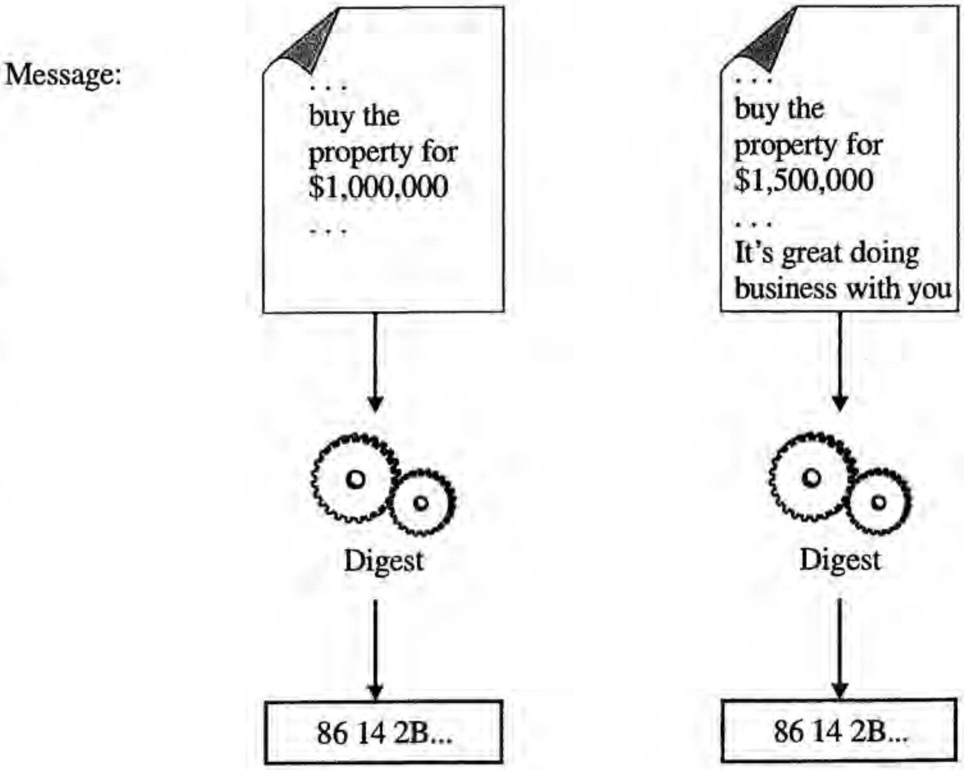
Now suppose that each message produced a different digest. (For all we know, there were messages that produced the same digest, but for the sake of argument, let's say each message produced a different digest.) You had 2^{160} pigeonholes and 2^{160} messages, each message going into a different pigeonhole. This means that all pigeonholes are now occupied. Now consider Pao-Chi's message to Daniel (ordering four presses for Satomi). This 424-bit message is not a message you've already examined. So far in this pigeonhole exercise, if you've operated on a message, it's been 160 or fewer bits. To place Pao-Chi's message into a pigeonhole, you would place it into `46 73 . . . 5a`. But that pigeonhole, like all the others, is already occupied. Which message it contains doesn't matter; you simply know it's occupied. You have a collision.

Now consider that "all possible messages" includes messages of any size.

Collisions exist, but no one can find a collision on demand (for some digest algorithms, no one has found any collision, even by accident). The worst possible scenario for a digest algorithm would be if someone could take any message and produce a similar message that produces the same digest. Figure 5-6 shows an example of that. One message mentions $1,000,000, and the second message mentions $1,500,000. If someone changes only the 5, the digests will not match. But what if someone could change the 5, change a few other things here and there, maybe add a phrase or two, and get the same digest?

Figure 5-6

If a digest algorithm were predictable enough that an attacker could change a message slightly and produce the same digest, the algorithm would be broken



The Three Important Digest Algorithms

There are many digest algorithms, but three have dominated the market: MD2, MD5, and SHA-1.

MD2

Ron Rivest created a digest algorithm and named it MD. Then he thought he could do better and so developed the next generation, MD2. Because MD2 produces a 128-bit (16-byte) digest, it has 2^{128} possible digest values. MD2 has been widely used, but over the years, analysts found flaws with it. Eventually, a few collisions were discovered. Nobody was able to find collisions on demand with any arbitrary message, but certain classes of messages produced collisions. Hence, MD2 isn't used very much anymore except on old certificates created before MD2 lost favor (Chapter 6 describes certificates). Most of those old certificates have probably expired or will expire soon. No good cryptographer would recommend using MD2 in new applications.

MD5

Rivest wanted a faster digest, and when MD2 began to show weaknesses, he also wanted one that was stronger. He started creating new digests. MD3 was a bust, and when he showed MD4 to the world it was quickly shown to be weak. (Despite that weakness, at least one application used it. See "Crypto Blunders" on the accompanying CD for that story.) MD5 was more successful.

MD5, a lot faster and much stronger than MD2, became the dominant algorithm and is still in common use. Like MD2, MD5 is a 16-byte digest. Over the years, research has led to potential weaknesses. MD5 isn't broken, and no one has found collisions; rather, some of the internals of the algorithm are vulnerable. If a component or two were missing from the algorithm, it would be broken. But because those components are there, the algorithm survives.

Some people say that it doesn't matter that the algorithm would be weak if certain pieces were missing; the pieces are there, so it's not weak. Others say that you don't break an algorithm all at once; you break it piece by piece. Now that there are only a few pieces (maybe one or two) preventing a total collapse, they argue, it would be better to move on to another algorithm.

SHA-1

The SHA-1 algorithm looks a lot like MD5 (Ron Rivest played a role in the design of SHA-1). SHA-1 contains stronger internals than MD5, and it produces a longer digest (160 bits compared with 128 bits). Size alone makes it stronger. SHA-1 has survived cryptanalysis and comes highly recommended by the crypto community. In development are SHA-1 variants that produce 192-bit and 256-bit digests.

A Representative of Larger Data

If you're looking for something to produce a representative of a larger amount of data, it's easy to see that a message digest does that job fairly well. First, the output of a digest algorithm is usually smaller than the data itself, and no matter how big the data gets, the digest as a representative will always be the same size. If someone tries to surreptitiously change the original message, the new, fake message will not produce the same digest. If the digest produced by the algorithm does not represent the data, you know that something went wrong (see Figure 5-7). Maybe the data has been altered, maybe the digest is wrong. You might not know what exactly happened, but you do know something happened.

Here's how an application can check a digest. Pao-Chi is sending Daniel some data, such as an e-mail or a contract; for this example, it's the message about selling four units to Satomi. Before Pao-Chi sends the message, he digests it. Now he sends the data and the digest. When Daniel gets the data, he also digests it. If his digest matches Pao-Chi's, he knows the data has not been changed in transit. If Satomi had intercepted and altered the message, the digest that Daniel produced would not have matched the digest Pao-Chi produced. Daniel would know that something happened and would not trust the data.

Your immediate response might be, "If Satomi could alter the data, she could alter the digest." That's true, but there are two ways to prevent that. One is to use a digital signature, a topic we'll return to shortly. For now, let's look at the second way: a keyed digest. The most common keyed digest is called HMAC.

HMAC

MAC stands for message authentication checksum (or message authentication code), and *H* stands for hash or hash-based function, so an HMAC

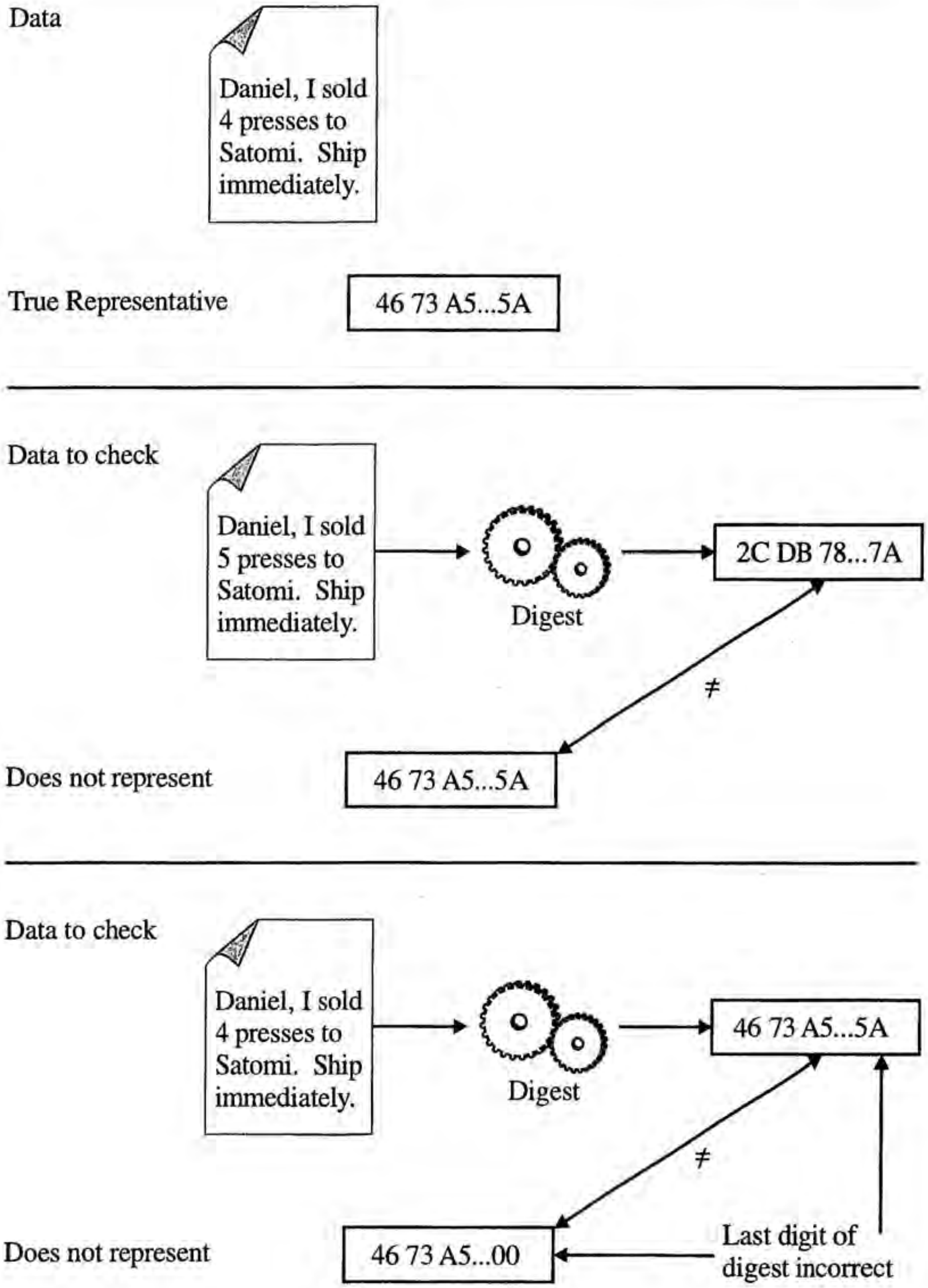


Figure 5-7

If the data does not match the digest, you know that something went wrong (pronounced “aitch mac”) is a hash-based message authentication algorithm. A *checksum* is an algorithm that checks data by summing it. Suppose you had a column of numbers (say, in an accountant’s ledger). If the correct numbers are there, the sum of the column is a specific value. Later, to check that the ledger is still correct, you don’t compare each number individually; rather, you find the sum of the column. If the second sum matches the first sum, the check passes. Of course, if someone can change one number, it’s easy also to change the sum at the bottom of the ledger so that it matches the change in the single number. It would also be easy to change another number in the column to offset the first change. A MAC is a way to detect changes in the data or in the sum. To detect changes in the data, a MAC can be based on a digest, block cipher, or stream cipher (see Chapter 2). To detect changes in the actual checksum, the MAC uses a key.

Most HMACs work this way. Two parties share a secret key (Chapter 4 shows how that’s done), and then each digest the key and message. The digest depends on the message and the key, so an attacker would have to know what the key is to alter the message and attach a correct checksum. For example, suppose Pao-Chi sends Daniel message 1 shown earlier (the message instructing him to ship four units to Satomi). Pao-Chi uses an HMAC so that Daniel can verify that the data did not change. Using a key exchange algorithm (RSA, DH, ECDH), the two agree on a 128-bit key. Pao-Chi uses SHA-1 to digest the key and message as one chunk of data. The result is as follows. (The two vertical lines | | indicate concatenation; see also Figure 5-8.)

```
Pao-Chi’s HMAC result (SHA-1 digest of key || message 1):
      60 c4 65 a8 a4 9d 35 6a 68 36 f8 f0 56 3d d2 7f
      7e 26 35 b2
```

NOTE:

We haven’t told you what the key is, so you can’t verify that the result we present is the actual result of an HMAC. If you want to know what the key is, you can figure it out. Put together a chunk of data—a key candidate followed by the message—and then digest it. Is it the same result given here? No? Try another key, and another, and so on until you find the correct one. It’s a 128-bit key.

Now Pao-Chi sends Daniel the message and the HMAC result together. Suppose that Satomi intercepts the transmission and tries to get Daniel

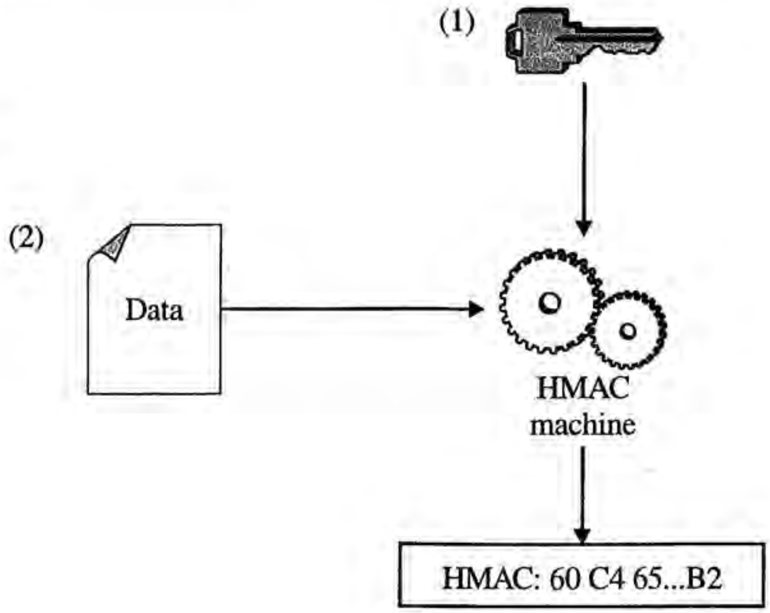


Figure 5-8

The HMAC algorithm digests the key and the data (in that order) to produce a value

to ship five presses instead of four by substituting message 2 for Pao-Chi's. After replacing the message, she sends it to Daniel. If she failed to replace the HMAC result, Daniel would digest the key and fake message and get the following (see Figure 5-9).

```
Daniel's HMAC result (SHA1 digest of key || message 2):
     a8 32 3b 8d f3 6b 3e e1 08 bb 6b 0b f0 cc a5 5b
     26 d4 d1 41
```

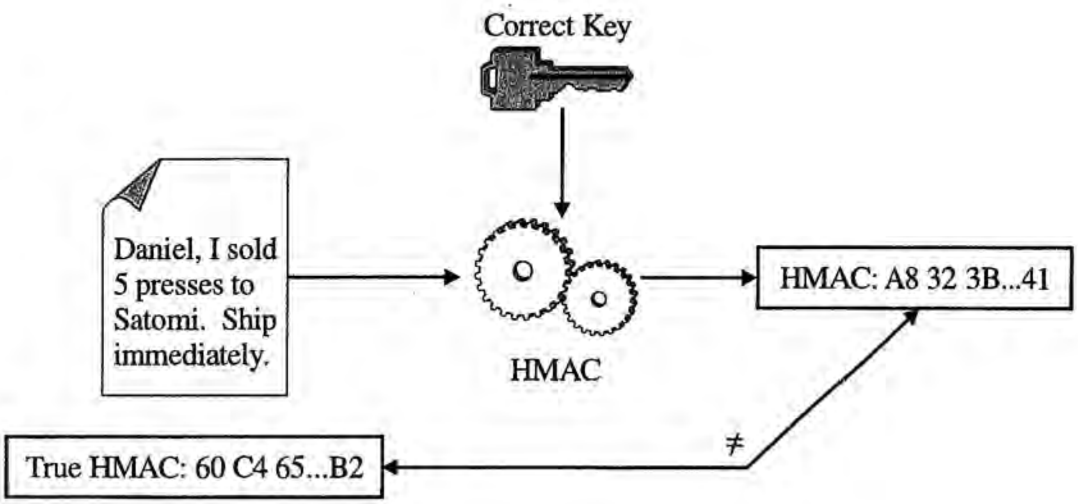


Figure 5-9

Daniel digests the correct key but the wrong message, so he knows that something is wrong

The digested message is not the same as Pao-Chi's. (Daniel knows what Pao-Chi got for an HMAC; that's part of the message.) So Daniel knows that what Pao-Chi digested and what he digested are not the same. Something—maybe the key, maybe the actual message, maybe even the HMAC value—was changed. Daniel doesn't know exactly what was changed, but that doesn't matter. He knows something went wrong. He contacts Pao-Chi again, and they start over.

Another possibility is for Satomi to substitute message 2 for message 1 *and* substitute the HMAC. But the problem is that Satomi can't know what the correct HMAC value should be. To demonstrate this, suppose Satomi substitutes six presses for four presses. Here's the SHA-1 digest.

```
   Daniel, I sold 6 presses to Satomi. Ship immediately.
SHA-1 digest:
     66 05 40 8c 24 6e 05 f8 00 20 f4 72 14 08 bc 22
     53 b2 eb d2
```

If Satomi substitutes this digest, Daniel will still know something is wrong because that's not the value he's going to get. He's not digesting the message; rather, he's digesting the key and the message. So what should Satomi use?

Data Integrity

We've described a message digest as the foundation of a pseudo-random number generator or password-based encryption, and now as a representative of a larger message. Another use for a message digest is to check *data integrity,* which is the term used to describe what the HMAC does. If you're concerned that the information may be altered, you send the data along with a check. If the message was altered, the check will also be different. Of course, you must ensure that the check value cannot be altered to match any changes in the message.

If the check value shows no alterations, the data has been shown to have integrity. "Integrity" is a word for honest, sound, and steadfast. When used in relationship to data, it may seem pretentious, but it does describe data that you can count on, at least in terms of its authenticity.

Back to Digital Signatures

In our example, the HMAC seems to serve as a signature. Daniel can know that the data came from Pao-Chi and that no one tampered with it in transit. But HMAC has some shortcomings. The first is the statement, "Daniel can know that the data came from Pao-Chi." Maybe *he* can know it came from Pao-Chi, but can anyone else? After all, to verify that the data came from Pao-Chi, the recipient must know what the key is to create the appropriate HMAC. Daniel knows what the shared secret key is, but no one else does. Daniel could write a bogus message (say, setting the number of presses to eight) and create the correct HMAC. So from anyone else's point of view, the message may have come from Pao-Chi or Daniel; no one else can know for sure who "signed" it. The second drawback is that for someone other than Pao-Chi or Daniel to verify the "signature," the correspondents must reveal the secret key. Now this third party has access to the key and can also create messages that appear genuine.

Usually, HMACs are used only to verify that contents have not been altered in transit. They are meant to be used as an on-the-fly check and not as a permanent record. For that reason, you need another way to create unique, verifiable signatures, and that way is to encrypt the digest with the signer's private RSA key.

It works like this. Pao-Chi digests the message and then encrypts the digest with his private key. He sends Daniel the message along with the encrypted digest, which serves as the signature. Daniel separates the two components and digests the message he received. He has a message in his possession and knows the digest that will produce it (he just computed it). He must determine whether the message he now has is the same message Pao-Chi sent. If Daniel knew what Pao-Chi computed as a digest, he could make that determination. Well, he has Pao-Chi's digest—it's the signature. So Daniel uses Pao-Chi's public key to decrypt the signature. That's the value Pao-Chi signed (see Figure 5-10). Is it the same answer Daniel got? If it is, he knows that the data was not altered in transit and that Pao-Chi is vouching for the contents.

Notice something powerful about the digital signature: Each chunk of data has its own signature. This means that no single digital signature is associated with an individual or key pair. Each signature is unique to the data signed and the keys used. When an individual signs two messages with the same key, the signatures will be different. Moreover, when two people with different keys sign the same data, they will produce different signatures. As a result, someone cannot take a valid signature and append

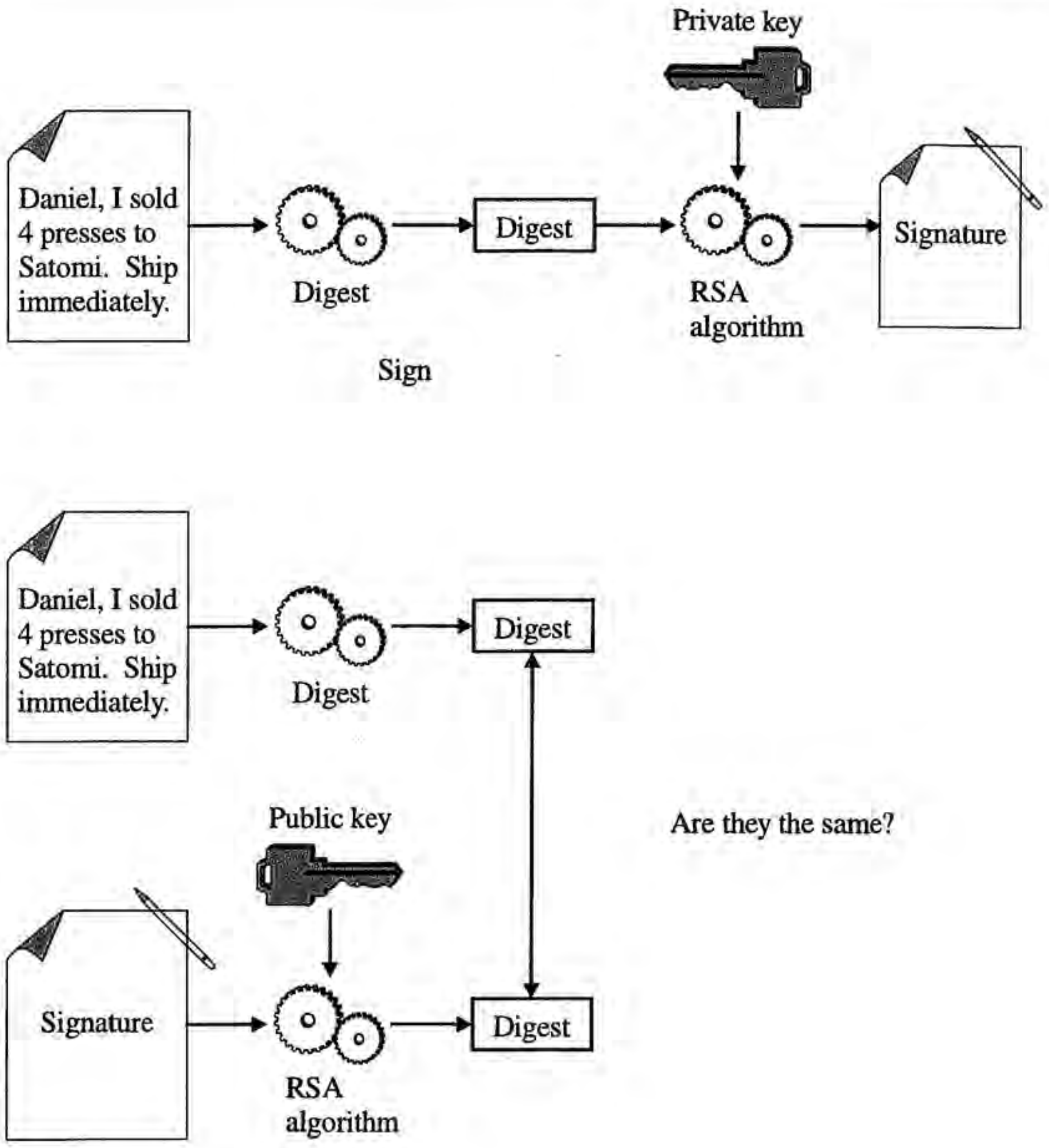


Figure 5-10
The RSA signature

it to the bottom of a different message, something that makes it much more difficult to forge a signature.

Think of it this way. Two people (a sender and a receiver) each have a copy of a message. Are they really copies or was the receiver's copy altered in transit? To find out, they digest the two messages and compare them. If the digests are the same, both parties know that the two versions match. If the digests don't match, something went wrong. How do you know that the sender's digest was not altered? You know that because it was encrypted with the sender's private key. How do you know that it was encrypted with the sender's private key? You know it because the public key decrypts it.

In addition, you can make a couple of other checks. In the real world, there will almost certainly be some digest algorithm identifier bytes (discussed in the next paragraph) and some pad bytes in addition to the digest. A signer will encrypt a block of data that is the padding, the digest algorithm identifier, and the digest. That encrypted value is the signature. Figure 5-11 shows an example. Using the appropriate public key, that signature decrypts to the padded value. The verifier checks not only for the digest but also the pad bytes and the SHA-1 algorithm identifier. (Technically, the program the verifier runs will make these checks.) Having three checks makes it harder to spoof.

The algorithm identifier bytes prevent an attacker from substituting an alternative digest algorithm. Suppose that Satomi looks at Pao-Chi's message and its correct digest. She then finds a second message and digests it using a different algorithm. Further suppose that this second algorithm on the second message produces the same digest as the first algorithm on the first message. If the signature were the encryption of the digest only, that one signature would look as if it also came from the second algorithm. But if you tie a signature to a digest *and* the algorithm, you thwart such an attack. On the one hand, it doesn't seem likely that someone would ever be able to generate the same digest from a different algorithm. On the other hand, might MD2 be broken completely someday? It doesn't cost anything to make the second check, so you might as well use it.

Trying to Cheat

Two people-Satomi and Pao-Chi-might try to cheat. Here's how they can try.

First, suppose that Satomi intercepts the message and replaces "4" with "5." She figures she'll pay Pao-Chi for four units but Daniel will send her five, and she'll get an extra press for free. In this scheme, when Daniel gets the e-mail, he digests it and gets the following value. (Using the same algorithm Pao-Chi used-SHA-1-that information is part of the e-mail although not part of the message digested.)

```
2c db 78 38 87 7e d3 1e 29 18 49 a0 61 b7 41 81
3c b6 90 7a
```

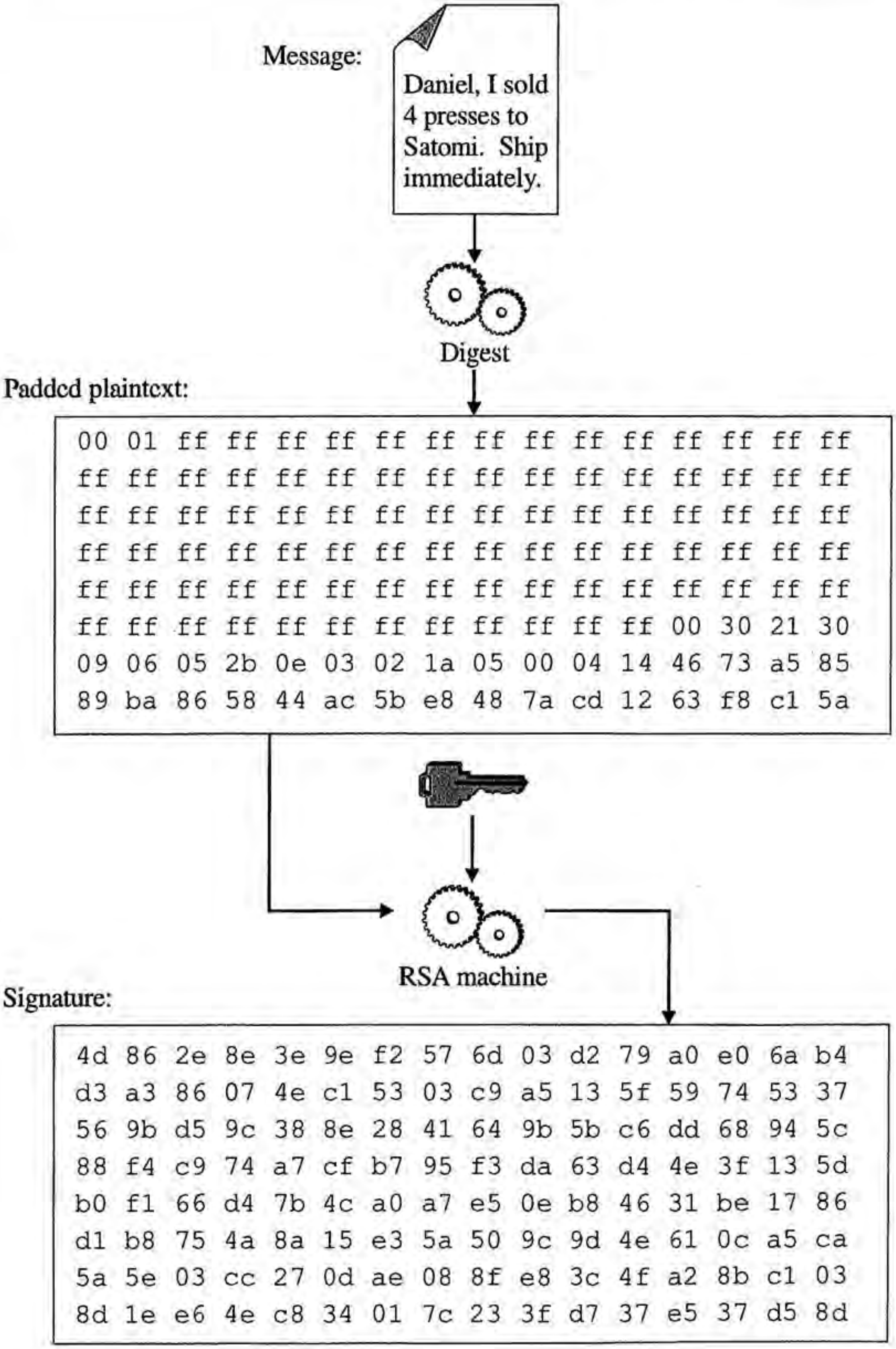


Figure 5-11

A digested message and the RSA signature. The private key used for this example is listed in Chapter 4

Daniel, I sold 4 presses to Satomi. Ship immediately.

```
00 01 ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff 00 30 21 30
09 06 05 2b 0e 03 02 1a 05 00 04 14 46 73 a5 85
89 ba 86 58 44 ac 5b e8 48 7a cd 12 63 f8 c1 5a
```

```
4d 86 2e 8e 3e 9e f2 57 6d 03 d2 79 a0 e0 6a b4
d3 a3 86 07 4e c1 53 03 c9 a5 13 5f 59 74 53 37
56 9b d5 9c 38 8e 28 41 64 9b 5b c6 dd 68 94 5c
88 f4 c9 74 a7 cf b7 95 f3 da 63 d4 4e 3f 13 5d
b0 f1 66 d4 7b 4c a0 a7 e5 0e b8 46 31 be 17 86
d1 b8 75 4a 8a 15 e3 5a 50 9c 9d 4e 61 0c a5 ca
5a 5e 03 cc 27 0d ae 08 8f e8 3c 4f a2 8b c1 03
8d 1e e6 4e c8 34 01 7c 23 3f d7 37 e5 37 d5 8d
```

Daniel must find out whether that value is the same one Pao-Chi got when he digested the message he sent. To find out, Daniel uses Pao-Chi's public key to decrypt the signature. After decryption, he gets a chunk of data. Does this data have the correct padding? He sees that the padding is correct, so he just throws that away. The next bytes are the identifying marks indicating that the algorithm is SHA-1; that's correct. Finally, he has the digest.

```
46 73 a5 85 89 ba 86 58 44 ac 5b e8 48 7a cd 12
63 f8 c1 5a
```

Daniel compares the digest value in the decrypted signature to his digest value (the value he just computed from the purported message) and sees that they are different. Something's not right. What went wrong? Daniel doesn't know exactly what caused the discrepancy, but he knows that the message he received is not the same message Pao-Chi sent. Because Daniel doesn't trust the message, he ignores it, asking Pao-Chi to try again. Meanwhile, Daniel doesn't send Satomi anything and she doesn't get her extra unit.

Now let's look at Pao-Chi's attempt at cheating. Suppose he made a mistake and quoted Satomi a price for two units. He got paid for two but told Daniel to ship four. He doesn't want to take the heat for the error, so he claims he wrote "2" instead of "4" in his e-mail. He figures he can shift the blame to Daniel or maybe just technology—some gremlin on the Internet that garbled the message.

Daniel points out that the signature attached to his e-mail matches the message with the number of presses to ship at four. Because that's Pao-Chi's signature and because each signature is unique to a message and private key, Daniel claims that Pao-Chi vouched for the information and can't back out now.

To counter this, Pao-Chi could claim that the signature was forged. To forge a signature would mean that someone was able to create a blob of data, through other means, that was the same as a signature. This would mean that some unknown forger had broken the RSA algorithm. That is highly unlikely (see Chapter 4). No, Pao-Chi signed the message, and he can't claim otherwise.

Or Pao-Chi could try another approach, claiming that someone stole his private key. Maybe it was protected on his hard drive using PBE, and someone cracked his password. Maybe it was stored on a smart card or other token, and someone broke that device or was able to log on as Pao-Chi (possibly by breaking a password). If that really is the case, Pao-Chi did a poor job of protecting his private key, and he will still be in trouble. We return to this subject later in this chapter in the section "Protecting Private Keys."

Implementing Authentication, Data Integrity, and Nonrepudiation

When Daniel checks to make sure that the data has come from Pao-Chi and not someone posing as him, it's called *authentication*. He authenticates Pao-Chi's identity. When Daniel examines the message to make sure it has not been altered in transit, that's called *data integrity checking*. And when Pao-Chi can't go back on his signature, that's called *nonrepudiation*. In addition to privacy, these are the main areas in which cryptography benefits those who use it.

Symmetric-key encryption provides privacy in that the sensitive data looks like gibberish to unauthorized eyes. Public-key technology solves the key distribution problem. A message digest—either a keyed digest such as HMAC or a digital signature—ensures data integrity in that what the sender sends is what the receiver receives. A digital signature also offers authentication in that the other entity in the data exchange is shown to be the entity it claims to be and the data is verified to have come from that entity. A digital signature also provides nonrepudiation in that a signer cannot later disavow any knowledge of the message.

Understanding the Algorithms

You can use the RSA algorithm to sign, but Diffie-Hellman can be used only to perform key exchange and not digital signatures. As discussed in Chapter 4, Diffie and Hellman proposed the idea of the ultimate public key algorithm. It would be one that could be used to encrypt data. The digital signature is the reason that such an algorithm would be the ultimate algorithm. In an interview, Whitfield Diffie explained that when he heard about the NSA's secure phone system, he was less concerned with the key exchange problem than with authentication—that is, verifying that you are talking to the person you think you are talking to.

At Stanford, cryptographer Taher El Gamal came up with a way to extend DH so that it could be used to sign as well as encrypt. But his idea never really caught on, possibly because RSA existed, and possibly because David Kravitz invented a digital signature algorithm for the U.S. government, and with the backing of an entity as powerful as the U.S. government, his algorithm became popular. Kravitz (or someone in the U.S. government) gave the new algorithm the lyrical name "Digital Signature Algorithm," known to this day as DSA. Like DH, DSA is based on the discrete log problem. It became the official U.S. government signature algorithm and probably is second only to RSA in use today. Kravitz was working for the NSA when he developed DSA, and it is based on work by El Gamal and Claus Schnorr, another cryptographer.

Finally, just as elliptic curve math can be adapted to solve the key distribution problem, it can be adapted to create signatures. There are a number of possibilities, but the most common way to use ECC to create signatures is called ECDSA. This approach does essentially the same thing as DSA but with elliptic curves.

NOTE:

Kravitz received a patent for DSA, but the U.S. government owns it because the inventor was working for the NSA at the time. The patent is in the public domain and can be used freely. Claus Schnorr invented a signature algorithm that is very similar to DSA. His patent on that algorithm predates Kravitz's. If you want the whole story, consult a patent attorney.

Many signature algorithms have been proposed over the years, but only RSA, DSA, and ECDSA have shown any long-lasting success in finding adopters. Let's look at these three algorithms in more detail.

RSA

We show RSA in detail in Chapter 4. It's the algorithm that is used to encrypt a digest with a private key to produce a digital signature. To forge an RSA signature, someone must find the private key. Lacking a private key, no one has been able to produce a chunk of data, call it a digital signature, and have it be verified.

DSA

To this point, we've described a digital signature as the private-key encryption of a digest. Now we come to DSA, which does not encrypt data. Although DSA uses the digest of the data, it does not encrypt the digest. Your first thought is likely to be, "If it can't encrypt data, how can it produce a digital signature?" Remember that DH cannot be used to encrypt data but can be used to solve the key distribution problem. Similarly, even though DSA cannot be used to encrypt data, it can be used to create a digital signature. A digital signature is a chunk of data that comes from the message and the private key. Only that particular message coupled with that particular private key will produce that particular signature. If you accomplish that by encrypting the digest, great. If you accomplish that in some other way, that's fine, too.

With DSA, the signer digests the message with SHA-1 and treats that digest as a number (it's a big number: 160 bits long). Another number sent to the algorithm is a random or pseudo-random value, usually called k. The last input is the private key. The algorithm then performs some mathematical operations, one of which is modular exponentiation, the same function at the heart of DH and RSA. The output is two numbers, usually called r and s. These two numbers are the signature.

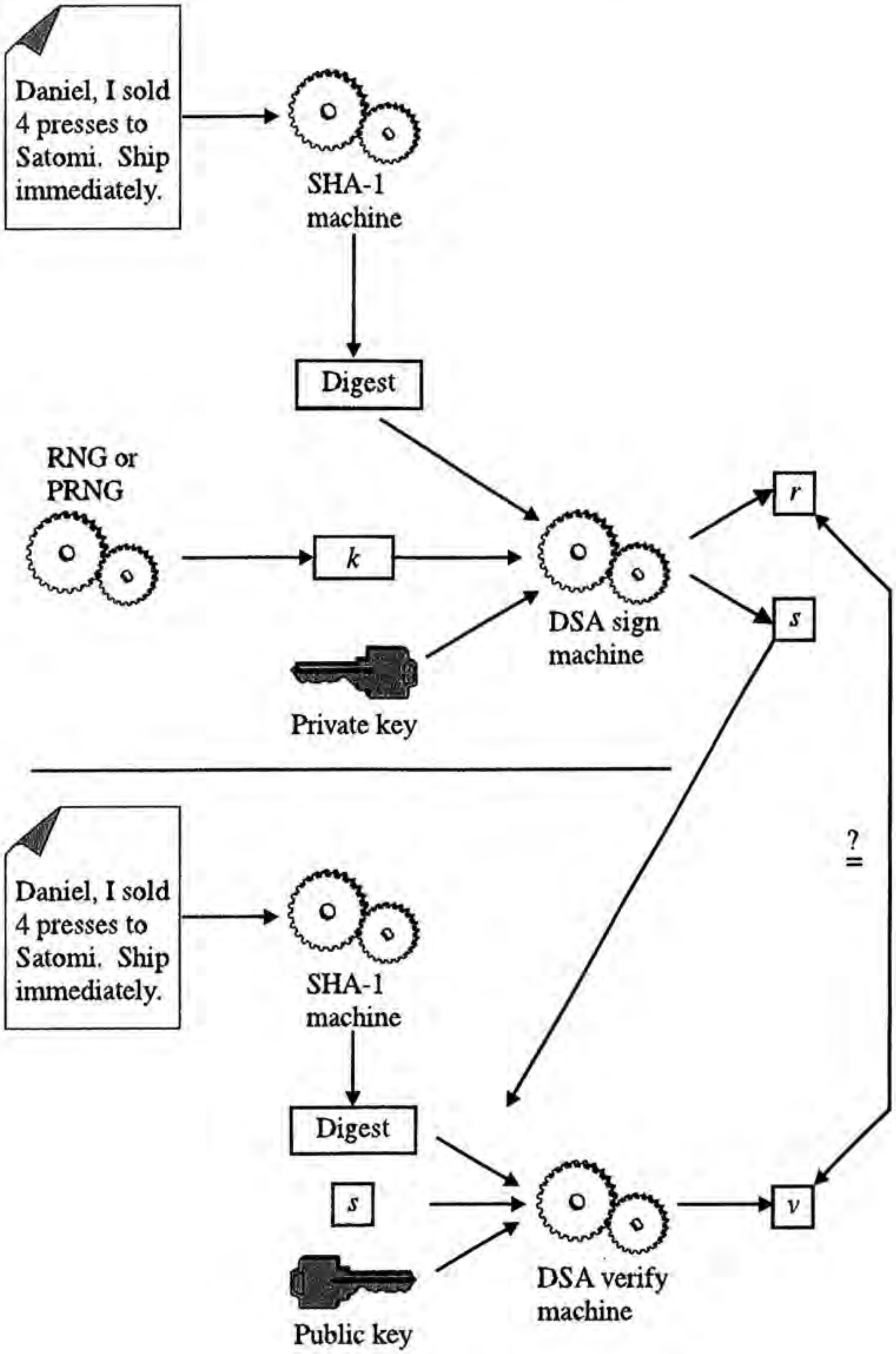
The verifier computes the SHA-1 digest of the message. Is it the same digest that the signer produced? The verifier does not have that digest available but does have r and s. Using the digest as a number, along with the public key and the s, the verifier performs some mathematical operations. The result of the computations is a number called v. If v is the same as r, the signature is verified (see Figure 5-12).

At its most basic, DSA computes the same number in two different ways. In Diffie-Hellman, two parties can generate the same secret value even though each one is using different input. The same thing is happening here with DSA. Two parties produce the same number using different input. The two sets of input are related. Well, they should be related; if something breaks down, the final answers will differ.

Each side has three inputs. The signer has the digest, k, and the private key. The verifier has the digest, s, and the public key. The digests are related; they should be the same thing. If that relationship breaks down-say, the signed data is not the same as the data being verified and the two parties produce different digests—the final answer from each individual will differ. The k and s are related (they're not the same number, but they're related). If the signature is wrong, the s will be wrong and the two

Figure 5-12

Producing and verifying a DSA signature



players will produce different final answers. The private key and the public key are also related; they are partners related mathematically. If that relationship is not there—if the public key used to verify is not the partner to the private key used to sign—the two agents will produce different final answers.

The security of DSA lies in the discrete log problem, the same problem that gives DH its security. So the size of DSA keys will be the same as that of DH keys. As always, you can find more detailed information in the RSA Labs FAQ on the accompanying CD.

ECDSA

This algorithm looks a lot like DSA. The signer has three inputs: the digest, *k*, and the private key. The output is *r* and *s*. The verifier has the digest, *s*, and the public key. The output is *v*. If *v* and *r* are the same, the signature is verified; if they're not the same, something went wrong. What went wrong? Was it the wrong digest? The wrong public key? Was the signature mangled in transmission? You probably can't know exactly what happened, but you do know that something went wrong. The math underlying ECDSA is elliptic curves, so key size is the same as with ECDH.

Comparing the Algorithms

Of the three algorithms that produce digital signatures, which one is the best? As we say in Chapter 4 regarding the key distribution problem, there's probably no single answer to that question. Each has its advantages and disadvantages. A more appropriate question might be, "Which algorithm works best in which situation?" Remember that all three of them are in use today because different problems call for different solutions.

Security

Everything we say in Chapter 4 on the security of the three algorithms applies here as well (the security of Diffie-Hellman and DSA are pretty much the same). There's no objective answer to the question of which algorithm is the most secure. It depends on what each individual feels is important.

One other factor with digital signatures, though, may be the concept of *message recovery*. With RSA, a signature verification recovers the message, but with DSA and ECDSA, a signature verification simply compares two numbers. Technically, RSA recovers the digest of the message instead of the message itself; that's really one level of indirection. DSA and ECDSA find a number based on the digest; that's two levels of indirection. Earlier in this chapter in "The Uniqueness of a Digital Signature," we mention that the crypto literature on digital signatures contains statements such as, "For some classes of signatures it is possible to prove certain security properties." Message recovery is one of those security properties. When you perform an RSA verification operation, you get to see what the signer produced; you recover the message digest because you're decrypting it. With DSA and ECDSA, you don't see what the signer produced. Instead, you generate a number, and if that number is equal to another number, you figure you produced the same thing that the signer produced.

Think of it this way. DSA and ECDSA produce surrogate numbers, let's call them the signer's surrogate and the verifier's surrogate. If the two numbers match, the signature is verified. With RSA, there is no surrogate; the verifier actually compares the signer's value.

Because DSA and ECDSA compare surrogates and not originals, it opens an avenue of attack not possible with RSA. An attacker could try to produce the appropriate surrogate number without the correct original key or data. That is, an attacker does not have to find a digest collision to substitute messages, but can try to find a DSA collision. But before you think that makes RSA much stronger than the other two, remember that no one has been able to create such an attack or even to come close. Still, although the probability of such an attack on DSA or ECDSA is extremely low, it's lower still with RSA.

Performance

In Chapter 4, you saw that no algorithm wins the performance race hands-down. Of the several factors, each algorithm compares favorably with the others in one way but unfavorably in another. The same is true with signatures. RSA performance does not change, but DSA and ECDSA are slightly more time-consuming than their DH counterparts.

If you want a faster signature scheme, you should go with ECC. But often, making a connection means that each party has to do two or more verifications; each one must verify a signature and then verify one or more certificates (Chapter 6 talks about certificates). If you have a fast signer (a server, for example) but a slow verifier (a hand-held device or smart card for example), you may get bogged down in verification. Again, each application may have different needs, and even though one algorithm may satisfy one application's needs better than another algorithm, the next application may find a different algorithm more suitable.

Table 5-1 shows some performance comparisons. The numbers are relative; if RSA public-key operations (such as verification) take one unit of time (whatever that unit may be) on a particular machine, the other operations will take the amounts of time shown.

Table 5-1

Estimated Relative Performance of the Public-Key Algorithms (in Relative Time Units)

	RSA	DSA	ECC	ECC with Acceleration
Private key (sign)	13	17	7	2
Public key (verify)	1	33	19	
Combined	14	50	26	21

Transmission Size

DSA and ECDSA signatures are about 340 bits, regardless of key size. An RSA signature is the same size as the key. So if you use a 1,024-bit RSA key pair, each time you send a digital signature you add 1,024 bits to the message. Again, if transmission size is important, you may want to look at DSA or ECDSA.

Interoperability

The story's the same with signatures as with key distribution. RSA is almost ubiquitous and has become the de facto standard. DSA was promoted by the U.S. government and has become a part of most cryptographic packages. So if you sign using RSA or DSA, other parties will almost certainly be able to verify it, whether or not they use the same application you do. ECC is less prevalent.

Protecting Private Keys

Chapter 3 shows how to protect symmetric keys, and Chapter 4 explains that you protect a private key in a similar way. Tokens such as smart cards add a dimension to protection, but for the most part, the way you protect one key is the way you protect any key. Many protocols (discussed in Chapters 7 and 8) require that you have two keys: a digital envelope (or key exchange key) and a separate signing key. So you'll likely have to protect two private keys.

But if you lose your private key, there are ways to *revoke,* or cancel, the public key affiliated with it. If Pao-Chi claims that someone obtained his private key and is signing under his name, he can have his public key revoked. After the effective date of the revocation, any signatures verified with Pao-Chi's public key are invalid because the public key is invalid. Now Pao-Chi has to generate a new key pair, this time protecting the private key more diligently. Chapter 6 talks about revoking keys.

For now, note that if attackers steal your signing key, they can do a lot more damage than if they steal other types of keys because your signing key lets them pose as you. By stealing your digital envelope or key exchange private key, attackers can get at secrets, but they cannot act on your behalf. If you don't protect your signing key or don't protect it well enough, you're making yourself much more vulnerable.

Introduction to Certificates

Throughout Chapters 4 and 5, we've talked about other individuals using someone else's public key. To send a secure message to Gwen, Pao-Chi found her public key and created a digital envelope. To verify Pao-Chi's message, Daniel acquired Pao-Chi's public key and verified the digital signature. But how can anyone truly know whether a public key belongs to the purported individual?

Pao-Chi has in his possession a public key, which is purportedly Gwen's. The key works; he is able to create a digital envelope. But what if

Satomi somehow substituted her public key for Gwen's? While Pao-Chi was out to lunch, Satomi may have broken into his laptop, found a file called "Gwen's public key" and edited it so that this file contained her public key, not Gwen's. Then when Pao-Chi sends the digital envelope, Satomi will be able to intercept and read it. Gwen won't be able to open it because she does not have access to the private key partner to the public key used.

Suppose the company Pao-Chi and Daniel work for has a centralized directory where everyone's public key is stored. When Daniel wants to verify Pao-Chi's signature, he goes to the directory and find's Pao-Chi's key. But what if Satomi broke into that directory and replaced Pao-Chi's public key with hers? Now she can send a fake message to Daniel with a valid digital signature. Daniel will think it came from Pao-Chi because he verifies the signature against what he thinks is Pao-Chi's public key.

The most common way to know whether or not a public key does belong to the purported entity is through a digital certificate. A *digital certificate* binds a name to a public key. An analogy would be a passport, which binds a photo to a name and number. A passport is supposed to be produced in such a way that it is detectable if someone takes an existing passport and replaces the true photo with an imposter's photo. It may be a valid passport, but not for the person in the photo. Immigration officials will not honor that passport.

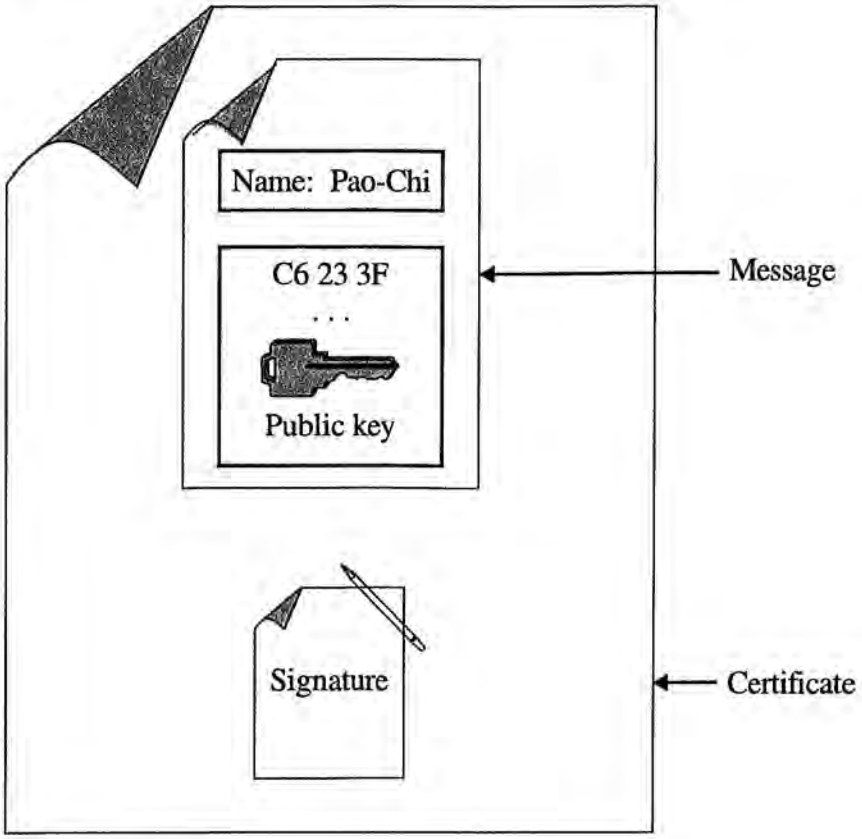
A digital certificate is produced in such a way that it is detectable if someone takes an existing certificate and replaces the public key or name with an imposter's. Anyone examining that certificate will know that something is wrong. Maybe the name or public key is wrong , so you don't trust that name/key pair combination.

Here's how it works. Take a name and public key. Consider those two things to be a message, and sign the message. The certificate is the name, public key, and signature (see Figure 5-13). The only thing left to determine is who will sign the certificate. Signing is almost always done by a *certificate authority*, also known as a CA. More on that later.

Gwen originally generated her key pair, protected the private key, and contacted her CA requesting a certificate. Depending on the CA's policy, Gwen may be required to show up in person. The CA verifies Gwen is who she claims to be by examining her passport, driver's license, company ID badge, or whatever method the CA uses to determine identity. Then Gwen uses her private key to sign something (the certificate request, probably). In that way, the CA knows that Gwen does indeed have access to the private key partner to the public key presented, and that the public key has not been replaced. The CA combines Gwen's name and public key into a

Figure 5-13

A certificate is the name, public key and signature



message and signs that message using its private key. Gwen now has a certificate and distributes it. So when Pao-Chi collects Gwen's public key, what he's really collecting is her certificate.

Now suppose Satomi tries to replace Gwen's public key with her own. She finds the file on Pao-Chi's laptop holding Gwen's public key and substitutes the keys. But when Pao-Chi loads the key, he's not loading just the key, he's loading the certificate. He can extract the public key from the certificate if he wants, but before he does that, he verifies that the certificate is valid using the CA's public key. Because the message has been altered, the signature does not verify and Pao-Chi does not trust that public key. Therefore, he will not create a digital envelope using that public key, and Satomi will not be able to read any private communications.

Of course, that scenario assumes that Pao-Chi has the CA's public key and that he can trust no one has replaced it with an imposter's. Because he can extract it from the CA's certificate, Pao-Chi knows he has the true CA public key. Just as Gwen's public key can be wrapped in a certificate, so can the CA's. Who signed the CA's certificate? Probably another CA. This could go on forever.

But it has to stop somewhere and that somewhere is the *root*. A root will sign a CA's certificate, and the root key is distributed outside the certificate hierarchy. Maybe the root is built into software; maybe Pao-Chi will have to enter it himself. Of course, if Satomi is able to substitute the root public key with one of her own, she can subvert the whole system. So Pao-Chi needs to protect the root key as he does his symmetric key and his own private keys.

Key Recovery

As discussed in Chapter 4, it's possible to set up a scheme to restore keys that someone loses by forgetting a password or losing a token. However, it's probably not a good idea to apply a key recovery plan to signing keys. If a signing key can be obtained by someone other than the owner (even if that is a trusted third party or a committee of trustees), that would make it possible to nullify nonrepudiation. Anyway, if someone loses a signing key, it's no great problem; any existing signatures are still valid because only the public key is needed to verify. For new signatures, you simply generate a new key pair and distribute the new public key. For this reason, many protocols specify that participants have separate signing and key exchange keys. As you will see in Chapters 6 and 7, it is possible to define a key as signing only or key encrypting (with the RSA digital envelope) or as key exchange only (with the Diffie-Hellman protocol).

Summary

To verify that a message came from the purported sender, you can use public-key cryptography. A private key is used to sign the data, and the public key is used to verify it. The only known way to produce a valid signature is to use the private key. Also, a signature is unique to a message; each message and private key combination will produce a different signature. So if a public key verifies a message, it must be that the associated private key signed that message. Three main algorithms are used as signature schemes: RSA, DSA, and ECDSA. Each algorithm has its advantages and disadvantages, and it's not really possible to say that one or the other is better. Each algorithm may be better suited for different applications.

For performance reasons, you don't sign the data but rather sign a representative of the data called a message digest. Also known as a hash, a message digest is the foundation of most PRNGs and PBE implementations. A keyed digest, such as HMAC, is also used to check data integrity.

Real-World Example

As discussed in Chapter 4, S/MIME uses public-key cryptography to solve the key distribution problem. As you've probably already surmised, S/MIME uses digital signatures as well. To implement a digital signature, follow these steps. First, launch Netscape Navigator, click the Security button, and then click Messenger (along the left-hand side of the security window). In Chapter 4, you saw the Encrypt choice checked. Notice the two Sign choices as well. If you select these menu items, when you send e-mail or post to newsgroups your message will be signed using your private key.

Before you can sign, you need a key pair. The browser has a module that generates a key pair for you, or, if you have a token (such as a smart card), you can specify that it be used to generate the key pair and store the private key. In the security window, click Yours under Certificates. The resulting window displays a button labeled Get A Certificate. This is the starting point for generating a key pair and getting a certificate. (Chapter 6 discusses certificates.)

With Microsoft Outlook, click Tools and then Options. In the resulting window, click the Security tab. You saw the Encrypt choice in Chapter 4. Here, notice the Sign option. Again, you need a key pair and a certificate. Start the process by clicking the Get A Digital ID button at the bottom of the window.

CHAPTER 6

Public-Key Infrastructures and the X.509 Standard

As you learned in Chapter 4, public-key cryptography gives you not only a powerful mechanism for encryption but also a way to identify and authenticate other individuals and devices. Before you can use this technology effectively, however, you must deal with one drawback. Just as with symmetric-key cryptography, key management and distribution are an issue with public-key crypto. Instead of confidentiality, the paramount issue for public-key crypto is the integrity and ownership of a public key.

For end users and *relying parties* (relying parties are those who verify the authenticity of an end user's certificate) to use this technology, they must provide their public keys to one another. The problem is that, like any other data, a public key is susceptible to manipulation while it is in transit. If an unknown third party can substitute a different public key for the valid one, the attacker could forge digital signatures and allow encrypted messages to be disclosed to unintended parties. That's why it's crucial to assure users that the key is authentic and that it came from (or was received by) the intended party.

Within a small population of trusted users, this task is not very difficult. An end user could distribute the public key by simply hand-delivering it on disk to a recipient, an approach known as manual public-key distribution. For larger groups of individuals, however, this task is much more difficult, especially when the people are geographically dispersed. Manual distribution becomes impractical and leaves room for security holes. For that reason, a better solution has been developed: public-key certificates. Public-key certificates provide a systematic, scalable, uniform, and easily controllable approach to public-key distribution.

A *public-key certificate* (PKC) is a tamperproof set of data that attests to the binding of a public key to an end user. To provide this binding, a set of trusted third parties vouches for the user's identity. The third parties, called *certification authorities* (CAs), issue certificates to the user that contain the user's name, public key, and other identifying information. Digitally signed by the CA, these certificates can now be transferred and stored.

This chapter covers the necessary technology needed to understand and use a *public-key infrastructure* (PKI). First, we describe the X.509 standard and the structure of an X.509 public-key certificate. Then we explain how the PKI components work as a collaborative process to let you create, distribute, manage, and revoke certificates.

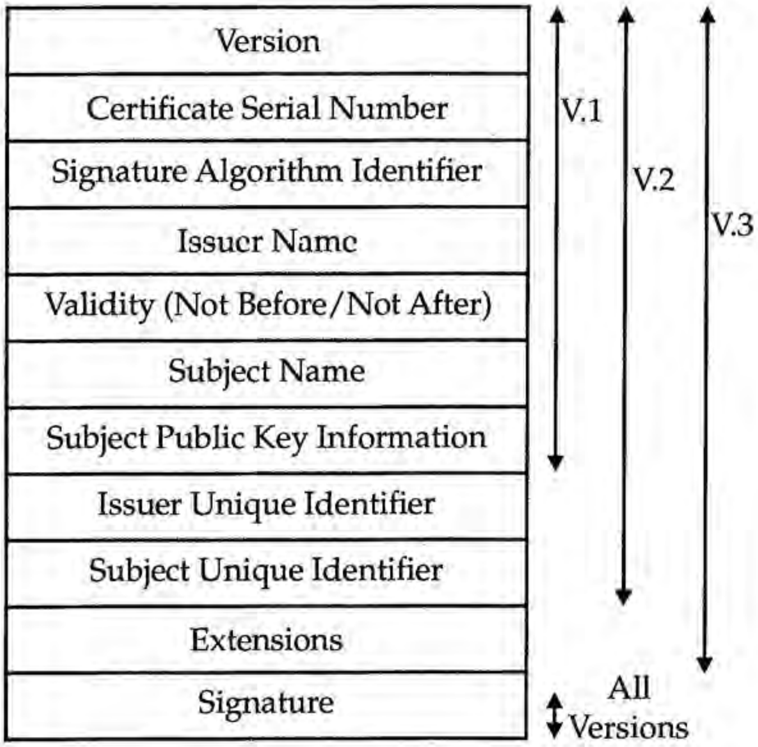
Public-Key Certificates

Public-key certificates are a secure means of distributing public keys to relying parties within a network. In many ways, PKCs are analogous to a driver's license. Both a driver's license and a PKC are certified by a trusted third party, which affirms the user's identity and privileges. In its most basic form, a certificate contains a public key, the identity of the individual it belongs to, and the name of the party that is attesting to the validity of these facts.

Various certificates are in use. Some of them, such as *Pretty Good Privacy* (PGP), are proprietary. Other popular certificates are application-specific, such as SET and *Internet Protocol Security* (IPSec) certificates. The most widely accepted certificate format is the International Telecommunication Union's X.509 Version 3. The original X.509 standard was published in 1988 as part of the X.500 directory recommendations. Since then, it has been revised twice—in 1993 and again in 1995. RFC2459, a profile for the X.509 standard, was published in 1999 by the *Internet Engineering Task Force* (IETF). Although RFC2459 is targeted to the Internet community, a number of its useful components can be applied in an enterprise environment. Therefore, we provide references to some of its recommendations where appropriate. Figure 6-1 illustrates the structure of an X.509 certificate.

Figure 6-1

X.509 certificate structure



All versions of X.509 certificates contain the following fields:

- **Version** This field differentiates among successive versions of the certificate, such as Version 1, Version 2, and Version 3. The Version field also allows for possible future versions.
- **Certificate Serial Number** This field contains an integer value unique to each certificate; it is generated by the CA.
- **Signature Algorithm Identifier** This field indicates the identifier of the algorithm used to sign the certificate along with any associated parameters.
- **Issuer Name** This field identifies the *distinguished name* (DN) of the CA that created and signed this certificate.

- **Validity (Not Before/After)** This field contains two date/time values—Not Valid Before and Not Valid After—which define the period that this certificate can be considered valid unless otherwise revoked. The entry can use the following formats: UTC time (*yymmddhhmmssz*) or generalized time (*yyyymmddhhmmssz*).
- **Subject Name** This field identifies the DN of the *end entity* to whom this certificate refers, that is, the subject who holds the corresponding private key. This field must have an entry unless an alternative name is used in the Version 3 extensions.
- **Subject Public Key Information** This field contains the value of the subject's public key as well as the algorithm identifier and any associated parameters of the algorithm for which this key is used. This field must always have an entry.

Unique Identifiers

Version 2 and 3 certificates may contain unique identifiers that pertain to the subject and issuer. These fields are designed to handle the possibility of reuse of these names over time. RFC2459 recommends that names not be reused for different entities and that Internet certificates not use unique identifiers. This means that CAs conforming to this profile *should not* generate certificates with unique identifiers. Nevertheless, applications conforming to this profile should be capable of parsing unique identifiers and making comparisons.

- **Issuer Unique Identifier** This optional field contains a unique identifier that is used to render unambiguous the X.500 name of the CA in cases when the same name has been reused by different entities over time. This field can be used only in Version 2 and Version 3 certificates, and its use is not recommended according to RFC2459.
- **Subject Unique Identifier** This optional field contains a unique identifier that is used to render unambiguous the X.500 name of the certificate owner when the same name has been reused by different entities over time. This field can be used only in Version 2 and Version 3 certificates, and its use is not recommended according to RFC2459.

Standard Version 3 Certificate Extensions

After the release of Version 2, it was apparent that the certificate profile still contained deficiencies. For this reason, a set of extensions was created to append to the Version 3 format of the certificate. These extensions cover key and policy information, subject and issuer attributes, and certification path constraints.

The information contained in extension fields can be marked as either critical or noncritical. An extension field has three parts: extension type, extension criticality, and extension value. The *extension criticality* tells a certificate-using application whether it can ignore an extension type. If this extension is set to critical and the application does not recognize the extension type, the application should reject the certificate. On the other hand, if the extension criticality is set to noncritical and the application does not recognize the extension type, it is safe for the application to ignore the extension and to use the certificate.

The following standard certificate extension fields are available only in Version 3 certificates:

- **Authority Key Identifier** This extension is used to differentiate between multiple certificate signing keys of the same CA. The CA provides a unique key identifier or provides a pointer to another certificate, which can certify the issuer's key. The RFC2459 mandates the use of this field for any certificate that is not self-signed.
- **Subject Key Identifier** This extension is used to differentiate between multiple certificate signing keys of the same certificate owner. The owner provides a unique key identifier or provides a pointer to another certificate that can certify the issuer's key. RFC2459 mandates the use of this field for any CA signing certificate and also recommends it for end entities.
- **Key Usage** This extension is used to define restrictions on the operations that can be performed by the public key within this certificate. Such operations include digital signature, certificate signing, *certificate revocation list* (CRL) signing, key enciphering, data enciphering, and Diffie-Hellman key agreement. This field can also be flagged as critical or noncritical. If it is flagged critical, it can be used only for its intended use; otherwise, it will be considered in violation of the CA's policy. RFC2459 recommends a flag of critical when this field is used.

- **Extended Key Usage** This extension can be used in addition to or in place of the Key Usage extension to define one or more uses of the public key that is certified within this certificate. This extension enables the certificate to interoperate with various protocols and applications (such as, *Transport Layer Security* [TLS] server authentication, client authentication, time stamping, and others). RFC2459 states that this field may be flagged critical or noncritical.
- **CRL Distribution Point** This extension indicates a *uniform resource identifier* (URI) to locate the CRL structure where revocation information associated with this certificate resides. RFC2459 recommends that this field be flagged noncritical, although it also recommends that CAs and applications support this extension.
- **Private Key Usage Period** Similar to the Validity field of the certificate, this extension indicates the time frame of use for the private key associated with the public key in this certificate. In the absence of this extension, the validity period of use for the private key is that of the associated public key. RFC2459 recommends against the use of this extension.
- **Certificate Policies** This extension identifies the policies and optional qualifier information that the CA associates with the certificate. If this extension is marked critical, the processing application must adhere to at least one of the policies indicated, or the certificate is not to be used. To promote interoperability, RFC2459 recommends against the use of policy identifiers, but it does specify two possible qualifiers: the *certification practice statement* (CPS) qualifier and the user notice qualifier. The CPS qualifier contains a pointer to a CPS that applies to this certificate. The notice reference qualifier can be made up of a notice reference or an explicit notice (or both), which can in turn provide a text message of the policy required for this certificate.
- **Policy Mappings** This extension is used only when the subject of the certificate is also a CA. It indicates one or more policy *object identifiers* (OIDs) within the issuing CA's domain that are considered to be equivalent to another policy within the subject CA's domain.
- **Subject Alternative Name** This extension indicates one or more alternative name forms associated with the owner of this certificate. Use of this field enables support within various applications that employ their own name forms, such as various e-mail products, *electronic data interchange* (EDI), and IPSec. RFC2459 specifies that

if no DN is specified in the subject field of a certificate, it must have one or more alternative names and this extension must be flagged critical.

- **Issuer Alternative Name** This extension indicates one or more alternative name forms associated with the issuer of this certificate. As with the Subject Alternative Name extension, use of this field enables support within various applications.
- **Subject Directory Attributes** This extension can be used to convey any X.500 directory attribute values for the subject of this certificate. It provides additional identifying information about the subject that is not conveyed in the name fields (that is, the subject's phone number or position within a company). RFC2459 recommends against the use of this extension at this time. However, if it is used, RFC2459 mandates the use of a noncritical flag to maintain interoperability.
- **Basic Constraints** This extension indicates whether the subject may act as a CA, providing a way to restrict end users from acting as CAs. If this field is present, a certification path length may also be specified. The certification path length limits the certifying powers of the new authority (for example, whether Verisign could allow RSA Inc. to act as a CA but at the same time not allow RSA Inc. to create new CAs). RFC2459 mandates that this extension be present and marked critical for all CA certificates.
- **Name Constraints** This extension, to be used only within CA certificates, specifies the namespace within which all subject names must be located for any subsequent certificate that is part of this certificate path. RFC2459 mandates that this extension be marked critical.
- **Policy Constraints** This extension, to be used only within CA certificates, specifies policy path validation by requiring policy identifiers or prohibiting policy mappings (or both). RFC2459 simply states that this extension may be marked critical or noncritical.

Entity Names

In a public-key certificate, entity names for both the issuer and the subject must be unique. Version 1 and 2 certificates use the X.500 DN naming convention.

Distinguished names were originally intended to identify entities within an X.500 directory tree. A *relative distinguished name* (RDN) is the path from one node to a subordinate node. The entire DN traverses a path from the root of the tree to an end node that represents a particular entity. A goal of the directory is to provide an infrastructure to uniquely name every communications entity everywhere (hence the "distinguished" in "distinguished name"). As a result of the directory's goals, names in X.509 certificates are perhaps more complex than one might like (compared with, for example, e-mail addresses). Nevertheless, for business applications, DNs are worth the complexity because they are closely coupled with legal name registration procedures, something not offered by simple names such as e-mail addresses. A distinguished name is composed of one or more RDNs, and each RDN is composed of one or more *attribute-value assertions* (AVAs). Each AVA consists of an attribute identifier and its corresponding value information, for example, "CountryName = US" or "CommonName = Jeff Hamilton".

X.509 Version 3 certificates grant greater flexibility with names, no longer restricting us solely to X.500 name forms. Entities can be identified by one or more names using various name forms. The following name forms are recognized by the X.509 standard:

- Internet e-mail address
- Internet domain name (any official DNS name)
- X.400 e-mail address
- X.500 directory name
- EDI party name
- Web URI, of which a *uniform resource locator* (URL) is a subtype
- Internet IP address (for use in associating public-key pairs with Internet connection endpoints).

Alternative names provide more flexibility to relying parties and applications that may not have any connections to the end user's X.500 directory. For example, a standard e-mail application could use a certificate that provides not only an X.500 name form but also a standard e-mail address.

ASN.1 Notation and Encoding

Most encrypted data ends up being transferred to other entities, so it is crucial that the data follow a standard format, syntax, and encoding so that it makes sense to other users or applications. We've talked about how the X.509 standard provides such a format. In this section we explain the X.509 rules for data syntax and encoding.

The syntax for all certificates that conform to the X.509 standard are expressed using a special notation known as *Abstract Syntax Notation 1* (ASN.1), which was originally created by *Open Systems Interconnection* (OSI) for use with various X.500 protocols. ASN.1 describes the syntax for various data structures, providing well-defined primitive objects as well as a means to define complex combinations of those primitives.

ASN.1 has two sets of rules that govern encoding. *Basic Encoding Rules* (BER, defined in X.690) are a way of representing ASN.1-specified objects as strings of 1's and 0's. *Distinguished Encoding Rules* (DER), a subset of BER, provide a means to uniquely encode each ASN.1 value.

NOTE:

For more information about these rules, see Appendix B, which includes a copy of RSA Laboratories' "A Layman's Guide to a Subset of ASN.1, BER, and DER."

The Components of a PKI

As we've mentioned, CAs serve as trusted third parties to bind an individual's identity to his or her public key. CAs issue certificates that contain the user's name, public key, and other identifying information. Signed by the CA, these certificates are stored in public directories and can be retrieved to verify signatures or encrypt documents. A public-key infrastructure involves a collaborative process between several entities: the CA, a *registration authority* (RA), a certificate repository, a key recovery server, and the end user. In this section we discuss each of these components in detail.

Certification Authority

If we think of a certificate as being similar to a driver's license, the CA operates as a kind of licensing bureau analogous to a state's Department of Motor Vehicles or similar agency. In a PKI, a CA issues, manages, and revokes certificates for a community of end users. The CA takes on the tasks of authenticating its end users and then digitally signing the certificate information before disseminating it. The CA is ultimately responsible for the authenticity of its end users.

In providing these services, the CA must provide its own public key to all the certified end users as well as all relying parties who may use the certified information. Like end users, the CA provides its public key in the form of a *digitally signed* certificate. However, the CA's certificate is slightly different in that the Subject and Issuer fields contain the same information. Thus, CA certificates are considered *self-signed.*

CAs fall into to two categories: public and private. *Public* CAs operate via the Internet, providing certification services to the general public. These CAs certify not only users but also organizations. *Private* CAs, on the other hand, are usually found within a corporation or other closed network. These CAs tend to license only to end users within their own population, providing their network with stronger authentication and access controls.

Registration Authority

Although an RA can be considered an extended component of a PKI, administrators are discovering that it is a necessity. As the number of end entities increases within a given PKI community, so does the workload placed on a CA. An RA can serve as an intermediate entity between the CA and its end users, assisting the CA in its day-to-day certificate-processing functions.

An RA commonly provides these functions:

- Accepting and verifying registration information about new registers
- Generating keys on behalf of end users
- Accepting and authorizing requests for key backup and recovery
- Accepting and authorizing requests for certificate revocation
- Distributing or recovering hardware devices, such as tokens, as needed

RAs are also commonly used for the convenience of end users. As the number of end users increases within a PKI domain, it's likely that they will become more geographically dispersed. CAs can delegate the authority to accept registration information to a local RA. In this way, the CA can be operated as an offline entity, making it less susceptible to attacks by outsiders.

Certificate Directory

After a certificate is generated, it must be stored for later use. To relieve end users of the need to store the certificate on local machines, CAs often use a *certificate directory,* or central storage location. An important component of a PKI, a certificate directory provides a single point for certificate administration and distribution. There is no one required directory standard. Lotus Notes and Microsoft Exchange use proprietary directories, and directories based on the X.500 standard are also gaining popularity.

X.500 directories are becoming more widely accepted because in addition to acting as a certificate repository, they give administrators a central location for entry of personal attribute information. Entries might include network resources such as file servers, printers, and URLs. User information, such as e-mail address, telephone privileges, and certificates, is accessible from numerous clients in a controlled fashion. Directory clients can locate entries and their attributes using a directory access protocol such as *Lightweight Directory Access Protocol* (LDAP).

LDAP, defined by RFCs 1777 and 1778, was designed to give applications a means to access X.500 directories. It has been widely adopted because it is simpler and easier to use than the X.500 standard protocols. Because it is not directory-specific, LDAP has also found its way into various environments, enhancing its interoperability.

NOTE:

Because of the self-verifying nature of certificates, certificate directories themselves do not necessarily have to be trusted. Should a directory be compromised, certificates can still be validated through the standard process of checking the certificate chain through the CA. If the directory server contains personal or corporate data, however, it may be necessary to provide security and access control to it.

Key Recovery Server

In a PKI population of any size, one thing is sure to happen: End users will lose their private keys. Whether the loss results from hardware failure or a forgotten password, it can create a significant burden on all parties in the PKI. With the loss of a private key, for example, the CA must revoke the corresponding PKC; in addition, a new key pair must be generated, and a new corresponding PKC must be created. As a result, all data encrypted before the incident becomes unrecoverable.

One solution is to provide a *key recovery server* (or, more accurately, a key backup and recovery server). As the name implies, the key recovery server gives the CA a simple way of backing up private keys at the time of creation and recovering them later.

Although key recovery servers can save considerable time and money, problems can arise. For example, the key used to decrypt data could be the same key used to sign messages (that is, the user's private key). In this case, an attacker could access the user's private key and forge messages in the user's name. For that reason, some CAs support two key pairs: one for encryption and decryption and another one for signature and verification. We discuss the storage of multiple key pairs later in this chapter in the section titled "Managing Multiple Key Pairs."

NOTE:

The term "escrow" is sometimes used interchangeably with "recovery." There is, however, a clear distinction between the two. A key recovery server is implemented in a given PKI by its administrators to provide recovery functions for end users. In key escrow, on the other hand, a third party (such as a federal or local law enforcement agency) is given keys needed as evidence in an investigation.

Management Protocols

Management protocols assist in the online communication between end users and management within a PKI. For example, a management protocol might be used to communicate between an RA and an end user or between two CAs that cross-certify each other. Examples of PKI management protocols include *Certificate Management Protocol* (CMP) and message formats such as *Certificate Management Message Format* (CMMF) and PKCS #10.

Management protocols should support the following functions:

- **Registration** This is the process whereby a user first makes herself or himself known to a CA (directly or through an RA).
- **Initialization** Before an end user system can operate securely, it is necessary to install key materials that have the appropriate relationship with keys stored elsewhere in the infrastructure. For example, the end-user system must be securely initialized with the public key and other assured information of the trusted CA(s), to be used in validating certificate paths. Furthermore, a client typically must be initialized with its own key pair(s).
- **Certification** This is the process in which a CA issues a certificate for a user's public key and then either returns the certificate to the end user's client system or posts the certificate in a repository (or both).
- **Key recovery** As an option, end user client key materials (for example, a user's private key used for encryption purposes) can be backed up by a CA or a key backup system. If a user needs to recover these backed-up key materials (for example, as a result of a forgotten password or a lost key chain file), an online protocol exchange may be needed to support such recovery.
- **Key update** All key pairs must be updated regularly. In this process, key pairs are replaced and new certificates are issued.
- **Revocation** This process is invoked when an authorized person advises a CA of an abnormal situation requiring certificate revocation.
- **Cross-certification** Two CAs exchange information used in establishing a cross-certificate. A cross-certificate is a certificate issued by one CA to another CA that contains a CA signature key used for issuing certificates.

NOTE:

Online protocols are not the only way to implement these functions. Offline methods can also be used.

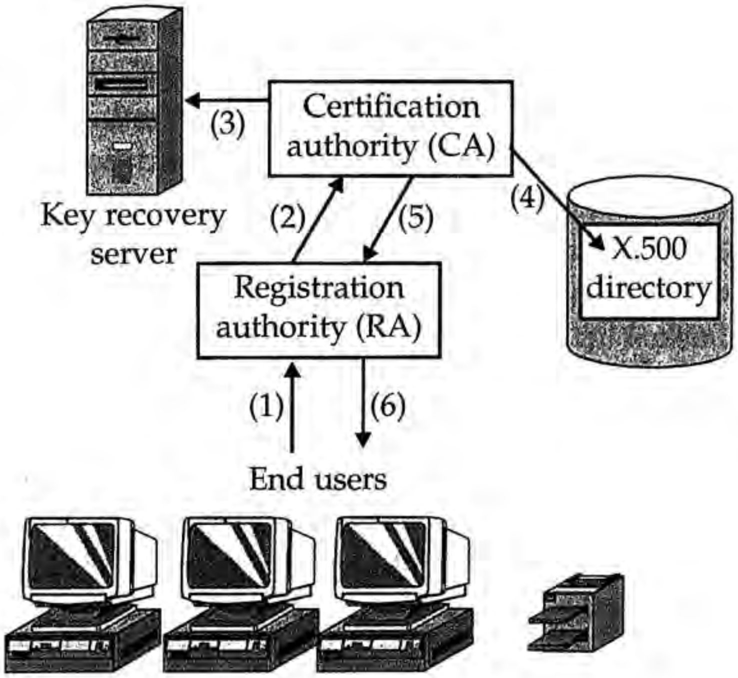
Operational Protocols

Operational protocols are those protocols that enable the transfer of certificates and revocation status information between directories, end users, and relying parties. The X.509 standard does not specify any single protocol for use within a PKI domain. Instead, the standard specifies how the data should be structured for transport. The following protocols are commonly used within an environment: HTTP, FTP, e-mail, and LDAP.

Figure 6-2 illustrates the ways in which the various components of PKI interact.

Figure 6-2

The interaction between the various PKI components



Registering and Issuing Certificates

CAs can register end users in various ways, often depending greatly on the environment. Many end users simply register with the CA or RA via the Internet using a Web browser. A private corporate PKI may use an automated system to register newly hired employees.

In either case, registration is one of the most important processes in a PKI. It is at this point that the end user and the CA establish trust.

Depending on the type of certificate being issued, each party may go to great lengths to validate the other. For its part, the end user may review the CA's published certificate policies and certification practice statements. For the CA to establish trust with the end user, the CA may require financial documentation and proof of identity through in-person communications.

After registration is complete and a relationship of trust has been established between the CA and the end user, a certificate request can be initiated. One of two approaches can be used. The end user generates a key pair and provides the public key in the form of a standard PKCS #10 *certificate-signing request* (CSR), or the CA can generate a key pair on behalf of the end user.

Revoking a Certificate

Certificates are created in the belief that they will be valid and usable throughout the expected lifetime indicated in the Validity field. In some cases, however, an unexpired certificate should no longer be used. For example, the corresponding private key may have been compromised, the CA has discovered that it has made a mistake, or the holder of the key is no longer employed at a company. As a result, CAs need a way to revoke an unexpired certificate and notify relying parties of the revocation.

The most common method is the use of a *certificate revocation list* (CRL). Simply stated, a CRL is a signed data structure containing a time-stamped list of revoked certificates. The signer of the CRL is typically the same entity that originally issued it (the CA). After a CRL is created and digitally signed, it can be freely distributed across a network or stored in a directory in the same way that certificates are handled.

CAs issue CRLs periodically on schedules ranging from every few hours to every few weeks. A new CRL is issued whether or not it contains any new revocations; in this way, relying parties always know that the most recently received CRL is current. A PKI's certificate policy governs its CRL time interval. Latency between CRLs is one of the major drawbacks of their use. For example, a reported revocation may not be received by the relying party until the next CRL issue, perhaps several hours or several weeks later.

NOTE:

Currently, most applications (such as Web browsers and e-mail readers) do not use the various revocation mechanisms that are in place. However, this is beginning to change as PKIs are becoming more widespread.

Certificate Revocation Lists

As stated previously, a CRL is nothing more than a time-stamped, digitally signed list of revoked certificates. The following section describes, in detail, the various fields that make up a CRL. Figure 6-3 illustrates these fields.

Figure 6-3

The standard structure of a CRL

Version
Signature Algorithm Identifier
Issuer Name
This Update (Date/Time)
Next Update (Date/Time)
User Certificate Serial Number / Revocation Date
CRL Entry Extensions
⋮
User Certificate Serial Number / Revocation Date
CRL Entry Extensions
CRL Extensions
Signature

- **Version** This field indicates the version of the CRL. (This field is optional for Version 1 CRLs but must be present for Version 2.)
- **Signature Algorithm Identifier** This field contains the identifier of the algorithm used to sign the CRL. For example, if this field

contains the object identifier for SHA-1 with RSA , it means that the digital signature is a SHA-1 hash (see Chapter 5) encrypted using RSA (see Chapter 4).

- **Issuer Name** This field identifies the DN, in X.500 format, of the entity that issued the CRL.
- **This Update (Date/Time)** This field contains a date/time value indicating when the CRL was issued.
- **Next Update (Date/Time)** This optional field contains a date/time value indicating when the next CRL will be issued. (Although this field is optional, RFC2459 mandates its use.)
- **User Certificate Serial Number/Revocation Date** This field contains the list of certificates that have been revoked or suspended. The list contains the certificate's serial number and the date and time it was revoked.
- **CRL Entry Extensions** These fields are discussed in the following section.
- **CRL Extensions** These fields are discussed in the section "CRL Extensions."
- **Signature** This field contains the CA signature.

CRL Entry Extensions

Just as an X.509 Version 3 certificate can be enhanced through the use of extensions, Version 2 CRLs are provided a set of extensions that enable CAs to convey additional information with each individual revocation. The X.509 standard defines the following four extensions for use with a Version 2 CRL:

- **Reason Code** This extension specifies the reason for certificate revocation. Valid entries include the following: unspecified, key compromise, CA compromise, superseded, certificate hold, and others. (For valid reasons, RFC2459 recommends the use of this field.)
- **Hold Instruction Code** This noncritical extension supports the temporary suspension of a certificate. It contains an OID that describes the action to be taken if the extension exists.
- **Certificate Issuers** This extension identifies the name of the certificate issuer associated with an indirect CRL (discussed later in the section titled "Indirect CRLs"). If this extension is present, RFC2459 mandates that it be marked critical.

- **Invalidity Date** This noncritical extension contains a date/time value showing when a suspected or known compromise of the private key occurred.

CRL Extensions

The following CRL extensions have been defined on a per-CRL basis:

- **Authority Key Identifier** This extension can be used to differentiate between multiple CRL signing keys held by this CA. This field contains a unique key identifier (the subject key identifier in the CRL signer's certificate). The use of this field is mandated by RFC2459.
- **Issuer Alternative Name** This extension associates one or more alternative name forms with the CRL issuer. RFC2459 specifies that if no DN is specified in the subject field of a certificate, it must have one or more alternative names, and this extension must be flagged critical. RFC2459 recommends the use of this extension when alternative name forms are available but mandates that it not be marked critical.
- **CRL Number** This noncritical extension provides a means of easily recognizing whether a given CRL has been superseded. It contains a unique serial number relative to the issuer of this CRL. Although this extension is noncritical, RFC2459 mandates its use.
- **Delta CRL Indicator** This critical extension identifies the CRL as a delta CRL and not a base CRL (see later section, "Delta CRLs"). If this extension is present, RFC2459 mandates that it be marked critical.
- **Issuing Distribution Point** This critical extension identifies the name of the CRL distribution point for a given CRL (see next section). It also indicates whether the CRL covers revocation of end user certificates only or of CA certificates only, and it specifies whether the certificate was revoked for a set reason. This extension can also be used to indicate that the CRL is an indirect CRL. If this extension is present, RFC2459 mandates that it be marked critical.

CRL Distribution Points

What happens when the CRL for a given PKI domain becomes too large? CRL *distribution points* (sometimes referred to as CRL *partitions*) provide a simple solution. The idea is that instead of a single large CRL, several smaller CRLs are created for distribution. Relying servers retrieve and process these smaller CRLs more easily, saving time, money, and bandwidth.

To use CRL distribution points, the CA supplies a pointer to a location within the Issuing Distribution Point extension. Examples of such pointers are a DNS name, an IP address, or the specific filename on a Web server. The pointer enables relying parties to locate the CRL distribution point.

Delta CRLs

A *delta* CRL lists only incremental changes that have occurred since the preceding CRL. In this way, delta CRLs provide a way to significantly improve processing time for applications that store revocation information in a format other than the CRL structure. With this approach, such applications can add new changes to their local database while ignoring unchanged information already stored there. After an initial full CRL (*base* CRL) posting, an accurate list of revoked certificates is maintained through delta CRLs. As a result, delta CRLs can be issued much more often than can base CRLs.

CAs use the Delta CRL Indicator extension to indicate the use of delta CRLs. In addition, a special value, the "Remove from CRL" value, can be used in the Reason Code extension to specify that an entry in the base CRL may now be removed. An entry might be removed because certificate validity has expired or the certificate is no longer suspended.

Indirect CRLs

Indirect CRLs are another alternative for improving the distribution of CRLs. As the name implies, an *indirect* CRL is provided to the relying party by a third party that did not necessarily issue the certificate. In this way, CRLs that otherwise would be supplied by numerous CAs (or other revoking authorities) can be consolidated into a single CRL for distribution. For example, suppose that a private PKI is served by multiple CAs. By using indirect CRLs, the PKI can receive one CRL issued by one CA (or other trusted third party) on behalf of the other CAs.

Two CRL extensions enable the use of indirect CRLs. To indicate that a CRL contains revocation information from multiple CAs, the Indirect CRL attribute is set to True. It's also important to provide the relying party with additional information concerning revocation of each entry. A CRL entry for each certificate is used to identify its CA. If there is no CRL entry, the certificate is assumed to have been issued by the CA listed on the first line of the CRL.

Suspending a Certificate

At times, a CA needs to limit the use of a certificate temporarily but does not require that it be revoked. For example, a corporate end user may be going on vacation. In such cases, the certificate can be *suspended*, disabling the use of PKI-enabled applications that should not be accessed in the employee's absence. When the employee returns, the CA removes the suspension. This approach saves the CA time by not requiring it to revoke and then reissue the certificate. To suspend a certificate, the CA uses the value Certificate Hold in the Reason Code extension of the CRL.

Authority Revocation Lists

Like end users, CAs themselves are identified by certificates. Just as end user certificates may require revocation, so do CA certificates. An *authority revocation list* (ARL) provides a means of disseminating this revocation information for CAs. ARLs are distinguished from CRLs via the Issuing Distribution Point field within the revocation list.

Online Certificate Status Protocol

Depending on the size of the PKI population, CRLs can become unwieldy. Even if you use the CRL techniques we've discussed (CRL distribution points, indirect CRLs, and delta CRLs), the workload associated with CRLs can become burdensome. On the other end, relying parties must spend considerable resources obtaining the most current CRL.

A newer protocol, the *Online Certificate Status Protocol* (OCSP), can be used to check whether a digital certificate is valid at the time of a given transaction. OCSP enables relying parties to conduct these checks in real time, providing a faster, easier, and more dependable way of validating digital certificates than the traditional method of downloading and processing CRLs. Figure 6-4 illustrates the interaction between various OCSP components.

Here's how it works. The CA provides a server, known as an OCSP *responder,* that contains current revocation information. Relying parties can query the OCSP responder to determine the status of a given certificate. The best way to obtain the information is to have the CA feed it directly into the responder. Depending on the relationship between the CA and the OCSP responder, the CA can forward immediate notification of a certificate's revocation, making it instantly available to users.

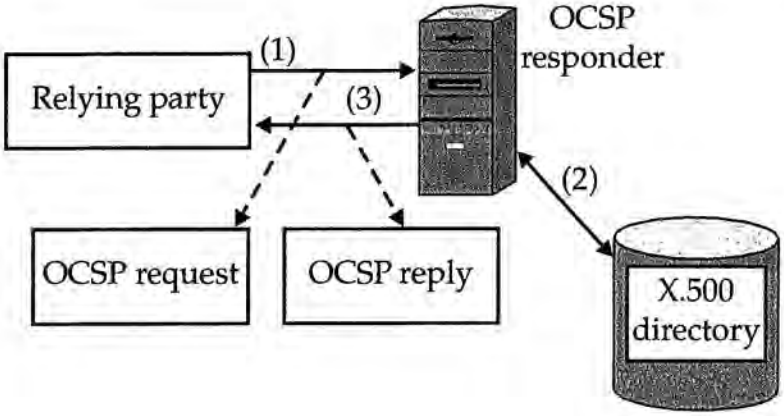


Figure 6-4
Interaction between a relying part and an OCSP responder

The relying party sends a simple request to the OCSP responder, suspending the use of the certificate in question until a response is received. The OCSP request contains the protocol version, the service requested, and one or more certificate identifiers. The certificate identifier consists of a hash of the issuer's name, a hash of the issuer's public key, and the certificate serial number.

The OCSP responder provides a digitally signed response for each of the certificates in the original request. Replies consist of a certificate identifier, one of three status values (Good, Revoked, or Unknown), and a validity interval (This Update and, optionally, Next Update). The response may also include the time of revocation as well as the reason for revocation.

NOTE:
RFC2560 states that an OCSP request must be protocol-independent, although HTTP is the most common approach in use.

Trust Models

Trust models are used to describe the relationship between end users, relying parties, and the CA. Various models can be found in today's PKIs. The following describes the two most widely used and well known: certificate hierarchies and cross-certification models.

It should be noted, however, that each of these can be used not only alone but in conjunction with one another as well.

Certificate Hierarchies

As a PKI population begins to increase, it becomes difficult for a CA to effectively track the identities of all the parties it has certified. As the number of certificates grows, a single authority may become a bottleneck in the certification process. One solution is to use a *certificate hierarchy,* in which the CA delegates its authority to one or more subsidiary authorities. These authorities, in turn, designate their own subsidiaries, and the process travels down the hierarchy until an authority actually issues a certificate. Figure 6-5 illustrates the concept of certificate hierarchies.

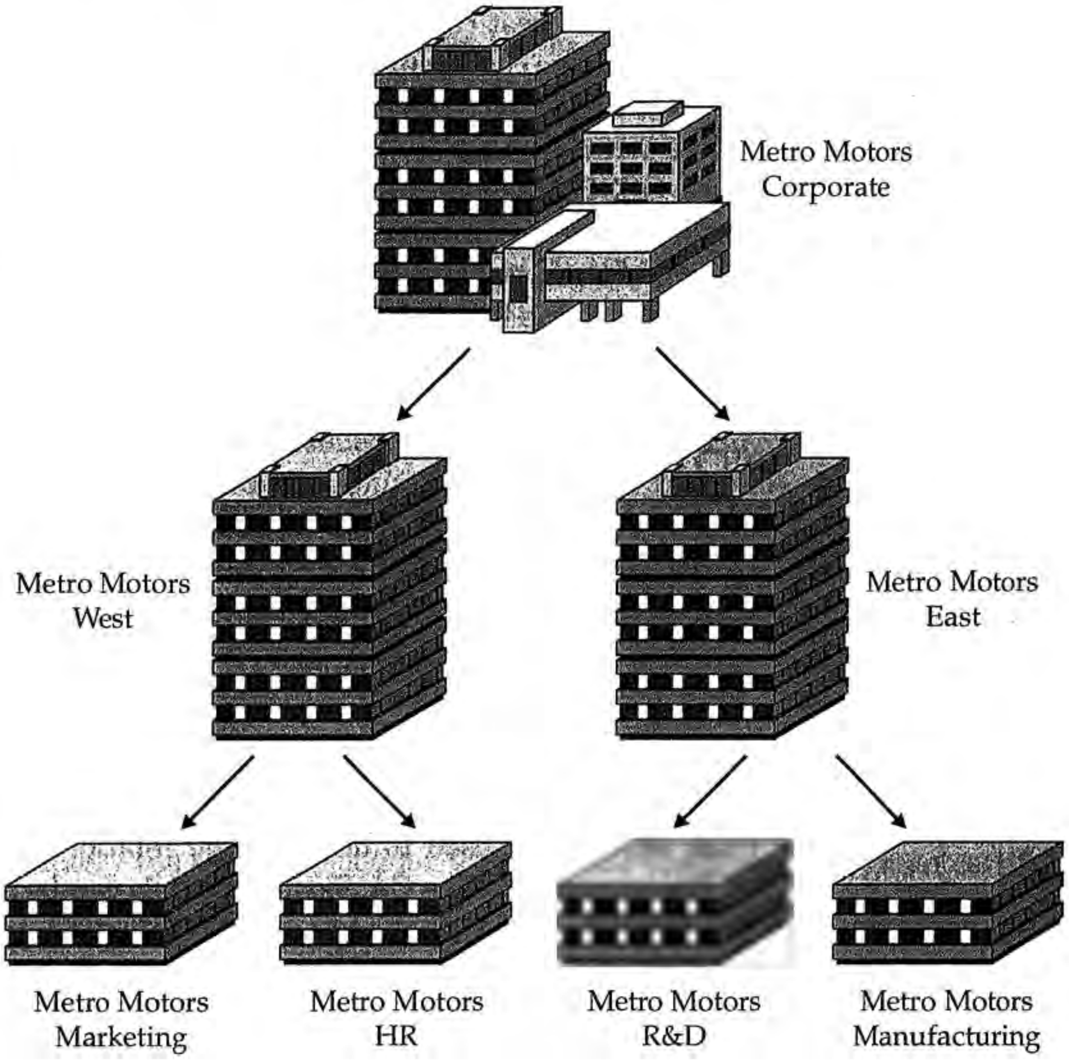


Figure 6-5

This simple certificate hierarchy might occur in a large corporation

A powerful feature of certificate hierarchies is that not all parties must automatically trust all the certificate authorities. Indeed, the only authority whose trust must be established throughout the enterprise is the highest CA. Because of its position in the hierarchy, this authority is generally known as the *root* authority. Examples of current public root CAs include Verisign, Thawte, and the U.S. Postal Service's root CA.

Cross-Certification

The concept of a single, monolithic PKI serving every user in the world is unlikely to become a reality. Instead, we will continue to see PKIs established between nations, political organizations, and businesses. One reason for this practice is the policy that each CA should operate independently and follow its own rules. Cross-certification enables CAs and end users from different PKI domains to interact. Figure 6-6 illustrates the concept of cross-certification.

Cross-certification certificates are issued by CAs to form a nonhierarchical *trust path.* A mutual trust relationship requires two certificates, which cover the relationship in each direction. These certificates must be supported by a cross-certification agreement between the CAs. This agreement governs the liability of the partners in the event that a certificate turns out to be false or misleading.

After two CAs have established a trust path, relying parties within a PKI domain are able to trust the end users of the other domain. This capability is especially useful in Web-based business-to-business communications. Cross-certification also proves useful for intradomain communications when a single domain has several CAs.

NOTE:

The use of cross-certification instead of or in conjunction with certificate hierarchies can prove to be more secure than a pure hierarchy model. In a hierarchy, for example, if the private key of the root CA is compromised, all subordinates are rendered untrustworthy. In contrast, with cross-certification, the compromising of one CA does not necessarily invalidate the entire PKI.

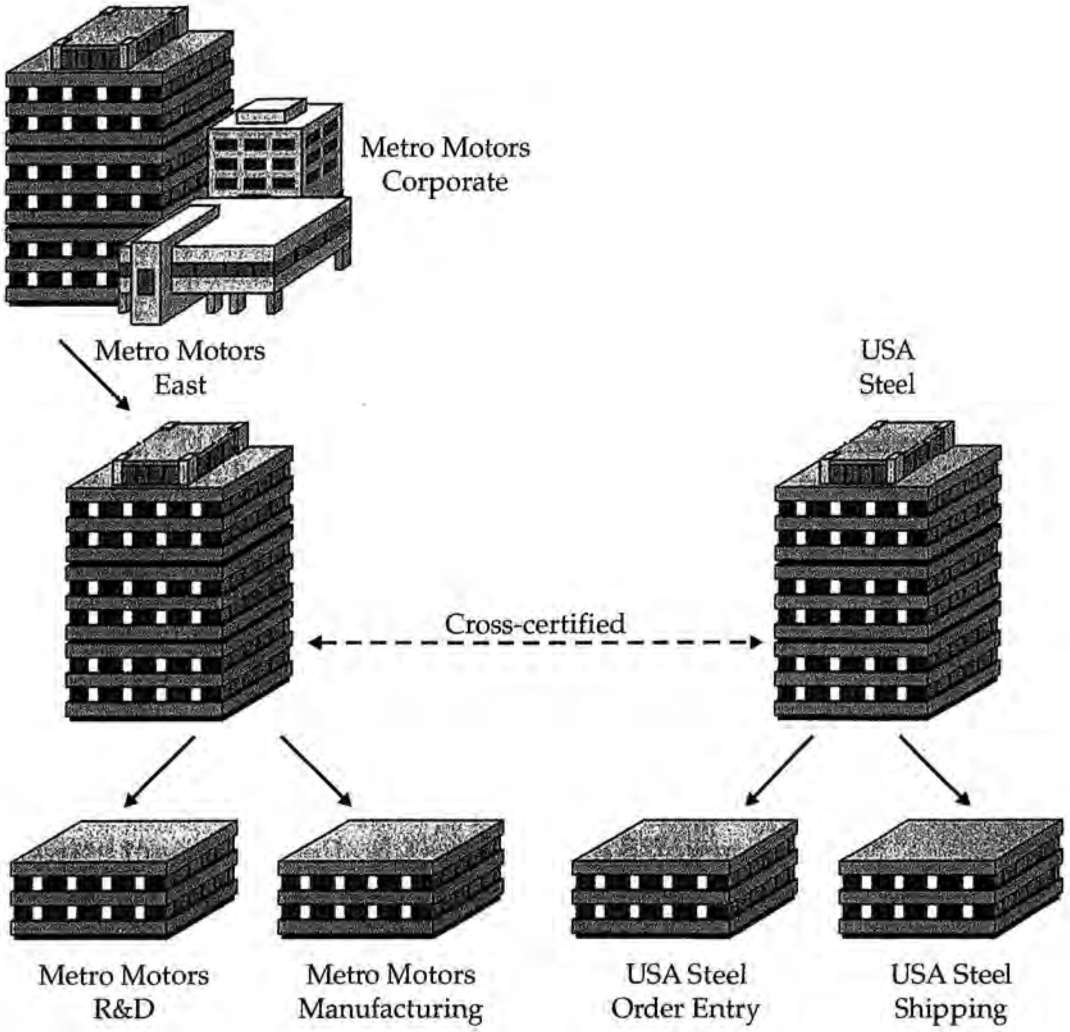


Figure 6-6
Cross-certification

X.509 Certificate Chain

A *certificate chain* is the most common method used to verify the binding between an entity and its public key. To gain trust in a certificate, a relying party must verify three things about each certificate until it reaches a trusted root. First, the relying party must check that each certificate in the chain is signed by the public key of the next certificate in the chain. It must also ensure that each certificate is not expired or revoked and that each certificate conforms to a set of criteria defined by certificates higher up in the chain. By verifying the trusted root for the certificate, a certificate-using application that trusts the certificate can develop trust in the entity's public key. Figure 6-7 illustrates certificate chains and how they may be used.

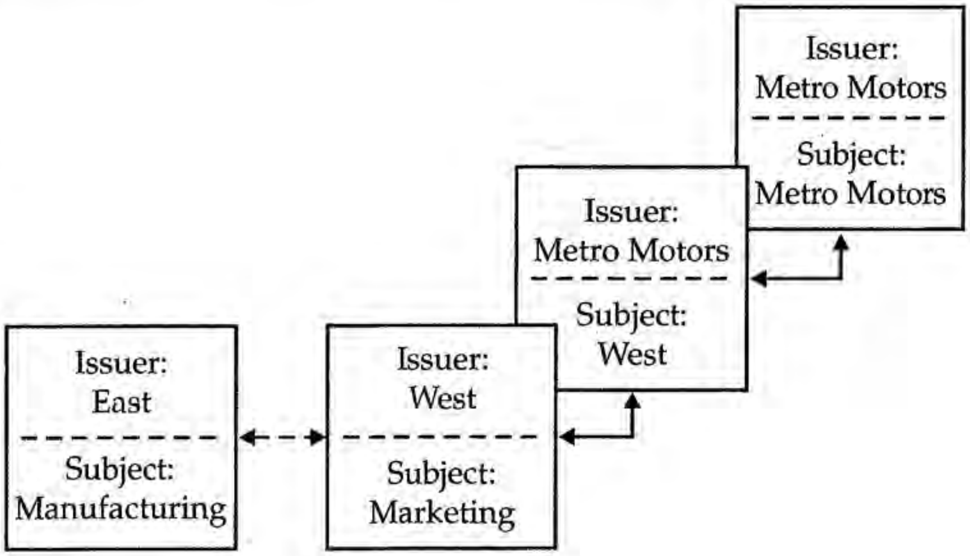


Figure 6-7
A certificate chain

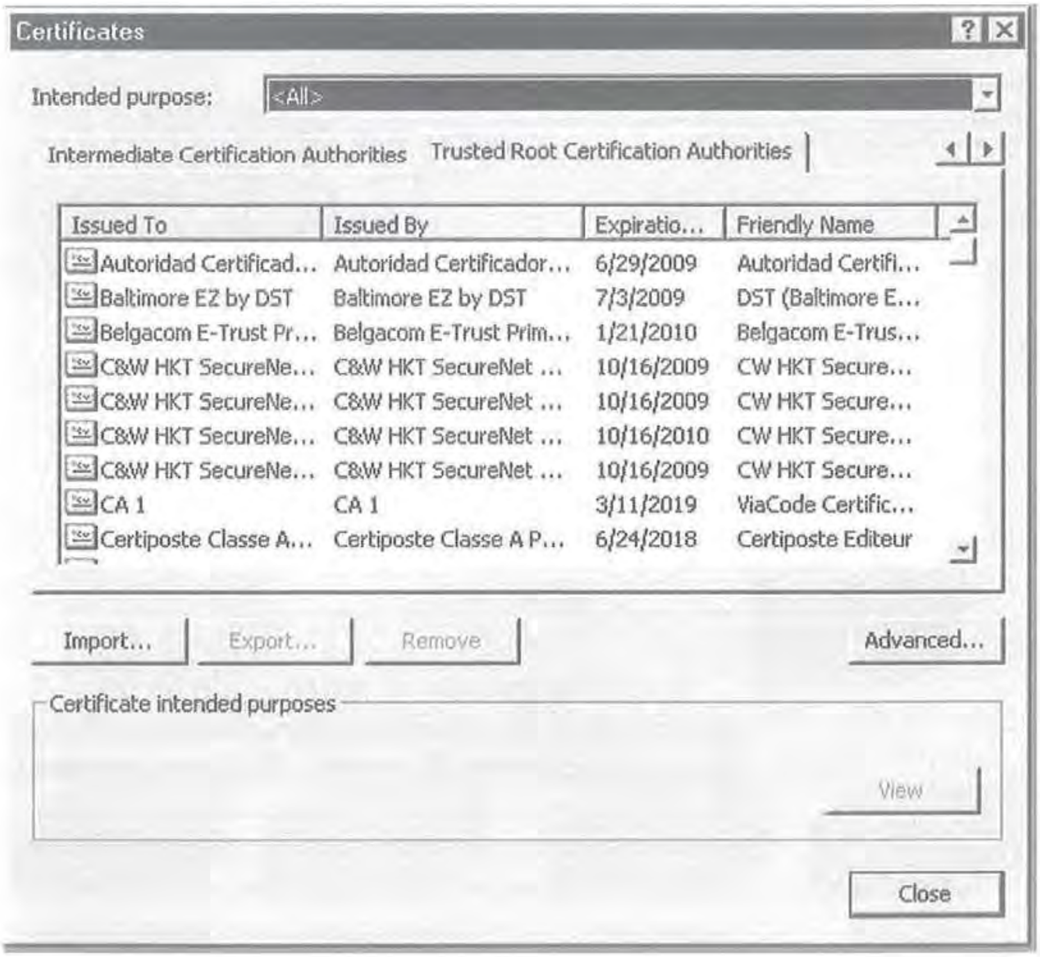
To see this process in action, consider what happens when a client application in the marketing department verifies the identity of the marketing department's Web server. The server presents its certificate, which was issued by authority of the manufacturing department. The marketing client does not trust the manufacturing authority, however, so it asks to see that authority's certificate. When the client receives the manufacturing authority's certificate, it can verify that the manufacturing authority was certified by the corporation's root CA. Because the marketing client trusts the root CA, it knows that it can trust the Web server.

The Push Model Versus the Pull Model

The chaining described here relies on individuals having access to all the certificates in the chain. How does the relying party get these certificates? One way is for the issuer to send an entire chain of certificates when sending one certificate (see Figure 6-8). This is the *push* model, in which the sender pushes the entire chain of certificates to the recipient, and the recipient can immediately verify all the certificates. The *pull* model sends only the sender's certificate and leaves it up to the recipient to pull in the CA's certificate. Because each certificate contains the issuer's name, the recipient knows where to go to verify the certificate. (To make searches easier, Version 3 certificates offer more fields to hold information.) Even with the push model, however, some recipient chaining may be necessary.

Figure 6-8

Internet Explorer provides a set of trusted root certification authorities



Issued To	Issued By	Expiratio...	Friendly Name
Autoridad Certificad...	Autoridad Certificador...	6/29/2009	Autoridad Certifi...
Baltimore EZ by DST	Baltimore EZ by DST	7/3/2009	DST (Baltimore E...
Belgacom E-Trust Pr...	Belgacom E-Trust Prim...	1/21/2010	Belgacom E-Trus...
C&W HKT SecureNe...	C&W HKT SecureNet ...	10/16/2009	CW HKT Secure...
C&W HKT SecureNe...	C&W HKT SecureNet ...	10/16/2009	CW HKT Secure...
C&W HKT SecureNe...	C&W HKT SecureNet ...	10/16/2010	CW HKT Secure...
C&W HKT SecureNe...	C&W HKT SecureNet ...	10/16/2009	CW HKT Secure...
CA 1	CA 1	3/11/2019	ViaCode Certific...
Certiposte Classe A...	Certiposte Classe A P...	6/24/2018	Certiposte Editeur

Managing Key Pairs

The management of key pairs—the policies whereby they are generated and protected—is important in any PKI. As described in this section, such policy decisions depend greatly on the intended purpose of the keys. For example, keys that enable nonrepudiation for e-commerce transactions are more likely to be handled with greater care than those used to provide for secure e-mail.

Generating Key Pairs

Keys can be generated in one of two ways. In the first option, key pairs are generated on the end user's system. The second option requires that a trusted third party (such as the CA or its delegated RA) generate the key pair. Which of these options is more appropriate is a matter of debate. Each approach has advantages and disadvantages.

End-user generation of keys can be effective, especially for generating keys for the purpose of nonrepudiation. This option enables the user to build greater confidence in the trust shared with relying parties because the key value is never exposed to another entity. One problem is that the end user must provide software or hardware to generate strong keys. Even though most browsers incorporate this functionality, it tends to be CPU-intensive and slow. In addition, end users face the task of securely transporting the public key to the CA (or corresponding RA) for certification.

The second method, in which a central system such as the CA or one of its RAs generates key pairs, also has its advantages. A central system commonly has greater resources to provide for faster key generation. Furthermore, an end user may require cryptographically strong keys that have been generated by a trusted and independently evaluated cryptographic module. In other cases, an end user may need private key backup, and this service can be easily accommodated without unnecessary transfer of the private key.

Because each approach offers benefits, many CAs support both options. Yet another option is the use of multiple key pairs. Here, end users generate keys used to provide nonrepudiation, and the central system provides the keys for encryption.

Protecting Private Keys

The strength of public-key cryptographic systems and their associated certificates relies greatly on the security of private keys. It is crucial that only the certified owner—the person or organization identified in the certificate—use the corresponding private key. The following mechanisms are used to safeguard and limit access to private keys:

- **Password protection** This is the most common form of protection employed by PKIs. A password or *personal identification number* (PIN) is used to encrypt the private key, which is stored on the local hard disk. However, if the key can be obtained from the hard disk, the problem of accessing the key is reduced to simple password guessing. As a result, this is considered the least secure method and is generally not thought to be a long-term solution.

- **PCMCIA cards** (Personal Computer Memory Card International Association) To reduce the risk of a key being stolen from the user's hard disk, many vendors have begun to offer the option of storing keys on chip cards. Because the key must still leave the card and enter the system's memory, however, it remains vulnerable to theft. Chip cards are discussed in Chapter 9.
- **Tokens** With *tokens*, the private key is stored in an encrypted format in a hardware device and can be unlocked only through the use of a one-time passcode provided by the token. Although this technique is more secure than those mentioned so far, the token still must be available to the end user whenever the private key is needed, and it can be lost.
- **Biometrics** The key is associated with a unique identifying quality of an individual user (for example, a fingerprint, a retinal scan, or a voice match). The idea is that biometrics can provide the same level of security as tokens while alleviating the need for the user to carry a device that can be lost.
- **Smart cards** In a true smart card (see Chapter 3), the key is stored in a tamperproof card that contains a computer chip, enabling it to perform signature and decryption operations. Thus, the key never leaves the card, and the possibility of compromise is greatly reduced. However, the user must carry a device, and if the card was used for encryption and is lost, the encrypted data may be unrecoverable.

NOTE:

Most users take few or no precautions to protect their private keys from theft. As public-key technology becomes more widely used, organizations will probably devote more time to awareness programs and education.

Managing Multiple Key Pairs

As stated throughout this chapter, it is not uncommon for end users to have more than one certificate for various purposes, and therefore they may have various key-pair types. For example, a key used to digitally sign a document for purposes of nonrepudiation is not necessarily the same one that would be used for the encryption of files. For this reason, it is crucial that end users as well as PKI administrators be aware of the various management techniques used to secure these keys.

A private key that is used to provide digital signatures for the purposes of nonrepudiation requires secure storage for the lifetime of the key. During its lifetime, there is no requirement for backup; if the key is lost, a new key pair should be generated. After the lifetime of the key has expired, the key should not be archived. Instead, it should be securely destroyed. This practice ensures against unauthorized use that may occur years after the key is considered expired. The use of secure time-stamping can also help reduce fraud. To authenticate data signed by these private keys, it is necessary to maintain the corresponding PKC.

NOTE:

For private keys used for nonrepudiation, the ANSI X9.57 standard requires that they be created, used, and destroyed within one secure module.

Conversely, a private key used to support encryption should be backed up during its lifetime to enable recovery of encrypted information. After the private key is considered expired, it should be archived to support later decryption of encrypted legacy data. Whether and how corresponding public keys should be backed up and archived greatly depends on the algorithm in use. With RSA, the public key does not require backup or archiving. If Diffie-Hellman key agreement was used, on the other hand, the public key will be required to recover data at a later time.

Updating Key Pairs

As mentioned earlier in this chapter in the section titled “Management Protocols,” good security practices dictate that key pairs should be updated periodically. One reason is that, over time, keys become susceptible to compromise through cryptanalytic attacks. After a certificate has expired, one of two things can occur: The CA can reissue a new certificate based on the original key pair, or a new key pair can be generated and a new certificate issued.

Key pairs can be updated in one of two ways. In a manual update, it is left to the end user to recognize that the certificate is about to expire and request an update. This approach places a considerable burden on users to keep track of a certificate's expiration date. Failing to request a timely update will put the user out of service and unable to communicate securely. As a result, the end user must perform an off-line exchange with the CA.

A better solution is an automated update, in which a system is in place to check the validity of the certificate each time it is used. As the certificate approaches expiration, the automated system initiates a request for key update with the appropriate CA. When the new certificate is created, the system automatically replaces the old certificate. In this way, the end user is free to carry out secure operations uninterrupted.

Keeping a History of Key Pairs

A CA's published policy states the time period during which a given certificate can be considered valid (typically, one year). As a result, it's not uncommon for a user to accumulate three or more key pairs within three years. A key history mechanism provides a way of archiving keys and certificates for later use. The other alternatives, such as decrypting and reencrypting data as new keys are generated, would be impractical in most environments.

Such a history is of great importance to any PKI. For example, suppose that a data file was signed with my private signing key three years ago. How does a relying party get a copy of the corresponding PKC to verify the signature? Similarly, what if the public key from my certificate was used to encrypt some data or another symmetric key to perform a digital enveloping process five years ago? Where can the corresponding private decryption key be found? If a key history has been kept, the necessary keys for both scenarios will be available.

NOTE:

As stated earlier, similar keys can be used for various purposes (for example, private keys can be used not only for decryption but also for signing). Because a key's purpose dictates the method of storage, it may be necessary to have two or more separate key pairs.

Deploying a PKI

As organizations plan for deploying PKIs, they have three basic options: outsourcing, insourcing, or running their own. With *outsourcing,* a third party runs a CA on behalf of the organization. This option requires the organization to have a great deal of trust in the third party and its policies and practices. The advantage of outsourcing is that the organization can leverage outside expertise and resources that it may not have in-house.

With *insourcing,* an organization provides its own resources, but the administrative staff is leveraged from outside. This option enables an organization to maintain control over its own CA policies while taking advantage of outside expertise. Many PKI vendors, including Entrust Technologies and Verisign, include this service in their standard offerings.

Finally, it is possible for an organization to run its own CA. By using PKI-enabling products or building its own, an organization manages every aspect of the PKI. This option greatly benefits organizations that have in-house expertise, affording them the most flexibility and control over the system.

The Future of PKI

PKIs have grown considerably in the past decade as increasing numbers of organizations have become dependent on them. However, many improvements are in the works, not only by noncommercial organizations such as the *International Organization for Standardization* (ISO) and *Internet Engineering Task Force* (IETF) but also by many PKI vendors. Two such improvements are roaming certificates and attribute certificates, discussed in the next two sections.

Roaming Certificates

As you've seen, standard certificates do a great job of binding an individual to a public key, but a new problem has arisen: the need for portability. It is not uncommon for a user to move among several computers within an organization. A certificate can be placed on every possible machine, but in order to be effective, the private key also must be present.

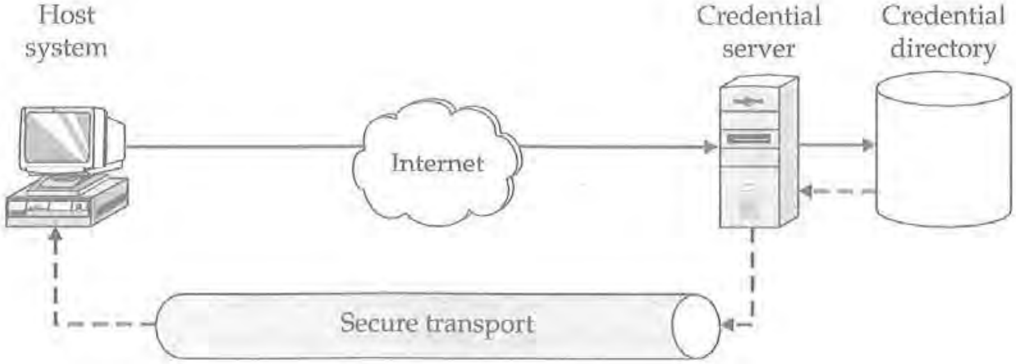
Until recently, only two real solutions have provided the mobility of certificates and their corresponding private keys. The first is smart card technology, in which the public/private key pair is stored on the card. However, this option has drawbacks, such as the inconvenience of carrying an item that can be lost or damaged. In addition, smart cards are usable only on systems that have a smart card reader. The second option, which is not much better, is to copy the certificate and private key onto a floppy for later use. Again, the user is forced to carry an item that can be lost or damaged, and a floppy is not as cryptographically secure as a smart card.

A new solution is the use of *roaming certificates* (perhaps better stated as roaming certificates and private keys), which are provided through third-party software. Properly configured on any system, the software (or plug-in) enables a user access to his or her public/private key pairs. The concept is simple. Users' certificates and private keys are placed in a secure central server. When the user logs into a local system, the public/private key pair is securely retrieved from the server and placed in the local system's memory for use. When the user has completed work and logs off of the local system, the software (or plug-in) scrubs the user's certificate and private key from memory.

To date, this technology has been limited mainly to private PKIs, such as corporations, because of scalability issues. However, as roaming applications and users become more prevalent, it's conceivable that roaming certificate technology will be developed into a cost-effective way of providing virtual PKIs worldwide. Figure 6-9 illustrates the interaction of common roaming certificate systems.

Figure 6-9

Roaming certificates



NOTE:

Although the concept of roaming certificates has proven quite useful, some standards do not support this technology. One such standard is ANSI X9.57, which requires that private keys used for the purposes of nonrepudiation be created, used, and destroyed within one secure module.

Attribute Certificates

Another popular emerging standard is the *attribute certificate* (AC). Although ACs are similar in structure to public-key certificates, ACs provide different functionality. ACs do not contain a public key for an individual. Instead, they are used to bind an entity to a set of attributes that specify membership, role, security clearance, or other authorization information. Attribute certificates, like public-key certificates, are digitally signed to prevent changes after the fact.

In conjunction with current authentication services, ACs can provide a means to transport authorization information securely. Applications that can use this technology include those that provide remote access to network resources (such as Web servers and databases) and those that control physical access to buildings and facilities. For example, after a user signs on, his or her identity can be verified through the use of the current public-key certificate. After the user has logged in, his or her public key can be used to create a secure session with an access control server, and the user's attribute certificate can be checked against a list of valid users. Figure 6-10 illustrates a standard attribute certificate.

NOTE:

ISO has defined the basic attribute certificate, and IETF is currently profiling these definitions for use in Internet environments.

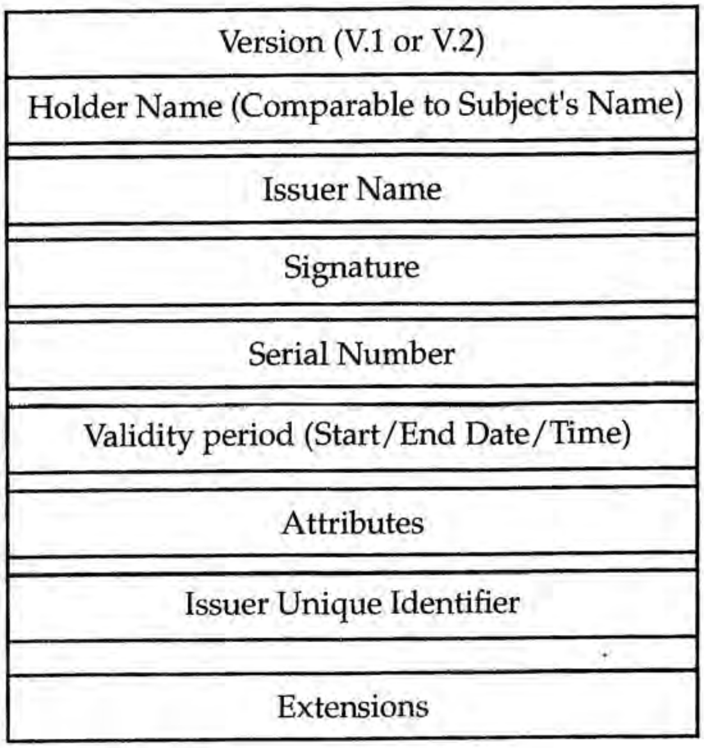


Figure 6-10 Controlling access with attribute certificates

Certificate Policies and Certification Practice Statements

Certification authorities act as trusted third parties, vouching for the contents of the certificates they issue. But what exactly does a CA certify? What makes one CA more trusted than another? Two mechanisms are used by CAs to establish trust among end users and relying parties. These are certificate policies and certification practice statements.

The X.509 standard defines a certificate policy as "a named set of rules that indicates the applicability of a certificate to a particular community and/or class of application with common security requirements." One or more certificate policies can be identified in the standard extensions of an X.509 Version 3 certificate. As relying parties obtain a certificate for processing, they can use the policies specified in that certificate to make a decision of trust.

A more detailed description of practices is made available through the use of a certification practice statement, a concept originated by the *American Bar Association* (ABA). According to the ABA's "Digital Signature Guidelines," a CPS is "a statement of the practices which the certification authority employs in issuing certificates." A CPS gives relying parties a basis for making a trust decision concerning a CA.

The relationship between certificate policies and CPSs is not entirely clear. Each kind of document was created for unique reasons by different sources. CPSs tend to provide a detailed statement about a CA's practices, whereas certificate policies tend to provide a broader definition of practices.

RFC2527 outlines the key components of a CPS as follows:

- **Introduction** This part of a CPS provides a general overview of the certificate policy definition, indicating any applicable names or other identifiers (for example, ASN.1 object identifiers) that are used in the statement. It should also provide all contact information (name, phone number, address, and so on) of the responsible authority.
- **General Provisions** This section describes the various obligations, rights, and liabilities of the CA or RA, end users, and relying parties. It also includes information about how and how often certificates and CRLs will be published.
- **Identification and Authentication** This section describes the procedures used by the CA or RA to authenticate an end user applicant. It also describes how end users should request certificate revocations and key updates.
- **Operational Requirements** This section describes the requirements for certificate enrollment, issuance, and acceptance. It also addresses suspension, revocation, and the frequency of CRLs. Various security concerns are also covered, such as audit procedures, compromise and disaster recovery, and procedures for CA termination.
- **Physical, Procedural, and Personnel Security Controls** This section defines the nontechnical controls that are in place to provide for secure key generation, subject authentication, certificate issuance, certificate revocation, audit, and archiving. Such controls, for example, might include off-site record storage and background investigations of employees who fill trusted roles.
- **Technical Security Controls** This section describes the security measures taken by a CA to protect its private keys. Examples include where and how private keys are stored and who can activate and deactivate a private key.

- **Certificate and CRL Profile** This section specifies the format to be used for certificates and CRLs, the current versions supported, and the name forms used by the CA, the RA, and the end user. Also identified here are the supported certificate and CRL extensions and their criticality.
- **Specification Administration** This section specifies how this certificate policy definition or CPS will be maintained. Covered are change procedures for updating this statement, how it will be distributed, and the approval procedures for this and any new statement.

Summary

Although public-key technology solves many of the problems associated with symmetric-key technology, it presents a new set of distribution problems. The most widely accepted standard for public-key technology is the X.509 standard, which describes the format of public-key certificates to assist in the secure distribution of these keys. X.509 Version 3 certificates, for example, contain various fields and extensions that help govern their use.

A public-key infrastructure (PKI) plays an important role in the operation of public-key certificates. A PKI manages the collaboration between end users and relying parties, enabling the secure issuance and operation of these certificates. Certificate revocation and status checking are supported through the use of a CRL or the Online Certificate Status Protocol (OCSP), or both.

Certificate policies and certification practice statements provide end users and relying parties with information on which to base a decision to trust a given CA.

Real-World Examples

Various products are available that provide public-key infrastructure support, including developer toolkits, which assist individuals in creating their own public-key infrastructures, and companies, such as Verisign, that have based their business on providing certificates as a service. The following is a description of two PKI products developed by RSA Security, Inc.

Keon Certificate Server

The Keon certificate server is a fully functional CA\RA with all of the necessary tools to run a full CA. This server provides useful functionality, such as the One Step function. The One Step function actually allows the CA administrator to set up Keon programmatically so that as new employees are added to a human resource database, a certificate is generated and stored for use. This functionality takes a lot of the burden off end users and administrators.

Keon Web PassPort

Another advancement in the PKI arena is the new Keon Web PassPort. The Web PassPort provides roaming-certificate technology, which is similar to certificates discussed in the "Roaming Certificates" section earlier in this chapter. Through the use of a browser plug-in, a user can download the necessary private and public information to make use of PKI-enabled applications. A user may now, through the use of strong authentication and one small plug-in, make use of any computer system anywhere in the world.