

Computer Security

PRACTICAL UNIX SECURITY



Simson Garfinkel and Gene Spafford

O'Reilly & Associates, Inc.

Practical UNIX Security

Simson Garfinkel and Gene Spafford

O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472

Practical UNIX Security

by Simson Garfinkel and Gene Spafford

Copyright © 1991 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Editor: Deborah Russell

Printing History:

June 1991:	First printing.
September 1991:	Minor corrections.
April 1992:	Minor corrections.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly and Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book is printed on acid-free paper with 50% recycled content, 10-15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 0-937175-72-2

Encryption

Who Needs Encryption?

Cryptographic Strength

Types of Encryption Systems

ROT13

crypt

The Data Encryption Standard (DES)

RSA and Public Key Cryptography

Proprietary Encryption Systems

Protect Your Key

Secret messages, ciphers, keys, and code-breaking: these words all stir up images of James Bond and international espionage, high-tech communication equipment, and exploding briefcases. But encryption isn't reserved for life and death situations: it can play a very important role in your day-to-day computing and communicating. Not only can encryption protect information you send over telephone lines from eavesdropping, it can also protect the information stored on your computer from disclosure in the event that the computer is broken into or stolen. Encryption can also protect data stored on your computer from being tampered with without your knowledge.

Encryption works by transforming a message (called *plaintext*) into another message (called *ciphertext*) using a mathematical function and a special encryption password, called the *key*. Most encryption systems use the same key for both encryption and *decryption*, which is the process of converting the encrypted message back into plaintext.

Encryption keys are similar to other computer passwords: you need the key to get access to the information stored inside an encrypted file. But unlike a password program, an encryption program doesn't take the key you give it, compare it to

the key you used to encrypt the file, and give you access if the two keys match. Instead, an encryption program *uses* your key to transform the ciphertext back into the plaintext. If you provide the correct key, you get back your original message. If you type the wrong key, you get garbage.

Some encryption systems, or *encryption algorithms*, as they are properly known, use mathematical functions for the keys, but they are used the same way—you need to know the key function to decrypt the message.

This chapter explores the uses of encryption in the UNIX environment and discusses several different encryption systems that are in use. It does not, however, recommend particular encryption products. There are many available, and you'll have to choose what's best for your data and your budget.

Who Needs Encryption?

You do.

You should consider using encryption for every piece of sensitive information on your computer system, because there is no other way to assure absolutely the security of the information stored on your computer. Even if you assign eight-character passwords and you don't connect your computer to a modem or a network, somebody might bribe (or force) one of your users to share his or her account. Even if you are the only person who uses your computer, an attacker can still forcibly enter your office and steal your entire computer system—perhaps to resell it. It makes no difference. You simply can't assure that the data on your computer will *never* fall into the wrong hands. But if that data is protected with a strong encryption system, the data that falls into the wrong hands will be in an unusable form.

Information can be disclosed through accident, too. Bugs in software or protocols, or simple forgetfulness or error can leave your files open to outsiders. Having sensitive files encrypted may spell the difference between a minor mishap and complete disclosure.

The protection that encryption provides depends on the fact that encrypted data remains encrypted even after an attacker penetrates your system. Even if the superuser account is used so an attacker can read or write any file on the system, the attacker will not be able to decipher information that is securely encrypted. For this reason, encryption also protects your information against insiders who are abusing their authority (like the bribed system administrator).

Encryption also protects data from tampering and forgery. An attacker who becomes the superuser on your system can delete your encrypted files, but he can't change them without your knowledge. If the attacker changes the contents of a file, the file will no longer decrypt properly, and you will know that it has been altered. This sort of protection can be vital in business reports, where an extra zero at the end of a number can mean the difference between success and failure, and in correspondence, where a missing "not" can lead to a lawsuit.

If encryption is so good, why bother with other forms of security at all? Three reasons:

1. Encryption can't prevent an attacker from deleting your data altogether.
2. An attacker can compromise the encryption program itself. The attacker might conceivably modify the program to use a key different from the one you provide, or might record all of the encryption keys in a special file for later retrieval.
3. An attacker could access your file before it was encrypted or after it was decrypted.

Cryptographic Strength

The security of an encrypted file—also known as the *strength* of the cryptographic cipher—depends on a variety of factors:

- The secrecy of the key.
- The difficulty of guessing the key or trying out all possible keys (a *key search*).
- The difficulty of inverting the encryption algorithm without knowing the encryption key (*breaking* the encryption algorithm).
- The existence (or lack) of *back doors*, or additional ways by which an encrypted file can be decrypted without knowing the key.
- The ability to decrypt an entire encrypted message if you know the way that part of it decrypts (called a *known text* attack).
- The properties of the plaintext and knowledge of those properties by an attacker. (For example, an attacker may find it easier to decrypt your ciphertext if he knows that the message he's trying to read appears at the ciphertext's beginning and is repeated at the end.)

Only one type of encryption system is unbreakable: the so-called *one-time pad* mechanism. The one-time pad often makes use of the mathematical function known as exclusive OR (XOR). If you XOR a number with a second number, you get a third number. But if you XOR the third number with the second number again, you get back your starting number. That is:

```
if    C = A XOR B
then A = C XOR B = (A XOR B) XOR B
```

To encrypt a message with a one-time pad, you need the *pad*, a sequence of truly random bits at least as long as your message. This sequence is XOR-ed bit by bit with your plaintext, and the result is the encrypted message. To decrypt, you simply XOR the encrypted message with the pad again. For example, if your message (A) is the number 3 and the pad (B) is the number 5, then the encrypted message (C) is 6:

```
C = 3   XOR 5
C = 11  XOR 101
C = 110 = 6
```

To decrypt, you would apply the XOR function a second time to get the original message:

```
decrypted = 6   XOR 5
           = 110 XOR 101
           = 11  = 3 = A
```

To be secure, the pad sequence must not be used more than once. Furthermore, it must be completely and totally random—a mathematical random number generator will not do. (Most one-time pads are generated with machines based on nuclear radioactive decay, a truly random process.) To use a one-time pad system, you need two copies of the pad: one to encrypt the message, and the other to decrypt the message. Each copy must be destroyed after use; after all, if an attacker should ever obtain a copy of the pad, then any messages sent with it would be compromised.

One of the main uses of one-time pads is for sensitive diplomatic communications that are subject to heavy monitoring and intensive code-breaking attempts—such as the communication lines between the U.S. embassy in Moscow and Washington, D.C. However, the general use of one-time pads is limited because it is difficult to generate the sequence of random bits, and it is expensive to distribute it securely to both the sender and the recipient of the message.

Because of the problems associated with one-time pads, other kinds of algorithms are normally employed for computer encryption. These tend to be compact, mathematically-based functions. The mathematical functions are frequently used

to generate a pseudo-random sequence of numbers that are XORed with the original message—much the way the one-time pad works. The difference between the two techniques is that, with mathematical functions, it is always possible—in principle, at least—to translate any message encrypted with these methods back into the original message without knowing the key.

Translating the ciphertext back into the original message without the key is called breaking the encrypted message. Cryptographic systems are often said to be *weak* or *strong*: a message encrypted with a weak cryptographic system can be broken quickly, while a message encrypted with a strong system might require many years (even thousands or millions of years) to decode without the key, using known methods.

Types of Encryption Systems

There are two basic kinds of encryption systems in use today:

- **Private key encryption**, which uses the same key to encrypt and decrypt the message.
- **Public key encryption**, which uses one key to encrypt the message and another to decrypt it. The name “public key” comes from the fact that you can make one of the keys public without compromising the secrecy of the message or the other key.

Most average UNIX users have four encryption systems to choose from. They are:

- **Private key systems:**
 - ROT13
 - **crypt(1)**
 - DES
- **Public key systems:**
 - RSA

Of these systems, only DES and RSA are considered “strong” methods.

ROT13

ROT13 is a simple substitution cipher that is used for distributing potentially objectionable material on the Usenet, a worldwide bulletin board system. It is a variation on the Caesar Cipher—an encryption method used by Caesar's troops thousands of years ago.

In the ROT13 cipher, each letter of the alphabet is replaced with a letter that is 13 letters further along in the alphabet (with A following Z).

Letters encrypt as follows:

$A \rightarrow N, B \rightarrow O, C \rightarrow P, \dots, Z \rightarrow M$

ROT13 may very well be the most widely used encryption system in the UNIX world today. However, it is totally nonsecure. Many news and mail-reading programs automatically decrypt ROT13-encoded text with a single keystroke. Some people have been known to be able to decrypt ROT13 text without any machine assistance whatsoever.

Needless to say, do not use ROT13 as a means of protecting your files!

The only real use for this encryption method is the one to which it is put on the Usenet: to keep someone who does not want to be exposed to material (such as the answer to a riddle, or an obscene joke) from reading it inadvertently.

crypt

The program is a simplified simulation of the Enigma encryption machine described below. **crypt(1)** is an encryption program that is included as a standard part of the UNIX operating system.

It is a very simple encryption program that is easily broken, as evidenced by AT&T's uncharacteristic disclaimer on the reference page:

BUGS: There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, Bell Telephone Laboratories assumes no responsibility for their use by the recipient. Further, Bell Laboratories assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

—**crypt(1)** reference page

Enigma Encryption System

Enigma was an encryption system developed early this century in Germany by Arthur Scherbius and used throughout World War II. The Enigma encryption machine, shown in Figure 18-1, consisted of a battery, a pushbutton for every letter of the alphabet, a light for every letter of the alphabet, and a set of turnable discs called rotors.



Figure 18-1. An Enigma Machine

The Enigma machine was like a child's toy: pressing a button lit a different light. If you turned one of the rotors, the correspondence between buttons and lights changed.

The rotors were the key to the machine's cryptographic abilities. Each rotor on the Enigma machine was like a sandwich, with 52 metal contacts on each side. Inside the rotor, shown schematically in Figure 18-2, were 52 wires, each wire connecting a pair of contacts, one on either side of the rotor. But instead of directly connecting the contacts on one side with the other, the wires scrambled the order, so that, for example, contact #1 on the left might be connected with contact #15 on the right, and so on.

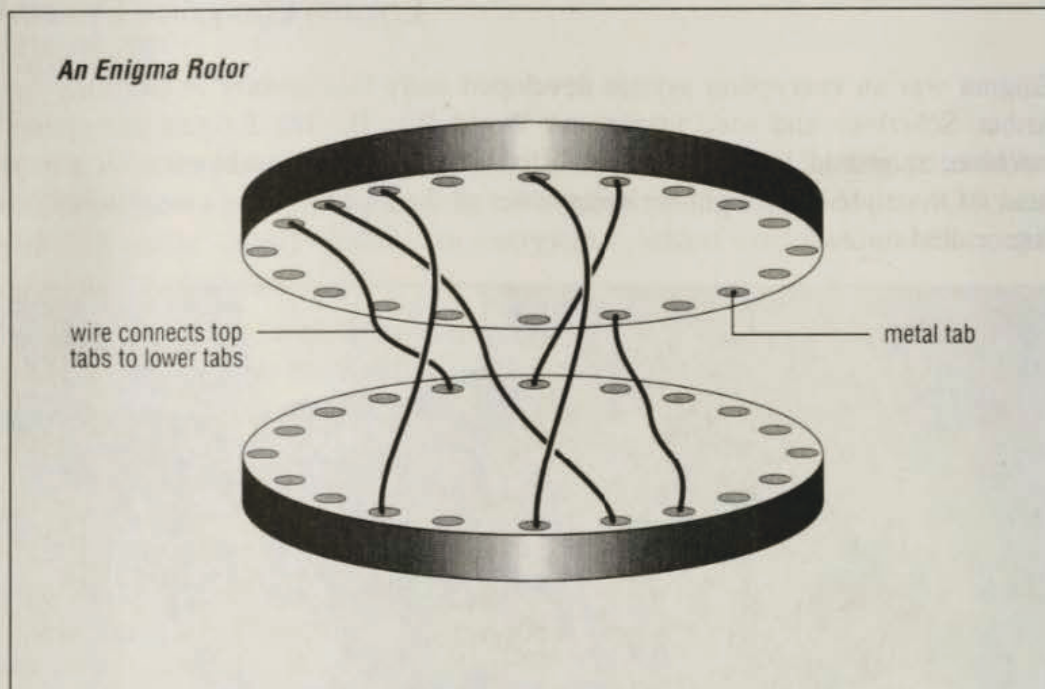


Figure 18-2. A Diagram of an Enigma Rotor

Enigma placed three of these rotors side by side. At the end of the row of rotors was a *reflector*, which sent the electrical signal back through the machine for a second pass. (Four rotors were used near the end of the war.) Half of the 52 contacts were connected with a pushbutton and the battery; the other half were connected with the lights. Each button closed a circuit, causing a light to brighten, but precisely *which* light brightened depended on the positioning of the three rotors and the reflector.

To encrypt or decrypt a message, a German code clerk would set the rotors to a specific starting position—the key. For each letter, the code clerk would then press the button, write down which letter lit, and then advance the rotors. Because the rotors were advanced after every letter, the same letter appearing twice in the plaintext would usually be encrypted to two different letters in the ciphertext. Enigma was thus a substitution cipher with a different set of substitutions for each letter in the message; these kinds of ciphers are called *polyalphabetic ciphers*. The letter Z was used to represent a space; numbers were spelled out.

UNIX crypt

Unlike Enigma, which had to encrypt only letters, **crypt**(1) must be able to encrypt any block of 8-bit data. As a result, the rotors used with crypt must have 256 connectors on each side. A second difference between Enigma and **crypt** is that, while Enigma used three or four rotors and a reflector, **crypt** uses just a single rotor and reflector. The encryption key provided by the user determines the placement of the virtual wires in the rotor and reflector.

Partially because it has but a single rotor, files encrypted with **crypt** are exceedingly easy for a cryptographer to break. For several years, it has been possible for noncryptographers to break messages encrypted with **crypt** as well, thanks to a program developed in 1986 by Robert Baldwin at the MIT Laboratory for Computer Science. Baldwin's program, Crypt Breaker's Workbench (**cbw**), automatically decrypts text files encrypted with **crypt** within a matter of minutes.

cbw breaks **crypt** by searching for arrangements of wires within the rotor that cause a file encrypted with **crypt** to decrypt into plain ASCII text. The task is considerably simpler than it may sound at first, because normal ASCII text uses only 127 of the possible 256 different code combinations (the ASCII codes 0 and 128 through 255 do not appear in normal text). Thus, most arrangements of the wires produce illegal characters when the file is decrypted; **cbw** automatically discards these arrangements.

cbw has been widely distributed; as a result, files encrypted with **crypt** should not be considered secure. (They weren't secure before **cbw** was distributed; it was simply that fewer people had the technical skill necessary to break them.)

Ways of Improving the Security of crypt

Although we recommend that you do not use **crypt** to encrypt files, you may have no other encryption system readily available to you. If this is the case, there are a few simple precautions that you can take to decrease the chances that your encrypted files will be decrypted:

- Encrypt the file multiple times, using different keys at each stage. This essentially changes the transformation.
- Compress your files before encrypting them. Compressing a file alters the information—the plain ASCII text—that programs like **cbw** use to know when they have correctly guessed part of the encryption key. If your message does not decrypt into plain text, **cbw** will not know when it has correctly decrypted your message. However, if your attackers know you have done this, they can modify their version of **cbw** accordingly.

- If you use **compress**(1) or **pack**(1) to compress your file, remove the 3-byte header. Files compressed with **compress** contain a 3-byte signature, or header, consisting of the hexadecimal values **1f, 9d** and **90** (in that order). If your attacker believes that your file was compressed before it was encrypted, knowing how the first three bytes decrypt can help him to decrypt the rest of the file. You can strip these three bytes with the **dd** command:*

```
% compress -c <plaintext | dd bs=3 skip=1 | crypt >encrypted
```

Of course, you must remember to put the 3-byte header back on before you attempt to uncompress the file:

```
% (compress -cf /dev/null; crypt <encrypted) | \
  uncompress -c >plaintext
```

- If you do not have **compress**, use **tar** to bundle your file to be encrypted with other files containing random data; then encrypt the **tar** file. The presence of random data will make it more difficult for decryption programs such as **cbw** to isolate your plaintext.

As encrypted files contain binary information, you must process them with **uencode** if you wish to email them.

Example

To compress, encrypt, uencode, and send a file with electronic mail:

```
% ls -l myfile
-rw-r--r-- 1 fred 166328 Nov 16 15:25 myfile
% compress myfile
% ls -l myfile.Z
-rw-r--r-- 1 fred 78535 Nov 16 15:25 myfile.Z
% dd if=myfile.Z of=myfile.Z.strip bs=3 skip=1
26177+1 records in
26177+1 records out
% crypt akey < myfile.Z.strip | uencode afile | mail spook@nsa.gov
```

To decrypt a file that you have received and saved in the **file** text file:

```
% head -3 file
begin 0600 afile
M?Z/#V3V,IGO!$](D!175:;S9_IU\A7K;:'LBB,8363R,T+/WZSOC4PQ,U/6Q
MX,T8&XZDQ1+[4$Y[*N4W@A3@9YM*4XV+U\ )X9NT.7@Z+W"WY^9-?(JRU,-4%
% udecode file
```

*Using **dd** this way is very slow and inefficient. If you are going to be encrypting a lot of compressed files, you may wish to write a small program to remove the header more efficiently.

```
% ls -l afile
-rw-r--r-- -l fred 78532 Nov 16 15:32 afile
% (compress -cf /dev/null; crypt < afile) | uncompress > myfile
```

myfile now contains the original file.

The Data Encryption Standard (DES)

One of the most widely used encryption systems today is the Data Encryption Standard (DES), developed in the 1970s and patented by researchers at IBM. The DES was an outgrowth of another IBM cipher known as LUCIFER. IBM made the DES available for public use, and the federal government issued Federal Information Processing Standard Publication (FIPS PUB) Number 46 in 1977 describing the system. Since that time, the DES has been reviewed and reaffirmed as a standard until at least 1993. It has also been adopted as an American National Standard (X3.92-1981/R1987).

The DES is basically a bit permutation, substitution, and recombination function performed on blocks of 64 bits of data and 56 bits of key (eight 7-bit characters). The 64 bits of input are permuted initially, and are then input to a function using static tables of permutations and substitutions. The bits are permuted in combination with 48 bits of the key in each round. This process is iterated 16 times (rounds), each time with a different set of tables and different bits from the key. The algorithm then performs a final permutation, and 64 bits of output are provided. The algorithm is structured in such a way that changing any bit in the input has a major effect on almost all of the output bits. Indeed, the output of the DES function appears so unrelated to its input that the function is sometimes used as a random number generator.

Although there is no standard UNIX program that performs encryption using the DES, Sun Microsystems' operating system does include such a program, called **des(1)**. (This command may not be present in international versions of the operating system.)

DES Modes

FIPS PUB 81 explains how the DES algorithm can be used in four modes:

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)

- Cipher Feedback (CFB)
- Output Feedback (OFB)

Each mode has particular advantages in some circumstances, such as when transmitting text over a noisy channel, or when it is necessary to decrypt only a portion of a file. The following provides a brief discussion of these four methods; consult FIPS PUB 81 or a good textbook on cryptography for details.

- **ECB Mode.** In electronic code book mode, each block of the input is enciphered using the key, and the output is written as a block. This method is simple encipherment of a message, a block at a time. This method may not indicate when portions of a message have been inserted or removed. It works well with noisy transmission channels—alteration of a few bits will corrupt only a single 64-bit block.
- **CBC Mode.** In CBC mode, the plaintext is first XOR'ed with the encrypted value of the previous block. Some known value is used for the first block. The result is then encrypted using the key. Because bits everywhere in the message propagate through the entire cipher, the last block can be used as a checksum/signature to check that the ciphertext has not been altered. Furthermore, long runs of repeated characters in the plaintext will be masked in the output. CBC mode is the default mode for Sun Microsystems' `des` program.
- **CFB Mode.** In CFB mode, the output is fed back into the mechanism. After each block is enciphered, part of it is shifted into a shift register. The contents of this shift register are encrypted with the user's key value using (effectively) ECB mode, and this output is XOR'd with the data stream to produce the encrypted result. This method is self-synchronizing, and enables the user to decipher just a portion of a large database by starting a fixed distance before the start of the desired data.
- **OFB Mode.** In OFB mode, the output is also fed back into the mechanism. A register is initialized with some known value. This register is then encrypted with (effectively) ECB mode using the user's key. The result of this is then used as the key to encipher the data block (using an XOR operation), and it is also stored back into the register for use on the next block.

Use and Export of DES

The DES is mandated as the encryption method to be used by all federal agencies in protecting sensitive but not classified information. The DES is also heavily used in many financial and communication exchanges. Many vendors make a DES chip that can encode or decode information fast enough to be used in data

encrypting modems or network interfaces. Note that the DES is not (and has never been) certified as an encryption method that can be used with U.S. Department of Defense classified material.

Additionally, export control rules restrict the export of hardware or software implementations of the DES, even though the algorithm has been widely published and implemented many times outside the United States. If you have the international version of UNIX, you may find that your system lacks a **des** command. If you find yourself in this position, don't worry; good implementations of the DES can be obtained from almost any archive service, including the Usenet **comp.sources** archives.

DES Strength

It is still not known exactly how cryptographically strong DES is. There has long been conjecture that the algorithm has an intentional built-in weakness that can be exploited by the National Security Agency. Many analysts believe that if the DES had longer keys, more rounds, or both, it would be a stronger algorithm. Nevertheless, the presence of a back door has never been proved. It may, in fact, be irrelevant; as hardware becomes cheaper, it appears that a special purpose computer could be designed to break a DES-encoded message using millions of chips operating in parallel. The cost of such a machine would be tremendous, but not beyond the capabilities of many governments and quite a few corporations.

The DES does appear to be strong in many ways that cryptographic strength is measured, and it is not easily susceptible to brute-force or known-text attacks. For the near future, the DES appears to be a reasonable method for use in encrypting your files, but remember that poor key selection makes the DES less secure than it could be. This, of course, is true of all encryption algorithms. Good key selection follows the same rules as good password selection.

Sun's des Command

The Sun implementation of the **des** command can use a hardware DES chip, if present, or a software implementation of the algorithm. As decryption and encryption are not identical, the user must specify an option to indicate if the input is being encrypted or decrypted.

Thus:

```
des -e <myfile | des -d >myfile2
```

will result in **myfile2** being identical to **myfile**.

The command, as used in the above example, will prompt for the key to be used in both the encryption and decryption. The user can avoid this by specifying the key on the command line:

```
des -k mykey -e <myfile | des -k mykey -d >myfile2
```

Specifying the key on the command line in shell files will compromise the security of the algorithm, and thus should never be done.

In the previous examples, the **des** command will use the hardware implementation of DES (if it exists), or else it will print a warning message and use the software implementation. The warning message can be suppressed with the **-f** option. The **-s** option forces the use of the software version with no warning message.

A **-b** option to the command selects Electronic Code Book (ECB) mode. The default is Cipher Block Chaining. As described elsewhere in this chapter, ECB mode encodes a block at a time, with identical input blocks encoding to identical output blocks. This will reveal if there is a pattern to the input. However, it also allows you to decrypt most of the file even if parts of it are corrupted or deleted. CBC mode hides repeated patterns, and results in a file that cannot be decrypted after any point of change or deletion.

Input and output file names are optional. If only one filename is given, it is assumed to be the input file.

A more complete example would therefore be:

```
des -e -k mykey -f myfile secret
des -d -s <secret >myfile2
Enter key: mykey
```

(The "Enter key:" prompt is from the program; the key is not echoed.)

Some export and other versions of the **des** command do not have the encryption algorithm included because of export control rules. If that happens, the command will print an error message explaining that the software version of **des** is not available.

RSA and Public Key Cryptography

Another popular encryption system is known as RSA, named after its inventors, R. L. Rivest, A. Shamir, and L. Adleman. RSA is a public key encryption system. Unlike private key systems, public key encryption uses two cryptographic keys: a *public key* and a *secret key*. The public key is used to encrypt a message and the secret key is used to decrypt it.

Note that the RSA encryption and decryption process is patented in the United States and is thus not available for general use in this country without a license. Because the algorithm was published before the patent was filed for, however, RSA may be used without royalty in Europe and Japan (which have slightly different laws covering prior disclosure and patent applicability). Not surprisingly, RSA is significantly more popular in Europe and Japan than in the United States, although its popularity in the U.S. is increasing.

How RSA Works

The strength of RSA is based on the difficulty of factoring a very large number. The following brief treatment does not fully explain the mathematical subtleties of the algorithm. If you are interested in more detail, you can consult the original paper* or a text such as those listed in Appendix E, *Other Sources*.

RSA is based on well-known number-theoretic properties of modular arithmetic and integers. One property makes use of the Euler Totient Function, $\phi(n)$. The Totient function of a number is defined as the number of integers less than that number that are *relatively prime* to it. (Two numbers are relatively prime if they have no common factors; for example, 9 and 8 are relatively prime.)

The property used by RSA is this: any integer i relatively prime to n raised to the power of $\phi(n)$ and taken mod n is equal to 1. That is:

$$i^{\phi(n)} \bmod n = 1$$

Suppose e and d are random integers that are inverses modulo $\phi(n)$, that is:

$$ed \equiv 1 \pmod{\phi(n)}$$

A related property used in RSA was also discovered by Euler. His theorem says that if M is any number relatively prime to n , then:

$$(M^e)^d \equiv M \pmod{n}$$

and:

$$(M^d)^e \equiv M \pmod{n}$$

Cryptographically speaking, if M is part of a message, we have a simple means for encoding it with one function:

$$s = M^e \pmod{n}$$

*Rivest, R., Shamir, A., Adleman, L., "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, Volume 21, Number 2, February 1978.

and decoding it with another function:

$$M = s^d \pmod{n}$$

So how do we get appropriate values for n , e , and d ? First, two large prime numbers p and q , of approximately the same size, are chosen, using some appropriate method. These numbers should be large—on the order of several hundred digits—and they should be kept secret.

Next, the Euler Totient function $\phi(pq)$ is calculated. In the case of n being the product of two primes, $\phi(pq) = (p-1)(q-1) = \phi(n)$.

Next, we pick a value e that is relatively prime to $\phi(n)$. A good choice would be to pick something in the interval $\max(p+1, q+1) < e < \phi(n)$. Then we calculate a corresponding d , such that $ed \equiv 1 \pmod{\phi(n)}$.^{*} That is, we find the modular inverse of $e \pmod{\phi(n)}$. If d should happen to be too small (i.e., less than about $\log_2(n)$), we pick another e and d .

Now we have our keys. To encrypt a message m , we split m into fixed-size integers M less than n . Then we find the value $(M^e) \pmod{n} = s$ for each portion of the message. This can be done quickly in hardware, or in software using special algorithms. These values are concatenated to form the encrypted message. To decrypt the message, it is split into the blocks, and each block is decrypted as $(s^d) \pmod{n} = M$.

An RSA Example

For this example, assume we pick two primes p and q :

$$p = 251 \text{ and } q = 269$$

The number n is therefore:

$$n = 251 \times 269 = 67519$$

The Euler Totient function for this is:

$$\phi(n) = (251-1)(269-1) = 67000$$

Let's arbitrarily pick e as 50253. d is then:

$$d = \text{inv}(e) \pmod{67000} = 27917$$

because:

$$50253 \times 27917 = 1402913001 = 20939 \times 67000 + 1 \equiv 1 \pmod{67000}$$

^{*} d is calculated with a straightforward algorithm based on Euclid's work.

Using $n = 67519$ allows us to encode any message M that is between 0 and 67518. We can therefore use this system to encode a text message two characters at a time. (Two characters have 16 bits, or 65536 possibilities.)

Using e as our key, let's encode the message "RSA works!" The sequence of ASCII characters encoding "RSA works!" is shown in the following table.

ASCII	Decimal Value	Encoded Value
"RS"	21075	48467
"A "	16672	14579
"wo"	30575	26195
"rk"	29291	58004
"s!"	29473	30141

As you can see, the encoded values do not resemble the original message.

To decrypt, we raise each of these numbers to the power of d and take the remainder mod n . After translating back to ASCII, we get back the original message.

Strength of RSA

The numbers n and either e or d can be disclosed without seriously compromising the strength of an RSA cipher. For an attacker to be able to break the encryption, it would be necessary to find $\phi(n)$, and that requires factoring n .

Factoring large numbers is very difficult—no known method can be used efficiently. The time required to factor a number can be several hundred years or several billion years with the fastest computers, depending on how large the number n is. If n is large enough, it is, for all intents and purposes, unfactorable. The RSA encryption system is therefore quite strong, provided that appropriate values of n , e , and d are chosen to begin with, and that they are kept secret.

To see how difficult it is to factor a large number, let's do a little rough calculation of how long it might take to factor a 200 decimal digit number.

All 200 digit values can be represented in at most 665 binary bits. (In general, 2^x has $\lfloor x \log_{10}(2) \rfloor + 1$ decimal digits.)

To factor a 665-bit number, using one of the fastest known factoring algorithms, would require approximately 1.2×10^{23} operations.

Let's assume you have a machine that will do 10 billion (10^{10}) operations per second. (A little faster than today's fastest computers.) To perform 1.2×10^{23} operations would require 1.2×10^{13} seconds, or 380,267 years worth of computer time. If you feel uneasy about having your number factored in 380,267 years, simply

double the size of your prime number: a 400-digit number would require a mere 8.6×10^{15} years to factor. This is probably long enough; according to Stephen Hawking's *A Brief History of Time*, the universe itself is only about 2×10^{10} years old.

To give you another perspective on the size of these numbers, assume that you (somehow) could precalculate the factors of all 200 decimal digit numbers. Just to store the unfactored numbers themselves would require approximately $(9 \times 10^{200}) \times 665$ bits of storage (not including any overhead or indexing). Assume you can store these on special media that hold 100GB (100×1024^4 , or approximately 1.1×10^{14}) of storage. You would need about 6.12×10^{189} of these disks.

Now assume that each of those disks is only one millionth of a gram in weight (1 pound is 453.59 grams). The weight of all your storage would come to over 6.75×10^{177} tons of disk. The planet Earth weighs only 6.588×10^{21} tons. The Chandrasekhar limit, the amount of mass at which a star will collapse into a black hole, is about 1.5 times the mass of our Sun, or approximately 3.29×10^{27} tons. Thus, your storage, left to itself, would collapse into a black hole from which your factoring could not escape! We are not sure how much mass is in our local galaxy, but we suspect it might be less than the amount you'd need for your storage.

Again, it looks fairly certain that without a major breakthrough in number theory, the RSA mechanism (and similar methods) are almost assuredly safe from brute-force attacks.

Proprietary Encryption Systems

A number of companies have invented their own "proprietary" encryption systems that can be used in place of DES or RSA. Although some proprietary systems are relatively secure, the vast majority are not. To make matters worse, it is nearly impossible to tell which are safe and which are not—especially if the company selling the encryption program refuses to publish the details of the algorithm.

A standard tenet in data encryption is that the security of the system should depend completely on the security of the encryption key. When choosing an encryption system, rely on formal mathematical proofs of security, rather than secret proprietary systems. If the vendor of an encryption algorithm or technology will not disclose the algorithm and show how it has been analyzed to show its strength, you are probably better off avoiding it.

Protect Your Key

No matter how secure your encryption system is, you should take the same precautions with your encryption key that you take with your password: there is no sense in going to the time and expense of encrypting all of your data with strong ciphers like DES or RSA if you keep your encryption keys in a file in your home directory, or write them on your terminal.

Finally, *never use any of your passwords as an encryption key!* If an attacker learns your password, your encryption key will be the only protection for your data. Likewise, if the encryption program is weak or compromised, you do not want your attacker to learn your password by decrypting your files. The only way to do this is by using different words for your password and encryption keys.