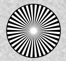
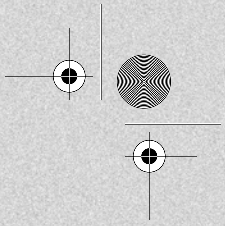




FOUR

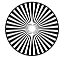
Asymmetric Cryptography

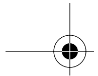
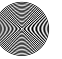
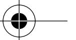
Modern computing has generated a tremendous need for convenient, manageable encryption technologies. Symmetric algorithms, such as Triple DES and Rijndael, provide efficient and powerful cryptographic solutions, especially for encrypting bulk data. However, under certain circumstances, symmetric algorithms can come up short in two important respects: key exchange and trust. In this chapter we consider these two shortcomings and learn how asymmetric algorithms solve them. We then look at how asymmetric algorithms work at a conceptual level in the general case, with emphasis on the concept of trapdoor one-way functions. This is followed by a more detailed analysis of RSA, which is currently the most popular asymmetric algorithm. Finally, we see how to use RSA in a typical program using the appropriate .NET Security Framework classes.

We focus on the basic idea of asymmetric algorithms, and we look at RSA in particular from the encryption/decryption point of view. In Chapter 5 we explore using the RSA and DSA asymmetric algorithms as they relate to authentication and integrity checking, involving a technology known as digital signatures. For a more thorough discussion of RSA from a mathematical point of view, please see Appendix B.

Problems with Symmetric Algorithms

One big issue with using symmetric algorithms is the key exchange problem, which can present a classic catch-22. The other main issue is the problem of trust between two parties that share a secret symmetric key. Problems of trust



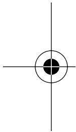
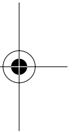
Thorsteinson.book Page 100 Tuesday, July 29, 2003 1:53 PM

may be encountered when encryption is used for authentication and integrity checking. As we saw in Chapter 3, a symmetric key can be used to verify the identity of the other communicating party, but as we will now see, this requires that one party trust the other.

The Key Exchange Problem

The key exchange problem arises from the fact that communicating parties must somehow share a secret key before any secure communication can be initiated, and both parties must then ensure that the key remains secret. Of course, direct key exchange is not always feasible due to risk, inconvenience, and cost factors. The catch-22 analogy refers to the question of how to securely communicate a shared key before any secure communication can be initiated.

In some situations, direct key exchange is possible; however, much commercial data exchange now takes place between parties that have never previously communicated with one another, and there is no opportunity to exchange keys in advance. These parties generally do not know one another sufficiently to establish the required trust (a problem described in the next section) to use symmetric algorithms for authentication purposes either. With the explosive growth of the Internet, it is now very often a requirement that parties who have never previously communicated be able to spontaneously communicate with each other in a secure and authenticated manner. Fortunately, this issue can be dealt with effectively by using asymmetric algorithms.[1]

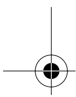
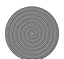
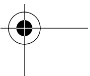
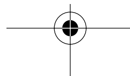
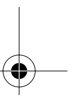
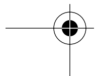
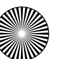



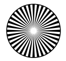
The Trust Problem

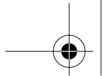
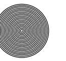
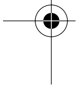
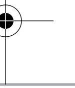
Ensuring the integrity of received data and verifying the identity of the source of that data can be very important. For example, if the data happens to be a contract or a financial transaction, much may be at stake. To varying degrees, these issues can even be legally important for ordinary email correspondence, since criminal investigations often center around who knew what and when they knew it. A symmetric key can be used to check the identity of the individual who originated a particular set of data, but this authentication scheme can encounter some thorny problems involving trust.

---

1. Asymmetric algorithms are also known as public key algorithms, which can be misleading, since there are actually two keys involved; one is public, and the other is private. The term *public key algorithm* is intended to contrast with the idea of symmetric algorithms, where there is no public key but rather only a single secret key.

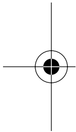
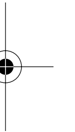










Thorsteinson.book Page 101 Tuesday, July 29, 2003 1:53 PM





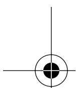
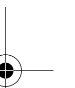
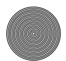
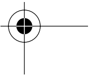
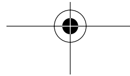
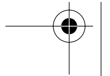
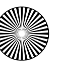
As you may recall from Chapter 3, in this technique the data is hashed, and the resulting hash is encrypted using a shared secret key with a symmetric algorithm. The recipient, who also knows the secret key, is sent the data along with the encrypted hash value. The recipient then decrypts the hash using the shared key, and the result is verified against a fresh recalculation of the hash value on the data received. This works because only someone who knows the secret key is capable of correctly encrypting the hash of the original data such that it will match the recalculated hash value computed by the recipient. This verifies the identity of the data source. As an added bonus, this technique verifies data integrity in that any individual who is ignorant of the secret key could not have tampered with the data.

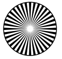
This is great if you have the luxury of establishing the shared secret beforehand, but there is an additional problem here. What if you cannot trust the other party with whom you have shared the secret key? The problem is that this scheme cannot discriminate between the two individuals who know the shared key. For example, your pen pal may fraudulently send messages using your shared key, pretending to be you. This would allow your friend to write IOUs to himself in your name, making this scheme useless in any trust-lacking relationship. Other problems could arise if your partner shared the secret key with others without telling you about it. Suddenly, you would have no leg to stand on if certain disputes were to arise. For example, your partner could renege on a contract by claiming that someone else must have obtained the key from you and signed off on a deal in his name. This problem is known as repudiation,[2] and we often need a way to enforce nonrepudiation between untrusting parties. The basic problem with all this is that any symmetric algorithm scheme requires that one party can safely trust the other party, which often is not realistic.

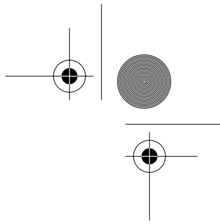



Fortunately, asymmetric algorithms can be used to solve these problems by performing the same basic operations but encrypting the hash using a private key (belonging to an asymmetric key pair) that one individual and only one individual knows. Then anyone can use the associated public key to verify the hash. This effectively eliminates the problems of trust and repudiation.[3]

2. The word *repudiation* means refusal to acknowledge a contract or debt. You will frequently encounter its antonym, nonrepudiation, in discussions on digital signatures.
3. Actually, asymmetric algorithms cannot solve all these problems entirely on their own. For a complete solution, we generally need to resort to enlisting the help of a trusted third party, known as a certificate authority, who takes on the formal responsibility of verifying and vouching for the identities of its clients. For the full story on how asymmetric algorithms and certificate authorities together solve these problems, please see Chapter 5.

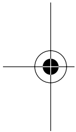
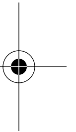


Thorsteinson.book Page 102 Tuesday, July 29, 2003 1:53 PM


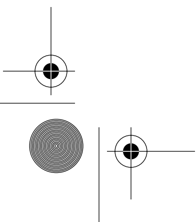
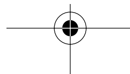
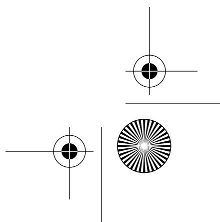
This technique is called a digital signature, which is the main topic of the next chapter.

The Idea Behind Asymmetric Cryptography

In the 1970s Martin Hellman, Whitfield Diffie, and, independently, Ralph Merkle invented a beautiful cryptographic idea. Their idea was to solve the key exchange and trust problems of symmetric cryptography by replacing the single shared secret key with a pair of mathematically related keys, one of which can be made publicly available and another that must be kept secret by the individual who generated the key pair. The advantages are obvious. First, no key agreement is required in advance, since the only key that needs to be shared with the other party is a public key that can be safely shared with everyone. Second, whereas the security of a symmetric algorithm depends on two parties successfully keeping a key secret, an asymmetric algorithm requires only the party that generated it to keep it secret. This is clearly much less problematic. Third, the issue of trusting the other party disappears in many scenarios, since without knowledge of your secret key, that party cannot do certain evil deeds, such as digitally sign a document with your private key or divulge your secret key to others.

Asymmetric cryptography does not replace symmetric cryptography. Rather, it is important to recognize the relative strengths and weaknesses of both techniques so that they can be used appropriately and in a complementary manner. Symmetric algorithms tend to be much faster than asymmetric algorithms, especially for bulk data encryption. They also provide much greater security than asymmetric algorithms for a given key size. On the down side, symmetric key cryptography requires that the secret key be securely exchanged and then remain secret at both ends. In a large network using symmetric encryption many key pairs will proliferate, all of which must be securely managed. Because the secret key is exchanged and stored in more than one place, the symmetric key must be changed frequently, perhaps even on a per-session basis. Finally, although symmetric keys can be used for message authentication in the form of a keyed secure hash, the full functionality of a digital signature requires asymmetric encryption techniques, such as RSA or DSA. As we shall see in the next chapter, a symmetric keyed secure hash algorithm can be used to implement a MAC (Message Authentication Code), which provides authentication and integrity but not nonrepudiation. In contrast, asymmetric digital signature algorithms provide authentication, integrity, and nonrepudiation, and enable the services of certificate authorities (CAs).

Thorsteinson.book Page 103 Tuesday, July 29, 2003 1:53 PM

Using Asymmetric Cryptography

To use asymmetric cryptography, Bob randomly generates a public/private key pair.[4] He allows everyone access to the public key, including Alice. Then, when Alice has some secret information that she would like to send to Bob, she encrypts the data using an appropriate asymmetric algorithm and the public key generated by Bob. She then sends the resulting ciphertext to Bob. Anyone who does not know the matching secret key will have an enormously difficult time retrieving the plaintext from this ciphertext, but since Bob has the matching secret key (i.e., the trapdoor information), Bob can very easily discover the original plaintext. Figure 4–1 shows how asymmetric cryptography is used.

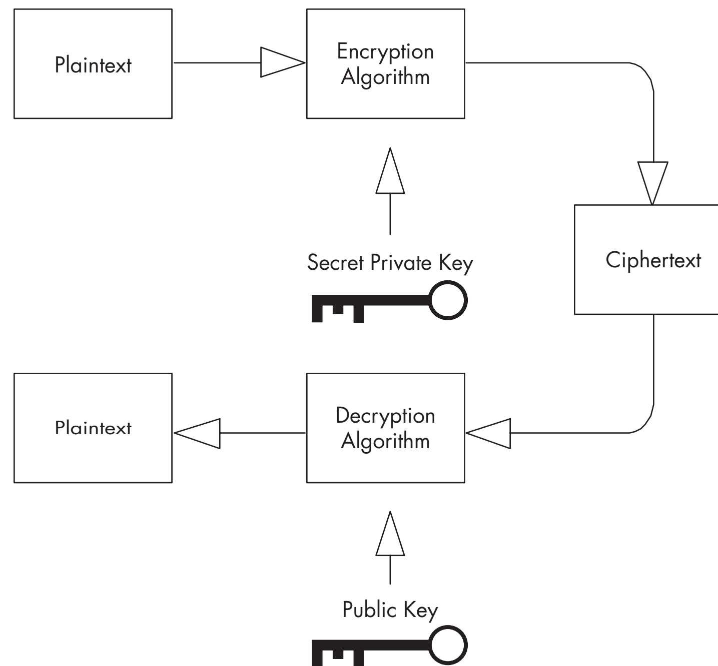
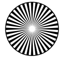
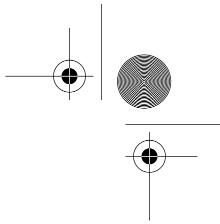


FIGURE 4–1 *How asymmetric cryptography is used.*

4. In real life this key pair is generated automatically by a cryptographic application, and the user is typically oblivious to this fact. For example, Microsoft Outlook generates such a key pair, using the underlying Windows CSP (cryptographic service provider), for encrypting and digitally signing secure email messages. PGP (Pretty Good Privacy), which is a freeware tool for secure messaging and data storage, works in a similar manner, but it generates its own keys and works on multiple platforms.



Thorsteinson.book Page 104 Tuesday, July 29, 2003 1:53 PM


The Combination Lock Analogy

A traditional symmetric cipher is analogous to a lockbox with a combination lock that has one combination used both to open it and close it.[5] The analogy for an asymmetric cipher is a somewhat stranger device: The single lock has two distinct combinations, one for opening it and another for closing it. By keeping one of these combinations secret and making the other combination public, you can effectively control who can place or remove the contents in the lockbox. This added flexibility supports two useful scenarios: confidentiality without prior key exchange and data integrity enforcement.

CONFIDENTIALITY WITHOUT PRIOR KEY EXCHANGE

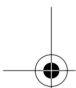
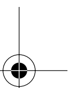
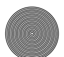
Here is the first scenario. If you know the public combination for closing the lock but not the private combination for opening the lock, then once you have placed something into the box and locked it, it becomes impossible for anybody who does not know the private opening combination[6] to obtain the contents. This demonstrates spontaneous confidentiality (i.e., keeping a secret without prior key exchange). Hence, we have a solution to the key exchange problem described earlier.

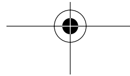
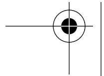
ENFORCING DATA INTEGRITY

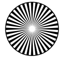
The other scenario is if only you know the private combination for closing the lock, and you place contents into the lockbox and then lock it. Then anyone can open the lock, but nobody else can lock other contents into the lockbox, since nobody else knows the private combination for closing the lock. Therefore, nobody else can tamper with its contents and then close the lock again. You might think that this is easy to defeat, since anyone could easily create his or her own key pair and then lock any data into the lockbox. However, only the newly created public key would then work, and the original public key would fail to unlock the lockbox. Therefore, anyone with knowledge of the original public key would not be fooled by such an attack. Since tampering is detectable, this scenario demonstrates how data integrity can be enforced.

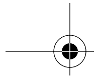
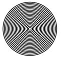
---

5. You have to use your imagination just a bit here. Most physical combination locks automatically lock simply by closing them. In this analogy the lock is a bit special in that it requires knowledge of the combination to lock it closed.
6. Even you cannot open it once you have locked it if you do not know the private unlocking combination. Since you placed the contents into the box in the first place, you implicitly know its contents, so this is not often an issue, but this subtle point can occasionally be significant.



Thorsteinson.book Page 105 Tuesday, July 29, 2003 1:53 PM

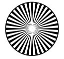
Trapdoor One-Way Functions

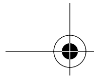
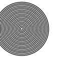
There are several asymmetric algorithms that make use of this great idea, but all these algorithms have certain mathematical characteristics in common. In each case an asymmetric algorithm is based on a type of function first suggested by Diffie and Hellman that has special properties known as *trapdoor one-way functions*. A trapdoor one-way function, if given some additional secret information, allows much easier computation of its inverse function.

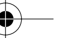
Before discussing what a trapdoor one-way function is, let's look at the broader class of one-way functions.[7] A one-way function is a mathematical function that is highly asymmetric in terms of its computational complexity with respect to its inverse function. Such a function is easy to compute in the forward direction but diabolically difficult to compute in the inverse direction. Such functions are based on *hard* mathematical problems, such as factoring large composites into prime factors, the discrete log problem, and the knapsack problem. The inherent difficulty of such problems falls under the branch of mathematics known as complexity theory, which is beyond the scope of this book. Suffice it to say that the difficulty of these problems grows rapidly in relation to the magnitude of the numbers (which correspond to the key size used in the corresponding cryptographic algorithm) involved.

While finding a solution to the mathematical problem on which a one-way function is based, it is very easy to test if a proposed solution is correct. Imagine yourself working on a large jigsaw puzzle, and you want to find the piece that belongs in a particular position. It might take you a long time to find the correct piece, but at any time that you believe you have good candidate, it takes very little effort to test it out. For a mathematical example, consider the problem of finding all of the prime[8] factors of a large composite number. This is believed to be very difficult,[9] but if you are given the prime factors, testing

7. One-way functions are widely believed to exist, and several presumed one-way functions are used heavily in cryptography. Unfortunately, we currently have no formal proofs that they actually do exist!
8. A prime number is a positive integer greater than one that is evenly divisible only by one and itself, such as 2, 3, 5, 7, 11, and so on. A composite number is a positive integer greater than one that is not prime and thus has devisors other than one and itself, such as 4, 6, 8, 9, 10, and so on.
9. Note that this problem is *believed* to be difficult. That is, nobody has yet publicly demonstrated a fast technique for solving this problem, and much evidence indicates that it is very hard. Currently, this is an educated guess, not a proof. But without any hard proof, how confident can we really be? Mathematics is full of examples of clever folks finding devious ways of easily solving problems that appeared to be difficult. According to a story about J. C. F. Gauss, when he was only seven years old, his teacher asked his class to add the integers from 1 to 100. The other children took a very long time, and most did not produce the correct answer. Gauss had the solution 5050 in the blink of an eye, since he realized that it was the sum of 50 pairs, where each pair is equal to 101, or 50*101, which he could do in his head faster than he could write it down.

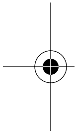
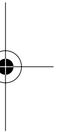


Thorsteinson.book Page 106 Tuesday, July 29, 2003 1:53 PM





them is a simple matter of multiplying them together and comparing the result with the original large composite number. To convince yourself of this, imagine trying to find all the prime factors of the composite number 3431 with only pencil and paper to work with. Without the aid of a computer, most people would take several hours to do this. However, if you were asked to test the candidate solution of 47 and 73, you would probably take only a few seconds to show that indeed they do multiply to produce the resulting value of 3431. Now consider that the number 3431 can be represented with a mere 12 bits. What if you tried this experiment with a 1024-bit number? Finding the prime factors is a problem that grows exponentially with bit size (think in terms of billions or trillions of years on a supercomputer), whereas testing a solution grows at a very moderate rate with bit size (microseconds or milliseconds on a personal computer).

The term one-way function is slightly misleading, since by definition, a cipher can only be based on a mathematical function that is invertible, meaning that its inverse function does indeed exist. After all, if a cipher were based on a function that had no inverse, you could encrypt data, but then you could not reliably decrypt that data. However, in practice, the inverse of a one-way function is so difficult to compute that the mere fact that the inverse function exists is of no help whatsoever in computing its value. Such a function is theoretically two-way, but, in practical terms, it is effectively one-way.

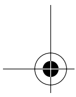
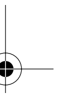
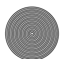
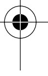
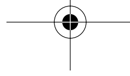
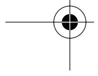
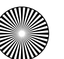



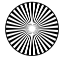
Now let's add in the idea of a trapdoor. A trapdoor turns something that is normally very difficult into something that is very easy, provided that you know a powerful little secret. This is analogous to a haunted house where the bookshelf revolves, opening a hidden passageway, but you first need to know that it exists and that the secret to opening it is to pull on the candelabra three times. A one-way function that has the additional property that its inverse function suddenly becomes very easy to compute provided an additional piece of secret information is known (i.e., a private key) becomes a trapdoor one-way function. You can think of a trapdoor one-way function as a haunted mathematical function if you like.

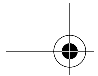
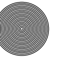
There are many one-way functions to choose from in mathematics, but finding one that allows you to incorporate the all-important backdoor is not so easy. A few candidates have been discovered that are convenient and appear to be secure. Finding those algorithms that are also efficient enough to be practical and use keys of a reasonable size reduces the candidates further. Examples of successful trapdoor one-way functions are the discrete log problem, which forms the basis of the DSA algorithm, and the factoring of large composites into prime factors, which forms the basis of the RSA algorithm.

Advantages of the Asymmetric Approach

With the asymmetric (also known as public key) approach, only the private key must be kept secret, and that secret needs to be kept only by one party.



Thorsteinson.book Page 107 Tuesday, July 29, 2003 1:53 PM

This is a big improvement in many situations, especially if the parties have no previous contact with one another. However, for this to work, the authenticity of the corresponding public key must typically be guaranteed somehow by a trusted third party, such as a CA. Because the private key needs to be kept only by one party, it never needs to be transmitted over any potentially compromised networks. Therefore, in many cases an asymmetric key pair may remain unchanged over many sessions or perhaps even over several years. Another benefit of public key schemes is that they generally can be used to implement digital signature schemes that include nonrepudiation. Finally, because one key pair is associated with one party, even on a large network, the total number of required keys is much smaller than in the symmetric case.

Combining Asymmetric and Symmetric Algorithms

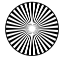
Since there is no secret key exchange required in order to use asymmetric algorithms, you might be tempted to solve the symmetric key exchange problem by simply replacing the symmetric algorithm with an asymmetric algorithm. However, that would be like throwing the baby out with the bath water. We still want to take advantage of the superior speed and security offered by symmetric algorithms, so, instead, we actually combine the two (and sometimes more than two) algorithms.

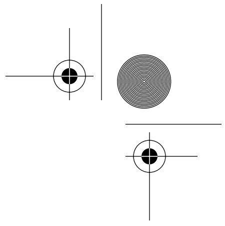
For example, Microsoft Outlook and Netscape Communicator implement secure email using the S/MIME (Secure/Multipurpose Internet Mail Extensions) specification. S/MIME is an IETF standard that supports both digital signatures for authentication and encryption for privacy. S/MIME provides bulk message data encryption using any of several symmetric algorithms, including DES, 3DES, and RC2. Secure symmetric key exchange and digital signatures are provided by the RSA asymmetric algorithm as well as either of the MD5 or SHA-1 hash algorithms.

As another example, the popular PGP software (developed by Philip Zimmermann) provides cryptographic services for email and file storage by combining several algorithms to implement useful cryptographic protocols.[10] In this way, message encryption and digital signatures are provided to email clients using an assortment of selected symmetric, asymmetric, and hash algorithms. RSA or ElGamal are used for PGP session key transport. 3DES is one of several alternatives used for bulk PGP message encryption. PGP digital signatures use either RSA or DSA for signing and MD5 or SHA-1 for generating message digests.

There are several other protocols that are built in a hybrid manner by combining asymmetric and symmetric algorithms, including IPSec (IP Security

10. For details, see the OpenPGP Message Format RFC 2440 at *http://www.ietf.org/rfc/rfc2440.txt*.

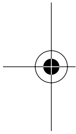
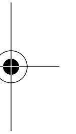


Thorsteinson.book Page 108 Tuesday, July 29, 2003 1:53 PM


Protocol) and SSL (Secure Sockets Layer). IPSec is an IETF standard that provides authentication, integrity, and privacy services at the datagram layer, allowing the construction of virtual private networks (VPNs). The SSL protocol, developed by Netscape, provides authentication and privacy over the Internet, especially for HTTP (Hypertext Transfer Protocol).

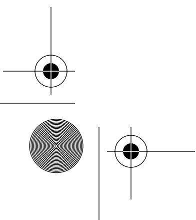
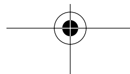
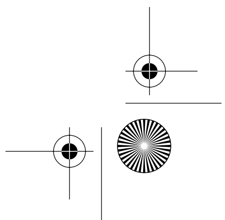
Existing Asymmetric Algorithms

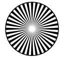
Recall that the only information that needs to be shared before initiating symmetric encryption is the secret key. Since this key is typically very small (typically no greater than 256 bits) compared to the bulk data (which could be megabytes) that must be encrypted, it makes sense to use the asymmetric algorithm to encrypt only the secret symmetric key, and then use this symmetric key for encrypting the arbitrarily large bulk message data. The secret symmetric key is often referred to as a *session key* in this scenario.

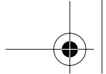
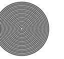



There are several asymmetric algorithms in existence today, including RSA, DSA, ElGamal, and ECC. Currently, the most popular is RSA, which stands for Rivest, Shamir, and Adelman, the names of its inventors. RSA is based on the problem of factoring large composite numbers into prime factors. RSA can be used for confidentiality or symmetric key exchange as well as for digital signatures. DSA, which was proposed by NIST in 1991, stands for Digital Signature Algorithm. DSA is somewhat less flexible, since it can be used for digital signatures but not for confidentiality or symmetric key exchange. The ElGamal algorithm, which was invented by Taher ElGamal, is based on the problem of calculating the discrete logarithm in a finite field. EEC stands for Elliptic Curve Cryptography, which was independently proposed in 1985 by Neal Koblitz and V. S. Miller. EEC is not actually an algorithm, but an alternate algebraic system for implementing algorithms, such as DSA, using peculiar mathematical objects known as elliptic curves over finite fields. ElGamal and ECC are not currently supported by .NET out of the box; however, the .NET Framework has been designed to be extensible, making it possible for you or other vendors to provide implementations.

Some asymmetric algorithms, such as RSA and ElGamal, can be used for both encryption and digital signatures. Other asymmetric algorithms, such as DSA, are useful only for implementing digital signatures. It is also generally true that asymmetric algorithms tend to be much slower and less secure than symmetric algorithms for a comparable key size. To be effective, asymmetric algorithms should be used with a larger key size, and, to achieve acceptable performance, they are most applicable to small data sizes. Therefore, asymmetric algorithms are usually used to encrypt hash values and symmetric session keys, both of which tend to be rather small in size compared to typical plaintext data.



Thorsteinson.book Page 109 Tuesday, July 29, 2003 1:53 PM



RSA: The Most Used Asymmetric Algorithm

The most common asymmetric cipher currently in use is RSA, which is fully supported by the .NET Security Framework. Ron Rivest, Adi Shamir, and Leonard Adleman invented the RSA cipher in 1978 in response to the ideas proposed by Hellman, Diffie, and Merkel. Later in this chapter, we shall see how to use the high-level implementation of RSA provided by the .NET Security Framework. But first, let's look at how RSA works at a conceptual level.

Underpinnings of RSA

Understanding the underpinnings of RSA will help you to develop a deeper appreciation of how it works. In this discussion we focus on the concepts of RSA, and in Appendix B we look at two examples of implementing RSA from scratch. One of these examples is **TinyRSA**, which is a toy version that limits its arithmetic to 32-bit integers, and the other is a more realistic, multiprecision implementation named **BigRSA**. You will probably never implement your own RSA algorithm from scratch, since most cryptographic libraries, including the .NET Security Framework, provide excellent implementations (i.e., probably better than I could do). However, the RSA examples in Appendix B should help you to fully understand what goes on in RSA at a deeper level.

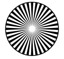
Here is how RSA works. First, we randomly generate a public and private key pair. As is always the case in cryptography, it is very important to generate keys in the most random and therefore, unpredictable manner possible. Then, we encrypt the data with the public key, using the RSA algorithm. Finally, we decrypt the encrypted data with the private key and verify that it worked by comparing the result with the original data. Note that we are encrypting with the public key and decrypting with the private key. This achieves confidentiality. In the next chapter, we look at the flip side of this approach, encrypting with the private key and decrypting with the public key, to achieve authentication and integrity checking.

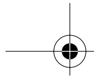
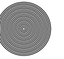
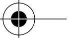
Here are the steps for generating the public and private key pair.

1. Randomly select two prime numbers p and q. For the algebra to work properly, these two primes must not be equal. To make the cipher strong, these prime numbers should be large, and they should be in the form of arbitrary precision integers with a size of at least 1024 bits.[11]
2. Calculate the product: $n = p \cdot q$.

---

11. In practice, there are other concerns when choosing prime p and q. For example, even if they are large, it turns out to be easy to factor the product if the difference between p and q is small, using a technique known as Fermat's factorization algorithm.



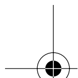
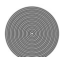
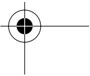
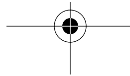
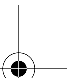
3. Calculate the Euler totient[12] for these two primes, which is represented by the Greek letter ϕ. This is easily computed with the formula $\phi = (p - 1) \cdot (q - 1)$.
4. Now that we have the values n and ϕ, the values p and q will no longer be useful to us. However, we must ensure that nobody else will ever be able to discover these values. Destroy them, leaving no trace behind so that they cannot be used against us in the future. Otherwise, it will be very easy for an attacker to reconstruct our key pair and decipher our ciphertext.
5. Randomly select a number e (the letter e is used because we will use this value during encryption) that is greater than 1, less than ϕ, and relatively prime to ϕ. Two numbers are said to be relatively prime if they have no prime factors in common. Note that e does not necessarily have to be prime. The value of e is used along with the value n to represent the public key used for encryption.
6. Calculate the unique value d (to be used during decryption) that satisfies the requirement that, if $d \cdot e$ is divided by ϕ, then the remainder of the division is 1. The mathematical notation for this is $d \cdot e = 1(\text{mod } \phi)$. In mathematical jargon, we say that d is the multiplicative inverse of e modulo ϕ. The value of d is to be kept secret. If you know the value of ϕ, the value of d can be easily obtained from e using a technique known as the Euclidean algorithm. If you know n (which is public), but not p or q (which have been destroyed), then the value of ϕ is very hard to determine. The secret value of d together with the value n represents the private key.

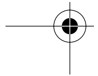
Once we have generated a public/private key pair, we can encrypt a message with the public key with the following steps.

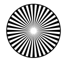
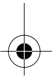
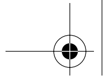
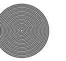
1. Take a positive integer m to represent a piece of plaintext message. In order for the algebra to work properly, the value of m must be less than the modulus n, which was originally computed as $p \cdot q$. Long messages must therefore be broken into small enough pieces that each piece can be uniquely represented by an integer of this bit size, and each piece is then individually encrypted.
2. Calculate the ciphertext c using the public key containing e and n. This is calculated using the equation $c = m^e(\text{mod } n)$.

---

12. The Euler totient, symbolized with the Greek letter phi, represents the number of positive integers less than or equal to n that are relatively prime to n (i.e., have no prime factors in common with n). One is considered to be relatively prime with all integers.




Thorsteinson.book Page 111 Tuesday, July 29, 2003 1:53 PM


Finally, we can perform the decryption procedure with the private key using the following steps.

1. Calculate the original plaintext message from the ciphertext using the private key containing d and n. This is calculated using the equation $m = c^d(\text{mod } n)$.
2. Compare this value of m with the original m, and you should see that they are equal, since decryption is the inverse operation to encryption.

A Miniature RSA Example

Here is an example of RSA that is almost simple enough to do with pencil and paper. It is similar in scale to the **TinyRSA** code example discussed in this chapter. The bit size of the numbers used in this example is ridiculously small (32-bit integers) and offers no real security whatsoever, but at a conceptual level, this example provides a complete picture of what actually happens in the RSA algorithm. The advantage of studying this tiny paper and pencil example is that with these very small bit sizes, the underlying concepts are much more tangible and easily visualized. After all, not too many people can do 1024-bit arithmetic in their head! Even working with such tiny 32-bit numbers, the exponentiation step of the algorithm will easily overflow this 32-bit capacity if you are not careful about how you implement it.[13]

Following the conceptual steps outlined above, we start off by choosing two unequal prime numbers p and q.[14] Since we intentionally choose very small values, we prevent subsequent calculations from overflowing the 32-bit integer arithmetic. This also allows us to follow along using the Calculator program provided with Windows to verify the arithmetic.

1. Assume that the random values for the primes p and q have been chosen as

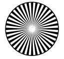
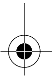
$$p = 47$$

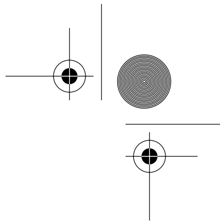
$$q = 73$$

2. Then the product n of these two primes is calculated:

$$n = p \cdot q = 3431$$

13. To avoid overflow, you must not use the exponentiation operator directly, but rather iterate multiplications in a loop, and in each iteration, you normalize the result to remain within the bounds of the modulus.
14. Again, in real life, end users are oblivious to these steps. The cryptographic application that they are using will automatically choose these two prime numbers and carry out all the required steps listed here. However, as a programmer, you may on rare occasion need to know how to implement a protocol such as this from scratch.



3. The Euler totient ϕ for these two primes is found easily using the following formula:

$$\phi = (p - 1) \cdot (q - 1) = 3312$$

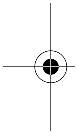
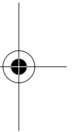
4. Now that we have n and ϕ, we should discard p and q, and destroy any trace of their existence.
5. Next, we randomly select a number e that is greater than 1, less than n, and relatively prime to phi. Of course, there is more than one choice possible here, and any candidate value you choose may be tested using the Euclidian method.[15] Assume that we choose the following value for e:

$$e = 425$$

6. Then the modular inverse of e is calculated to be the following:

$$d = 1769$$

7. We now keep d private and make e and n public.

Now that we have our private key information d and our public key information e and n, we can proceed with encrypting and decrypting data. As you would probably imagine, this data must be represented numerically to allow the necessary calculations to be performed. In a real-life scenario, the plaintext is typically a hash value or a symmetric key, but it could actually be just about any type of data that you could imagine. Whatever form this data takes, it will have to be somehow represented as a sequence of integer numbers, each with a size that will be limited by the key size that you are using. We do not concern ourselves here with the details of encoding and chunking of the data, but instead we focus on the conceptual aspects of RSA. For this reason, this example simply considers a scenario in which the plaintext data is one simple, small integer value.

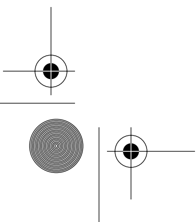
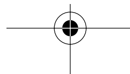
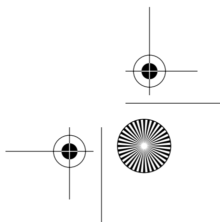
1. Assume that we have plaintext data represented by the following simple number:

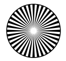
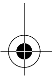
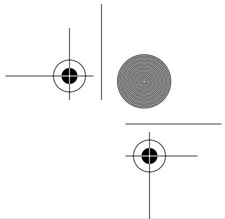
$$\text{plaintext} = 707$$

2. The encrypted data is computed by $c = m^e (\text{mod } n)$ as follows:

$$\text{ciphertext} = 707\text{^}425(\text{mod } 3431) = 2142$$

15. The Euclidian method is an efficient technique for finding the GCD (greatest common divisor) of any two integers.



Thorsteinson.book Page 113 Tuesday, July 29, 2003 1:53 PM


3. The ciphertext value cannot be easily reverted back to the original plaintext without knowing d (or, equivalently, knowing the values of p and q). With larger bit sizes, this task grows exponentially in difficulty. If, however, you are privy to the secret information that $d = 1769$, then the plaintext is easily retrieved using $m = c^d \pmod n$ as follows:

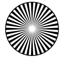
plaintext = 2142^1769(mod 3431) = 707

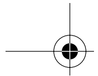
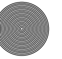
If you compile the following code, you will verify that the results shown above are correct. While you look at this code, keep in mind that a realistic RSA implementation uses a much larger modulus than $n = 3431$, and a realistic message typically contains too many bits to be represented by a tiny number such as $m = 707$.

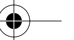
```
int m = 707;  //plaintext
int e = 425;  //encryption exponent
int n = 3431; //modulus
int c = 1;    //ciphertext

//encryption: c = m^e(mod n)
for (int i=0; i<e; i++) //use loop to avoid overflow
{
   c = c*m;
   c = c%n; //normalize within modulus
}
//ciphertext c is now 2142

int d = 1769; //decryption exponent
m = 1; //plaintext

//decryption m = c^d(mod n)
for (int i=0; i<d; i++) //use loop to avoid overflow
{
   m = m*c;
   m = m%n; //normalize within modulus
}
//plaintext m is now 707 matching original value
```

Caveat: Provability Issues

Every asymmetric algorithm is based on some trapdoor one-way function. This leads to a critically important question: How do we know for certain that a particular function is truly one-way? Just because nobody has publicly demonstrated a technique that allows the inverse function to be calculated quickly does not actually prove anything about the security of the algorithm.



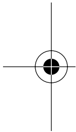
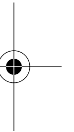
Thorsteinson.book Page 114 Tuesday, July 29, 2003 1:53 PM





If somebody has quietly discovered a fast technique to compute the inverse function, he or she could be busily decrypting enormous amounts of ciphertext every day, and the public may never become aware of it. It may seem paranoid, but high on the list of suspicions are major governments, since they employ large numbers of brilliant mathematicians who are outfitted with the most powerful computing resources available, putting them in a better position than most for cracking the trapdoor.

Most reassuring would be a rigorous mathematical proof of the *inherent difficulty* of computing the inverse function without knowledge of the secret key. With such a proof, further attempts at cracking the backdoor would be futile. Currently, rigorous formal proofs on the security of asymmetric algorithms are sorely lacking. Only in a few specialized (and not so useful) cases have any proofs been demonstrated in the public literature.

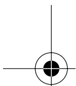
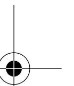
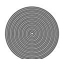
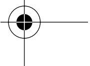
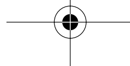
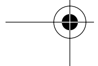
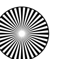
In spite of this worrisome lack of evidence, most cryptography experts currently believe that popular asymmetric algorithms, such as RSA, are quite secure given that a sufficient key size is used. Note that this is really just a consensus of learned opinion, which falls well short of a rigorous proof. Of course, considering that most ciphers used throughout history were assumed to be secure at the time that they were used, only to be broken using newly discovered attacks, a certain degree of anxiety is warranted.

In the case of RSA, the entire scheme depends on the widely held opinion that there are no techniques known that will allow an attacker to easily calculate the values of d, p, or q given only the public key containing n and e. Of course, this becomes more effective when you use larger values for p and q. C#'s built-in integer types top out at 64 bits for the long data type, which is nowhere near the size that we need for real asymmetric cryptography applications. Instead, we typically want to use integer data types that are represented by 1024-bit or larger words. Another worry is that this technique depends on a critical assumption that is widely believed to be true but has never been mathematically proven. The assumption is that there are in fact no tricks or fast techniques for factoring pq into its prime factors p and q. In fact, it has never been proven that factoring pq into p and q is the only way to attack RSA. If someone was either smart enough or lucky enough to dismantle these assumptions, that person could become the richest person in the world but would likely be a candidate for assassination.

Programming with .NET Asymmetric Cryptography

In this section, we look at the **RSAAlgorithm** and **SavingKeysAsXml** example programs provided for this chapter. These two code examples show how to encrypt and decrypt using the RSA algorithm as well as how to store and retrieve key information using an XML format. The RSA code example uses

the concrete **RSACryptoServiceProvider** class. Figure 4–2 shows where this class resides in the class hierarchy, under the abstract **AsymmetricAlgorithm** class. The other concrete class, **DSACryptoServiceProvider**, is discussed in Chapter 5, where we look at digital signatures.

An RSA Algorithm Example

The **RSAAlgorithm** example uses the **Encrypt** method of the **RSACryptoServiceProvider** class. This method takes two parameters, the first of which is a byte array containing the data to be encrypted. The second parameter is a boolean that indicates the padding mode to be used. Padding is required, since the data to be encrypted is usually not the exact number of required bits in length. Since the algorithm requires specific bit-sized blocks to process properly, padding is used to fill the input data to the desired length. If this second parameter is true, then the improved OAEP[16] padding is used. Otherwise, the traditional PKCS#1 v1.5 padding is used. PKCS#1 v1.5 has been traditionally the most commonly used padding scheme for RSA usage. However, it is recommended that all new RSA applications that will be deployed on platforms that support OAEP should use OAEP. Note that OAEP padding is available on Microsoft Windows XP and Windows 2000 with the

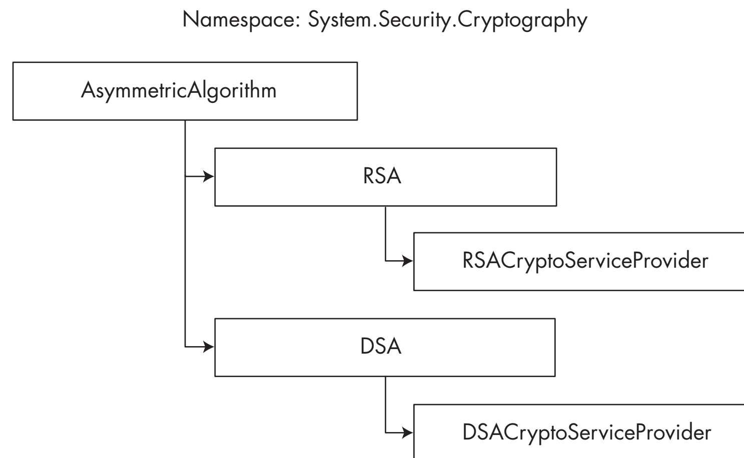
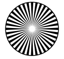
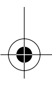
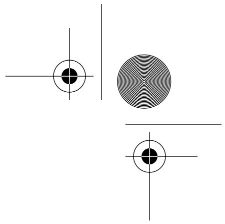


FIGURE 4–2 *The asymmetric algorithm class hierarchy.*

16. OAEP (Optimal Asymmetric Encryption Padding) is a padding technique developed by Mihir Bellare and Phil Rogaway in 1993 for use with RSA. OAEP provides significantly improved security characteristics over the popular PKCS#1 v1.5 padding scheme.

Thorsteinson.book Page 116 Tuesday, July 29, 2003 1:53 PM

high-encryption pack installed. Unfortunately, previous versions of Windows do not support OAEP, which will cause the **Encrypt** method, with the second parameter set to true, to throw a **CryptographicException**. The **Encrypt** method returns the resulting encrypted data as a byte array. Here is the syntax for the **Encrypt** method.

```
public byte[] Encrypt(
   byte[] rgb,
   bool fOAEP
);
```

The complementary method to **Encrypt** is of course **Decrypt**. You can probably guess how it works. The second parameter is a byte array containing the ciphertext to be decrypted. The second parameter is the same as that in the **Encrypt** method, indicating the padding mode, as described previously. The return value is a byte array that will contain the resulting recovered plaintext. Here is the syntax for the **Decrypt** method.

```
public byte[] Decrypt(
   byte[] rgb,
   bool fOAEP
)
```

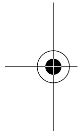
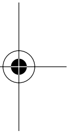
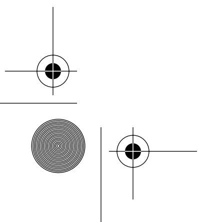
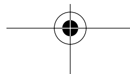
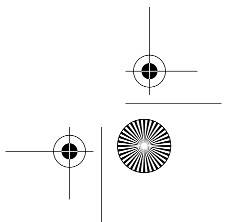



Figure 4–3 shows the **RSAAlgorithm** example being used to encrypt and decrypt a plaintext message. You enter the plaintext in the **TextBox** at the top of the form. You then click on the Encrypt button, which fills in all but the last form field, including the resulting ciphertext and RSA parameters that were used. You then click on the Decrypt button, which displays the recovered plaintext in the field at the bottom of the form. Of course, the recovered plaintext should be identical to the original plaintext.

Now let's look at the code in the **RSAAlgorithm** example code. The **buttonEncrypt_Click** method is called when the user clicks on the Encrypt button. This encrypts the contents of the plaintext textbox using the established public RSA key. The public/private RSA key pair is provided by the program automatically when it starts, but it may subsequently be changed using the New RSA Parameters button. There are a few places in the code where user interface elements are being enabled and disabled, which are not germane to our focus on RSA functionality. Therefore, these user interface code sections are ignored here. If you are curious about how these user interface details work, please study the simple code sections following each of the `//do UI stuff` comments.

We first generate the initial RSA parameters by calling the **GenerateNewRSAParams** method in the **RSAAlgorithm_Load** method. The **GenerateNewRSAParams** method is also called each time the user clicks on the New RSA Parameters button, which is handled by the **buttonNewRSAParams_Click** method. The **GenerateNewRSAParams** method is very simple. It just creates

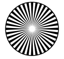
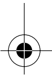
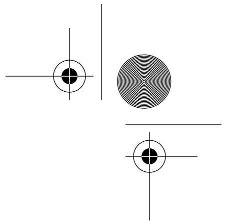




Thorsteinson.book Page 117 Tuesday, July 29, 2003 1:53 PM

FIGURE 4–3 *The RSAAlgorithm example program.*

an **RSACryptoServiceProvider** class object, stores its public and private RSA parameters by calling the RSA class's **ExportParameters** method, and displays a few of the more important of these parameters in the user interface. These RSA parameters are actually stored in two fields of type **RSAParameters**. The **RSAParameters** field named **rsaParamsExcludePrivate** gets a copy of the public-only RSA parameters (i.e., the modulus and exponent values only), which is required for encryption purposes in the **buttonEncrypt_Click** method. The other **RSAParameters** field, named **rsaParamsIncludePrivate** gets a copy of the combined public and private RSA parameters, which is required in the **buttonDecrypt_Click** method.

Here is the **GenerateNewRSAParams** method. Note that the **Export-Parameters** method is called twice. The first time, the parameter passed into this method is true, and the second time, it is false. Passing true indicates that you want to include all key parameter information, including the private key information. False indicates that only the public key information is to be stored. We separate these cases into two distinct fields to demonstrate how

the encryption will use only the public information, but the decryption will use both the public and private key information. This is a crucial point in understanding asymmetric cryptography. This would perhaps be even clearer if we broke the encryption and decryption portions of this example into two separate applications, but this example is provided as a simple monolithic program purely for easy study. You should at some point take a moment to verify that the encryption and decryption functions in this program do indeed use only their own appropriate version of this RSA parameter information, using the corresponding **ImportParameters** method.

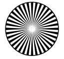
```
private void GenerateNewRSAParams()
{
   //establish RSA asymmetric algorithm
   RSACryptoServiceProvider rsa =
      new RSACryptoServiceProvider();

   //provide public and private RSA params
   rsaParamsIncludePrivate =
      rsa.ExportParameters(true);

   //provide public only RSA params
   rsaParamsExcludePrivate =
      rsa.ExportParameters(false);
```

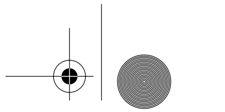
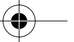
When we create an instance of the **RSACryptoServiceProvider** class, we actually get the RSA implementation provided by the underlying cryptographic service provider (CSP). This class is directly derived from the **RSA** class. The **RSA** class allows other RSA implementations to be implemented as other derived classes; however, the CSP implementation is currently the only one available.

The two fields that store the RSA parameter information when **ExportParameters** is called are declared as **RSAParameters** type fields, as shown in the following code snippet. The **rsaParamsExcludePrivate** filed will be used for encryption, and the **rsaParamsIncludePrivate** field will be used in decryption in this example.

```
//public modulus and exponent used in encryption
RSAParameters rsaParamsExcludePrivate;

//public and private RSA params use in decryption
RSAParameters rsaParamsIncludePrivate;
```

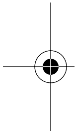
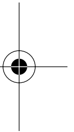
In the **buttonEncrypt_Click** method we then create a new instance of **RSACryptoServiceProvider** class, and we initialize it with the stored public key information by calling the **RSA** object's **ImportParameters** method,



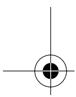
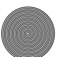
Thorsteinson.book Page 119 Tuesday, July 29, 2003 1:53 PM



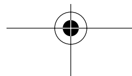
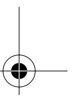
specifying **rsaParamsExcludePrivate** as the parameter. Next, we obtain the plaintext in the form of a byte array named **plainbytes**. Finally, we perform the main function of this method by calling on the **Encrypt** method of the **RSA** object. This returns another byte array, which is an instance field named **cipherbytes**. This is an instance field rather than a local variable, because we need to communicate this byte array to the decryption method, and local variables are not maintained across method calls.

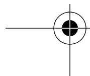
```
private void buttonEncrypt_Click(
   object sender, System.EventArgs e)
{
   //do UI stuff
   ...

   //establish RSA using parameters from encrypt
   RSACryptoServiceProvider rsa =
      new RSACryptoServiceProvider();

   //import public only RSA parameters for encrypt
   rsa.ImportParameters(rsaParamsExcludePrivate);

   //read plaintext, encrypt it to ciphertext
   byte[] plainbytes =
      Encoding.UTF8.GetBytes(textPlaintext.Text);
   cipherbytes =
      rsa.Encrypt(
      plainbytes,
      false); //fOAEP needs high encryption pack

 //display ciphertext as text string
 ...

 //display ciphertext byte array in hex format
 ...

 //do UI stuff
 ...
}
...
//variable communicated from encrypt to decrypt
byte[] cipherbytes;
```

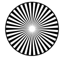
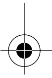
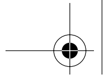
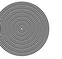



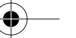
The **buttonDecrypt_Click** method is called when the user clicks on the Decrypt button. Again, an **RSA** object is created. The **RSA** object is repopulated with the information provided by calling the **RSA** object's **ImportParameters** method, but this time, the parameter to this method is the **rsaParamsIncludePrivate**, which includes both public and private RSA



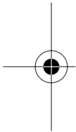
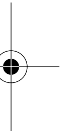







Thorsteinson.book Page 120 Tuesday, July 29, 2003 1:53 PM

key information. The plaintext is then obtained by calling the **Decrypt** method of the **RSA** object. Since a matching set of RSA algorithm parameters were used for both encryption and decryption, the resulting plaintext matches perfectly with the original plaintext.

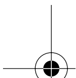
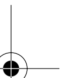
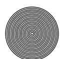
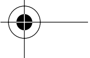
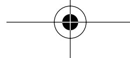
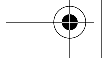
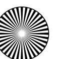
```
private void buttonDecrypt_Click(
   object sender, System.EventArgs e)
{
   //establish RSA using parameters from encrypt
   RSACryptoServiceProvider rsa =
      new RSACryptoServiceProvider();

   //import public and private RSA parameters
   rsa.ImportParameters(rsaParamsIncludePrivate);

   //read ciphertext, decrypt it to plaintext
   byte[] plainbytes =
      rsa.Decrypt(
      cipherbytes,
      false); //fOAEP needs high encryption pack

//display recovered plaintext
...

//do UI stuff
   ...
}

...
//variable communicated from encrypt to decrypt
byte[] cipherbytes;
```

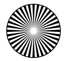
Saving Keys as XML

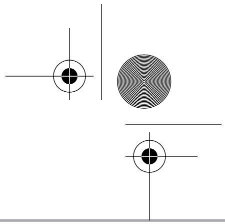
You might not always want to transmit the contents of the **ExportParameters** object directly between arbitrary applications, especially between different platforms and cryptographic libraries. After all, the **ExportParameters** class is very Microsoft- and .NET-specific. A much more convenient and generalized format for transmitting a public key is via an XML stream.[17] The **Sav-**

17. Transmitting a public key via an **ExportParameters** object or via XML is not a security issue. Of course, you should not make a habit of transmitting private asymmetric keys or symmetric session keys in the clear. To exchange such sensitive key information, you must actually encrypt the encryption key. This may sound a bit like a recursive statement, but it actually makes sense. In Chapter 6 we see how to exchange such encrypted secret key information using established XML cryptography standards.

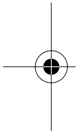
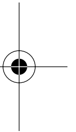
Thorsteinson.book Page 121 Tuesday, July 29, 2003 1:53 PM





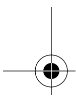
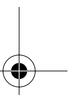
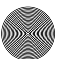
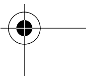
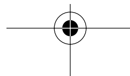
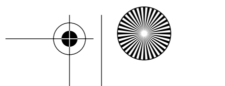
ingKeysAsXml example program shows how to read and write keys in XML format. This example is almost identical to the **RSAAlgorithm** example we just looked at. The significant difference is that we use XML for storing and transmitting the public key information from the encryption method to the decryption method rather than use an **ExportParameters** object. Another slight difference is that the RSA parameter information is not displayed; the contents of the key XML stream is displayed instead, but that is of course only a user interface detail.

For simplicity and ease of demonstration, this example is again implemented as a single monolithic application. This is purely for ease of demonstration, and it would be straightforward to take this example and break it up into two separate encrypting and decrypting programs. Our purpose here is to show both the sending (encrypting) and receiving (decrypting) code and how the XML data is used to store key information between the two. To make this example somewhat more realistic, the XML data is written to a file rather than stored in a shared field, as was done in the previous example. This simulates the case in a real-world scenario in which you would need to read and write this information to some type of external storage or perhaps via a socket stream. From the programmer's perspective, the most significant change from the previous example is that the calls to the **ExportParameters** and **ImportParameters** methods of the **RSACryptoServiceProvide** class have been replaced with calls to the **ToXmlString** and **FromXmlString** methods of the same class. Once again, a boolean parameter is used to indicate whether private information is included or excluded in the stored key information.

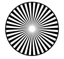
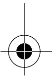
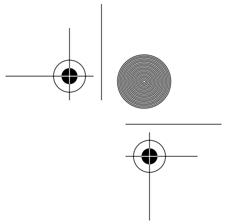



Here is the **GenerateNewRSAParams** method, which serves the same basic purpose as described in the previous program example. The difference is that we are storing the key information in XML format, in two files named **PublicPrivateKey.xml** and **PublicOnlyKey.xml**, by calling the **ToXmlString** method with a boolean parameter. These two files will be used later in the encryption and decryption functions.

```
private void GenerateNewRSAParams()
{
   //establish RSA asymmetric algorithm
   RSACryptoServiceProvider rsa =
      new RSACryptoServiceProvider();

   //provide public and private RSA params
   StreamWriter writer =
      new StreamWriter("PublicPrivateKey.xml");
   string publicPrivateKeyXML =
      rsa.ToXmlString(true);
   writer.Write(publicPrivateKeyXML);
   writer.Close();
```

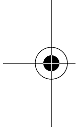
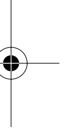







Thorsteinson.book Page 122 Tuesday, July 29, 2003 1:53 PM

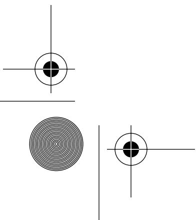
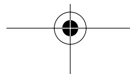
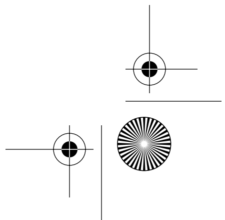





```
   //provide public only RSA params
   writer =
      new StreamWriter("PublicOnlyKey.xml");
   string publicOnlyKeyXML =
      rsa.ToXmlString(false);
   writer.Write(publicOnlyKeyXML);
   writer.Close();

   //display public and private RSA key
      textBoxPublicKeyXML.Text = publicPrivateKeyXML;

   //do UI stuff
   ...
}
```

Next, let's look at the **buttonEncrypt_Click** method. We create a new **RSACryptoServiceProvider** object and initialize it by calling the **FromXmlString** method with the public key information stored in the **PublicOnlyKey.xml** file. Then we call the **RSA** object's **Encrypt** method to perform the cryptographic transformation on the plaintext.

```
private void buttonEncrypt_Click(
   object sender, System.EventArgs e)
{
   //do UI stuff
   ...

   //establish RSA asymmetric algorithm
   RSACryptoServiceProvider rsa =
      new RSACryptoServiceProvider();

   //public only RSA parameters for encrypt
   StreamReader reader =
      new StreamReader("PublicOnlyKey.xml");
   string publicOnlyKeyXML = reader.ReadToEnd();
   rsa.FromXmlString(publicOnlyKeyXML);
   reader.Close();

   //read plaintext, encrypt it to ciphertext
   byte[] plainbytes =
      Encoding.UTF8.GetBytes(textPlaintext.Text);
   cipherbytes =
      rsa.Encrypt(
      plainbytes,
      false); //fOAEP needs high encryption pack

   //display ciphertext as text string
   ...
```

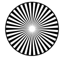
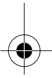
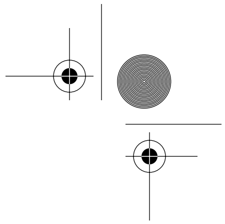
Thorsteinson.book Page 123 Tuesday, July 29, 2003 1:53 PM


```
   //display ciphertext byte array in hex format
   ...

   //do UI stuff
   ...
}
```

Finally, the **buttonDecrypt_Click** method creates its own new **RSACryptoServiceProvider** object, but it initializes it by calling **FromXml-String** using the **PublicPrivateKey.XML** file, which contains both public and private key information—a requirement of RSA decryption.

```
private void buttonDecrypt_Click(
   object sender, System.EventArgs e)
{
   //establish RSA using key XML from encrypt
   RSACryptoServiceProvider rsa =
      new RSACryptoServiceProvider();

   //public and private RSA parameters for encrypt
   StreamReader reader =
      new StreamReader("PublicPrivateKey.xml");
   string publicPrivateKeyXML = reader.ReadToEnd();
   rsa.FromXmlString(publicPrivateKeyXML);
   reader.Close();

   //read ciphertext, decrypt it to plaintext
   byte[] plainbytes =
      rsa.Decrypt(
      cipherbytes,
      false); //fOAEP needs high encryption pack

   //display recovered plaintext
   ...

   //do UI stuff
   ...
}
```

Figure 4–4 shows the **SavingKeysAsXml** example being used to encrypt and decrypt a plaintext message. Notice the XML display shows contents of the **PublicPrivateKey.xml** file that is being used by the decryption method. It is a bit difficult to read with all the XML elements running in a single, continuous stream, but if you look closely at it, you should be able to see each of the RSA parameter values used. The encryption method uses only the modulus and exponent elements.



Thorsteinson.book Page 124 Tuesday, July 29, 2003 1:53 PM


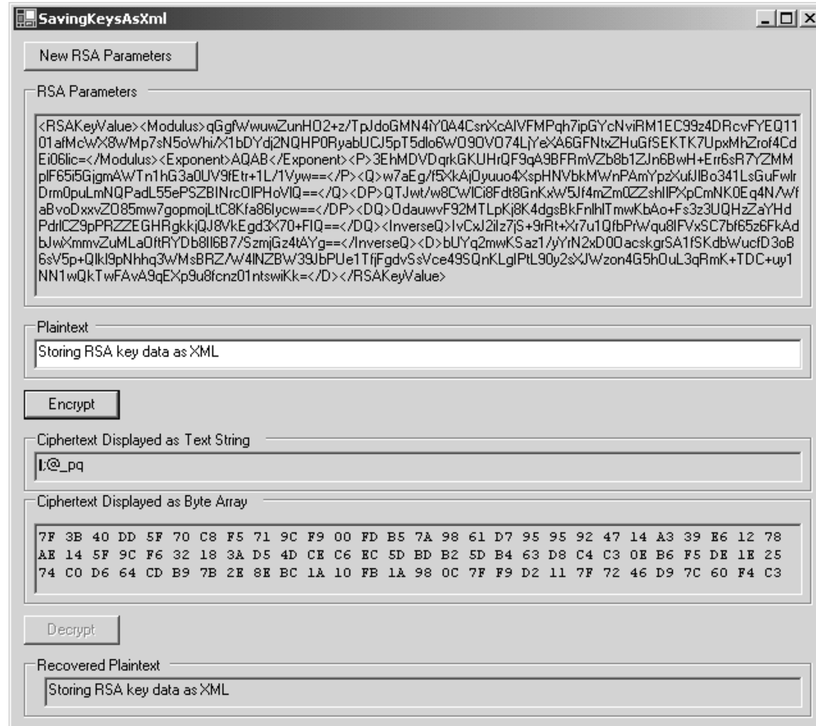
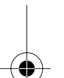
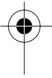
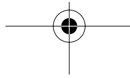
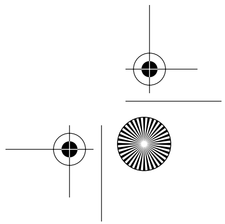
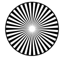


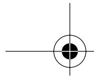
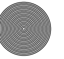
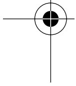
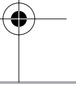
FIGURE 4–4 *The SavingKeysAsXml example program.*

Digital Certificates

In order to make an asymmetric algorithm such as RSA work, you need a way to expose the public key. A public key can be shared manually, but ideally, a CA is used to share a public key that is contained in a digital certificate (also known as a digital ID). A digital certificate is a document that you use to prove your identity in messages or electronic transactions on the Internet. You can obtain a digital certificate from a trusted third party, such as Verisign, or you can set up a locally trusted CA server within your own organization to provide digital certificates. In Microsoft Outlook, you access a CA and generate a digital certificate by selecting the Tools | Options menu item, clicking on the Security tab, and then clicking on the Get Digital ID button.

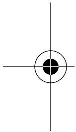
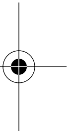







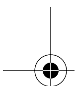
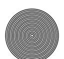
Thorsteinson.book Page 125 Tuesday, July 29, 2003 1:53 PM





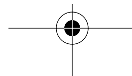
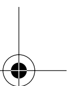
There are many commercial CAs and many levels of certificates, which differ in cost and levels of trust. To varying degrees, the CA attempts to verify that you are who you claim to be, and, if the CA is convinced of your identity, it will create a document containing the public key that you provide along with other identifying information about you. The CA will then digitally sign that document using its own private key. Of course, nobody other than the CA may ever see the CA's private key, and your private key is never divulged to anyone, including the CA. The resulting signed document is known as a digital certificate, which the CA makes available in a database or directory service to anyone who is interested in dealing with you in either a secure or an authenticated manner. Other parties simply access the database to obtain your digital certificate whenever they need it. Any such party can use the CA's public key to authenticate your digital certificate, and then use the contained public key belonging to you to carry on with whatever encryption or authentication protocol with you that is intended.

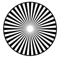
Summary

This chapter introduced asymmetric algorithms, particularly the RSA algorithm. We saw how the asymmetric algorithm can be used to solve certain problems with symmetric algorithms by making it unnecessary to share any secret key. We also looked at how RSA works and how to program with the **RSACryptoServiceProvider** class in the .NET Framework. Finally, we looked at how to format RSA parameters into an XML format so that they may be shared with other parties. In the next chapter, we will continue to study asymmetric algorithms, but we will shift our attention to digital signatures, which is the other important aspect of asymmetric algorithms.



Thorsteinson.book Page 126 Tuesday, July 29, 2003 1:53 PM