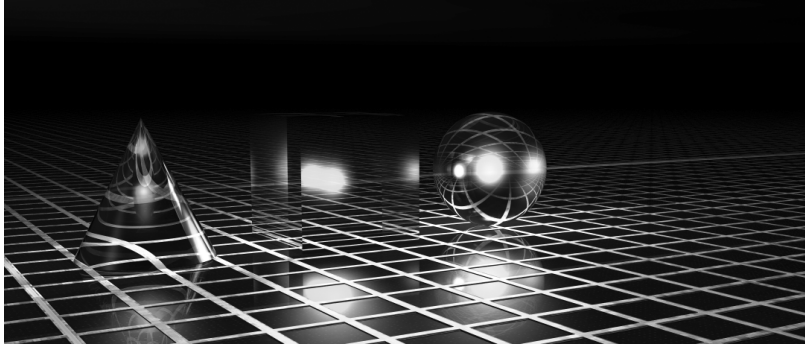
1



Encryption

Key concepts in this chapter are:

- Using hash digests for storing and verifying passwords
- Using private key encryption
- Writing a public key encryption routine
- Modifying a database to store passwords and bank account numbers in encrypted format
- Protecting password fields on forms
- Knowing where to use encryption in your own applications

If you read the Introduction, you'll recall that this book is for Visual Basic .NET programmers new to security, not security experts new to Visual Basic .NET. This book unashamedly simplifies concepts and leaves out unnecessary techno-babble with the goal of making security easier to understand and implement—without sacrificing accuracy. For many programmers, this simplified look at security is all they will ever need, whereas others, after given a taste of security, will want to know more. In a nutshell, this book is not the last word in security; instead, it is the first book you should read on the subject.

What is encryption? Before discussing how to implement encryption with Visual Basic .NET, you need to have an understanding of encryption in general. Encryption is about keeping secrets safe by scrambling messages to make them illegible. In encryption terms, the original message is known as *plain text*, the scrambled message is called *cipher text*, the process of turning plain text into cipher text is called *encryption*, and the process of turning cipher text back into plain text is called *decryption*.

Encryption isn't just used in cyberspace or in mysterious government work either. You can find examples of it in everyday activities such as baseball. For example, in the game of baseball, the catcher commonly uses hand signals to suggest to the pitcher the type of ball the pitcher should throw next. Curveballs, sinkers, sliders, and fastballs all have a different hand signal. As long as the batter and others on the opposing team don't understand the catcher's hand signals, their secret is safe. Figure 1-1 shows the process of encryption as it applies to baseball.

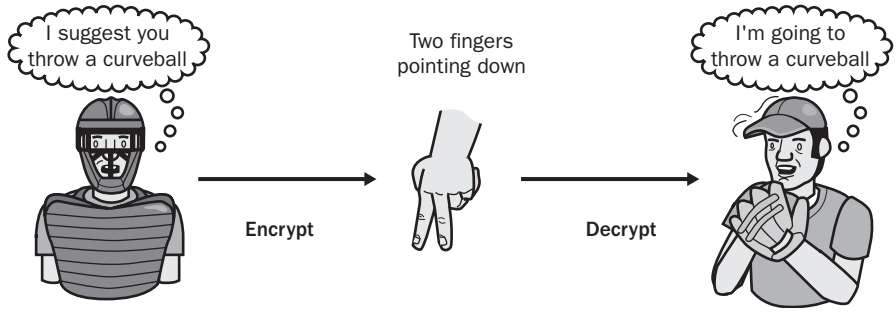


Figure 1-1 Encrypting and decrypting a secret message

Computers allow us to encrypt rich messages in real time, but the underlying principle is the same as in the simple baseball example. For encryption to be effective, the sender and the recipient must be the only parties who know how to encrypt and decrypt the messages. Microsoft Windows and the .NET Framework provide robust algorithms for doing encryption, and we'll use these routines in this chapter. Unless you're an encryption expert, you shouldn't try to write your own encryption algorithm, for exactly the same reason that only aviation engineers should build their own airplanes.

It's a common misconception that encryption algorithms and hash functions must be secret to be secure. The encryption algorithms and hash functions used in this book are commonly understood, and the associated source code is distributed freely on the Internet. They are, however, still secure because they are designed to be irreversible (in the case of hash functions) or they require the user to supply a secret key (in the case of encryption algorithms). As long as only the authorized parties know the secret key, the encrypted message is safe from intruders. Encryption helps to ensure three things:

- **Confidentiality** Only the intended recipient will be able to decrypt the message you send.
- **Authentication** Encrypted messages you receive have originated from a trusted source.

- **Integrity** When you send or receive a message, it won't be tampered with in transit.

Some cryptography mechanisms are one way; that is, they produce cipher text that can't be decrypted. A good example of a one-way cryptography is a *hash*. A hash is a very large number (the hashes in this chapter are 160 bits in size) mathematically generated from a plain-text message. Because the hash contains no information about the original message, the original message can't be derived from the hash. "What use is cipher text that can't be decrypted?" you might ask. As you'll see soon, a hash is useful for verifying that someone knows a secret without actually storing the secret.

In the examples in this chapter, you'll learn how to create and use a hash for verifying passwords. You'll also learn how to use private key encryption for storing and retrieving information in a database. We'll also begin building a library of easy-to-use encryption functions that you can reuse in your Visual Basic programs.

Practice Files

If you haven't already installed the practice files, which you can download from the book's Web site at *http://www.microsoft.com/mspress/books/6432.asp,* now would be a good time to do so. If you accept the default installation location, the samples will be installed to the folder C:\Microsoft Press\VBNETSec, although you'll be given an opportunity to change the destination folder during the installation process. The practice files are organized by version of Microsoft Visual Basic, chapter, and exercise. The practice files for each chapter give a starting point for the exercises in that chapter. Many chapters also have a finished version of the practice files so that you can see the results of the exercise without actually performing the steps. To locate the practice file for a particular exercise, look for the name of the exercise within the chapter folder. For example, the Visual Basic .NET 2003 versions of the practice files for the following section on using hash digests for encrypting database fields will be in the folder

```
C:\Microsoft Press\VBNETSEC\VB.NET 2003\CH01_Encryption\
EncryptDatabaseField\Start
```

In many of the exercises in this book, you'll modify an employee management system, adding security features to make the program more secure. The employee management system is a sample program that adds, edits, and removes employees for a fictional company. For background on the employee

management system, see Appendix A. The system uses a Microsoft Access database named EmployeeDatabase.mdb. The techniques you learn are equally relevant to Microsoft SQL Server, Oracle, DB2, and other databases. You don't need Microsoft Access to use the practice files because the database drivers are installed with Microsoft Visual Studio .NET. In some exercises, we modify the database structure. These exercises are optional. If you don't have Microsoft Access installed, don't worry: the practice files have been designed to work with the database whether or not you make the changes to the database structure.

Hash Digests

As we mentioned earlier in this chapter, a hash is a type of one-way cryptography. Some people refer to hashing as encryption; others feel it's not strictly encryption because the hash cannot be unencrypted. A hash is a very large number, generated by scrambling and condensing the letters of a string. In this chapter, you'll use the SHA-1 algorithm. SHA-1 is an acronym for Secure Hashing Algorithm. The "-1" refers to revision 1, which was developed in 1994. SHA-1 takes a string as input and returns a 160-bit (20-byte) number. Because a string is being condensed into a fixed-size number, the result is called a *hash digest*, where *digest* indicates a shortened size, similar to *Reader's Digest* condensed books. Hash digests are considered to be one-way cryptography because it's impossible to derive the original string from the hash. A hash digest is like a person's fingerprint. A fingerprint uniquely identifies an individual without revealing anything about that person—you can't determine someone's eye color, height, or gender from a fingerprint. Figure 1-2 shows the SHA-1 hash digests for various strings. Notice that even very similar strings have quite different hash digests.

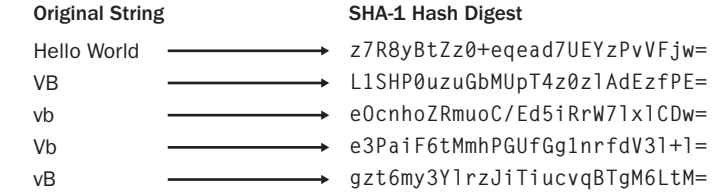


Figure 1-2 SHA-1 hash digests

It's common, as shown in Figure 1-2, to display a hash as a base-64 encoded 28-character string. This is easier to read than a 48-digit (160-bit) number.

Hash digests are useful for verifying that someone knows a password, without actually storing the password. Storing passwords unencrypted in the database opens two security holes:

- If an intruder gains access to the database, he can use the information to later log on to the system using someone else's username and password.
- People often use the same password for different systems, so the stolen passwords might allow the intruder to break into other systems.

Because the password is used solely for authenticating the user, there's no reason to store the password in the database. Instead, a hash digest of the password can be stored. When the user logs on to the system, a hash digest from the password she types in is created and compared with the hash digest stored in the database. If an intruder somehow gained access to the password table, he wouldn't be able to use the hash digest to log on to the system because he would need to know the unencrypted password, which isn't stored anywhere. In the following exercise, you'll change the employee management system to validate logons using hash digests instead of passwords.[1]

Create a hash digest function

In this exercise, you'll write a function that returns SHA-1 hash digests. You'll then use this function to create hash digests for all the passwords in EmployeeDatabase.mdb and store the hash digests in a field named *PasswordHash*. This field is already in the database, but it's currently unpopulated. The passwords are currently stored unencrypted in the *Password* field.

1. Start Visual Studio .NET, and open the empty project CH01_Encryption\EncryptDatabaseField\Start\EncryptDatabaseField.sln. This project is empty of code, but it has been set up with the database path, import statements, and a shared library module.
2. Open the module SecurityLibrary.vb in the Visual Basic .NET editor. This module is empty: it's where you'll put all your reusable security routines for use in this and other projects. Add the following function to the library:

```
Namespace Hash
  Module Hash
```

1. Validating against hashes is a good mechanism to use for an application that opens a database directly. For a client-server application or a Web application, this mechanism does not protect against "spoofing" the server component—where an intruder who knows the hashes constructs a fake client application that submits the hash to the server. However, if an intruder gains access to the list of passwords, they can do less damage if the passwords are hashed.

```
    Function CreateHash(ByVal strSource As String) As String
      Dim bytHash As Byte()
      Dim uEncode As New UnicodeEncoding()
         'Store the source string in a byte array
         Dim bytSource() As Byte = uEncode.GetBytes(strSource)
         Dim sha1 As New SHA1CryptoServiceProvider()
         'Create the hash
         bytHash = sha1.ComputeHash(bytSource)
         'return as a base64 encoded string
         Return Convert.ToBase64String(bytHash)
    End Function
  End Module
End Namespace
```

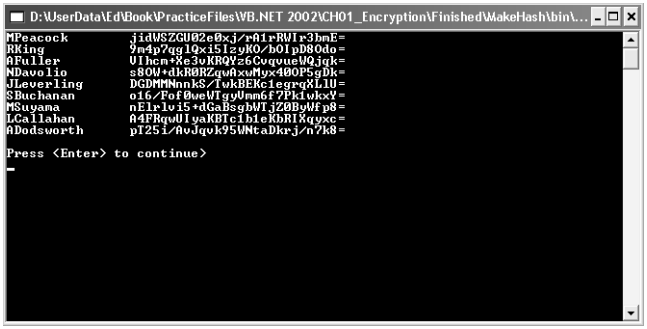
This function is all that is needed to create a hash. It converts a string to an array of bytes and then creates a SHA-1 hash. The result is returned as a 28-character string.

3. Open MainModule.vb. You'll now write a routine to store hash digests for all the passwords in the database. Add the following code to the module:

```
Sub Main()
  EncryptField("Password", "PasswordHash")
End Sub
Sub EncryptField(ByVal strSourceField As String, _
          ByVal strDestinationField As String)
  Dim strSQL, strUsername, strPlainText, strCipherText As String
  strSQL = "Select Username, " & strSourceField & " from Employee"
  Dim cnRead As New OleDbConnection(G_CONNECTIONSTRING)
  Dim cnWrite As New OleDbConnection(G_CONNECTIONSTRING)
  Dim cmdRead As New OleDbCommand(strSQL, cnRead)
  Dim cmdWrite As New OleDbCommand()
  cmdWrite.Connection = cnWrite
  Dim dr As OleDbDataReader
  'Open two connections,
  'one for reading and the other for writing
  cnRead.Open()
  cnWrite.Open()
  dr = cmdRead.ExecuteReader()
  'Loop through the table, reading strings
  'encrypting and writing them back
  While dr.Read
    strUsername = dr.GetString(0)
    strPlainText = dr.GetString(1)
    strCipherText = Hash.CreateHash(strPlainText)
    strSQL = "UPDATE Employee SET " & strDestinationField & " ='" & _
```

```
      strCipherText & "' WHERE Username ='" & strUsername & "'"
    cmdWrite.CommandText = strSQL
    cmdWrite.ExecuteNonQuery()
    Console.WriteLine(LSet(strPlainText, 16) & strCipherText)
  End While
  Console.WriteLine(vbCrLf & "Press <Enter> to continue>")
  Console.ReadLine()
End Sub
```

4. Now press F5 to run the project. It will populate the *PasswordHash* field and display the results in the console window. The output should look like this:



Verify passwords using a hash digest

Now you will modify the employee management system to verify passwords with the hash digests you just created.

1. In Visual Studio .NET, open the project CH01_Encryption\EMS\Start\EMS.sln.

2. Open the class clsEmployee.vb; find the declaration

   ```
   Private m_Password As String
   ```

   and change it to

   ```
   Private m_PasswordHash As String
   ```

3. In the *Create* function, find the line that reads

   ```
   Me.m_Password = CStr(dr("Password"))
   ```

   and change it to

   ```
   Me.m_PasswordHash = CStr(dr("PasswordHash"))
   ```

4. In the *IsValidPassword* function, find the line that reads

```
If strPassword = Me.m_Password AndAlso Me.m_IsValidUser Then
```

and change it to read

```
If Hash.CreateHash(strPassword) = Me.m_PasswordHash _
    AndAlso Me.m_IsValidUser Then
```

5. Open the form frmAddNew.vb, and double-click the Add button to open the *btnAdd_Click* event handler. Change the first line of code from

```
Dim strPassword As String = Me.txtPassword.Text
```

to

```
Dim strPassword As String = Hash.CreateHash(Me.txtPassword.Text)
```

6. Still in the *btnAdd_Click* event, find the line of code that reads

```
strSQL = _
  "INSERT INTO Employee ( UserName, [Password], Fullname ) " & _
    "SELECT '" & strUsername & "' As Field1," & _
    "'" & strPassword & "' As Field2," & _
    "'" & strUsername & "' As Field3"
```

and change it to

```
strSQL = _
  "INSERT INTO Employee ( UserName, [PasswordHash], Fullname ) " & _
    "SELECT '" & strUsername & "' As Field1," & _
    "'" & strPassword & "' As Field2," & _
    "'" & strUsername & "' As Field3"
```

7. Press F5 to run the project. You can log on using the username RKing with the password RKing, as shown in the following illustration. Congratulations—you are now checking passwords without storing passwords! Even if an intruder gains access to the database, the password hash digests can't then be used to log on.

How Does a Hash Digest Work?

How does a hash digest work? If each unique string results in a unique hash digest, is it possible to decrypt the hash digest and derive the original string?

To answer these two questions, let's create a simple hash algorithm. We'll start by assigning every letter in the alphabet a unique number, so A is equal to 1, B equal to 2, C equal to 3, and so on up to Z, which is equal to 26. Next we'll use these values to create a hash by adding them together for each character in a string. The string VB generates a hash of 24 because V is the 22nd letter in the alphabet and B is the second letter (22 + 2 = 24).

Can the hash of 24 be reverse-engineered to derive the original string? No. The hash doesn't tell us the length, starting character, or anything else about the original string. In this simple example, the strings VB, BV, BMDACA, FEJAAA, and thousands of other combinations all give a hash of 24. When different strings produce the same hash value, this is known as a *collision*. A good hashing algorithm should produce unique results and be *collision-free*. SHA-1 produces collision-free results, and it scrambles and condenses the original string in such a way that it's considered computationally infeasible to derive the original string.

Private Key Encryption

While hash digests are useful for one-way encryption, when you need to decrypt the encrypted information, you need to use two-way encryption. The most common two way-encryption technique is key-based encryption. A key is simply a unique string that you pass together with a plain-text message to an encryption algorithm, which returns the message encrypted as cipher text. The cipher text bears no resemblance to the original message. To decrypt the cipher text, you again pass the key with the cipher text to a decryption algorithm, which returns the original plain-text message.

The most common type of key-based encryption is private key encryption, also called symmetric, traditional, shared-secret, secret-key, or conventional encryption. (Encryption is one of those areas in computing in which many names mean the same thing.) Private key encryption relies on the sender and

recipient both knowing the key. This implies that potential intruders do not know the key and have no way to obtain the key. Private key encryption is good for communicating information over the Internet or for storing sensitive information in a database, registry, or file. Figure 1-3 shows private key encryption in action.

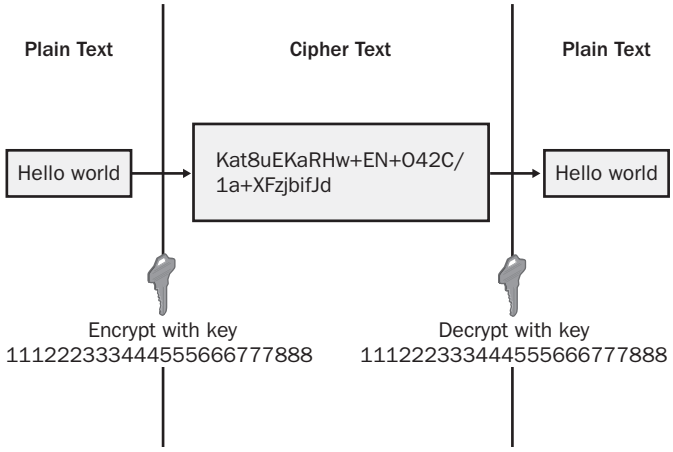


Figure 1-3 Private key encryption

Now you'll add functions for applying private key encryption to the security library you created in the preceding exercise, and you'll use private key encryption to store and retrieve bank account information in the database. The type of encryption you'll use is Triple-DES—DES is an acronym for Data Encryption Standard. Triple refers to how the encryption works—first the plain text is encrypted; this encrypted result is encrypted again; and finally, the encrypted-encrypted plain-text message is encrypted once more, resulting in the plain-text message being encrypted three times and earning the moniker *Triple-DES*. You get three encryptions for the price of one, and the result is a robust 192-bit encryption.

Encrypt the BankAccount field with a private key

The employee management system stores bank account information for the purpose of depositing the salaries of employees directly into their bank accounts. Currently this information is being stored as plain text. In this exercise, you'll add private key encryption and decryption functions to your security library, and you'll use these functions to encrypt the BankAccount field.

1. Open the same EncryptDatabaseField program we used when encrypting the password field earlier in this chapter. The project is located at CH01_Encryption\ EncryptDatabaseField\Start\EncryptDatabaseField.sln. We will be changing the program to encrypt the BankAccount field.

2. Add the following code to the end of SecurityLibrary.db:

```
Namespace PrivateKey
  Module PrivateKey
    Function Encrypt(ByVal strPlainText As String, _
      ByVal strKey24 As String) As String
      Dim crp As New TripleDESCryptoServiceProvider()
      Dim uEncode As New UnicodeEncoding()
      Dim aEncode As New ASCIIEncoding()
      'Store plaintext as a byte array
      Dim bytPlainText() As Byte = uEncode.GetBytes(strPlainText)
      'Create a memory stream for holding encrypted text
      Dim stmCipherText As New MemoryStream()
      'Private key
      Dim slt(0) As Byte
      Dim pdb As New PasswordDeriveBytes(strKey24, slt)
      Dim bytDerivedKey() As Byte = pdb.GetBytes(24)
      crp.Key = bytDerivedKey
      'Initialization vector is the encryption seed
      crp.IV = pdb.GetBytes(8)
      'Create a crypto-writer to encrypt a bytearray
      'into a stream
      Dim csEncrypted As New CryptoStream(stmCipherText, _
        crp.CreateEncryptor(), CryptoStreamMode.Write)
      csEncrypted.Write(bytPlainText, 0, bytPlainText.Length)
      csEncrypted.FlushFinalBlock()
      'Return result as a Base64 encoded string
      Return Convert.ToBase64String(stmCipherText.ToArray())
    End Function
    Function Decrypt(ByVal strCipherText As String, _
       ByVal strKey24 As String) As String
      Dim crp As New TripleDESCryptoServiceProvider()
      Dim uEncode As New UnicodeEncoding()
      Dim aEncode As New ASCIIEncoding()
      'Store cipher text as a byte array
      Dim bytCipherText() As Byte = _
        Convert.FromBase64String(strCipherText)
      Dim stmPlainText As New MemoryStream()
      Dim stmCipherText As New MemoryStream(bytCipherText)
```

```
        'Private key
        Dim slt(0) As Byte
        Dim pdb As New PasswordDeriveBytes(strKey24, slt)
        Dim bytDerivedKey() As Byte = pdb.GetBytes(24)
        crp.Key = bytDerivedKey

        'Initialization vector
        crp.IV = pdb.GetBytes(8)
        'Create a crypto stream decoder to decode
        'a cipher text stream into a plain text stream
        Dim csDecrypted As New CryptoStream(stmCipherText, _
          crp.CreateDecryptor(), CryptoStreamMode.Read)
        Dim sw As New StreamWriter(stmPlainText)
        Dim sr As New StreamReader(csDecrypted)
        sw.Write(sr.ReadToEnd)
        'Clean up afterwards
        sw.Flush()
        csDecrypted.Clear()
        crp.Clear()
        Return uEncode.GetString(stmPlainText.ToArray())
    End Function
  End Module
End Namespace
```

You can use these two functions in your code to encrypt and decrypt messages. The key is named *strKey24* because it must be 24 characters long.

3. Open the MainModule.vb file, and in *Sub Main()*, change the line

```
EncryptField("Password", "PasswordHash")
```

to read

```
EncryptField("BankAccount", "BankAccountEncrypted")
```

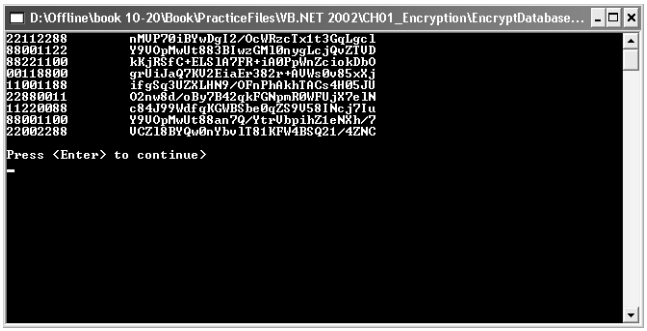
4. In *Sub EncryptField()*, find the line that reads

```
strCipherText = HashCreateHash(strPlainText)
```

and change it to the following:

```
strCipherText = PrivateKey.Encrypt(strPlainText, _
    "111222333444555666777888")
```

5. Now press F5 to run the program. The BankAccountEncrypted field will now contain the bank account information encrypted with the key 111222333444555666777888, and you should see output similar to what is shown here:



Store and retrieve account information using encryption

Next you'll change the employee management system to store and retrieve the bank account number using private key encryption.

1. In Visual Studio .NET, open the project CH01_Encryption\EMS\Start\EMS.sln. Open MainModule.vb, and add the following line to the Declarations section:

   ```
   Public G_PRIVATEKEY As String = "111222333444555666777888"
   ```

   This is the global variable you'll use to store the private key.

2. Open the class *clsEmployee*, and find the declaration

   ```
   Private m_BankAccount As String
   ```

   Change it to

   ```
   Private m_BankAccountEncrypted As String
   ```

3. In the property *Get* of *BankAccount*, change the line that reads

   ```
   Return m_BankAccount
   ```

   to

   ```
   Return PrivateKey.Decrypt(m_BankAccountEncrypted, G_PRIVATEKEY)
   ```

4. In the property *Set* of *BankAccount*, change the line that reads

   ```
   m_BankAccount = Value
   ```

   to

   ```
   m_BankAccountEncrypted = PrivateKey.Encrypt(Value, G_PRIVATEKEY)
   ```

5. In the *Create* function, change the line that reads

   ```
   Me.m_BankAccount = CStr(dr("BankAccount"))
   ```

to

```
Me.m_BankAccountEncrypted = CStr(dr("BankAccountEncrypted"))
```

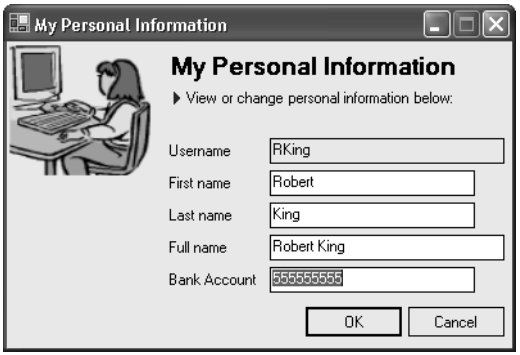
6. In the function *SaveToDatabase*, change the lines that read

```
Dim strSQL As String = "UPDATE Employee SET " & _
  "FirstName      ='" & Me.FirstName & "'," & _
  "LastName       ='" & Me.LastName & "'," & _
  "Fullname       ='" & Me.FullName & "'," & _
  "BankAccount ='" & Me.m_BankAccount & _
  "' WHERE Username ='" & Me.Username & "'"
```

to

```
Dim strSQL As String = "UPDATE Employee SET " & _
  "FirstName      ='" & Me.FirstName & "'," & _
  "LastName       ='" & Me.LastName & "'," & _
  "Fullname       ='" & Me.FullName & "'," & _
  "BankAccountEncrypted ='" & Me.m_BankAccountEncrypted & _
  "' WHERE Username ='" & Me.Username & "'"
```

7. Now press F5 to run the application. Log on using the username RKing and the password RKing. On the dashboard, click the View Or Change Personal Information button. On the My Personal Information form, you can change bank account information. Click OK to save the account to the database in encrypted format, as shown here:



Keeping Private Keys Safe

The Triple-DES encryption algorithm we use accepts a 24-character string for a key. The 24 characters are treated as a *passphrase* that is used to derive a 192-bit byte array, which is then used as the actual key. This is known as 192-bit encryption. The number of bits in the key determines the total combination of possible keys—for example, a 192-bit key has 6.3×10^{57} possible values. A common method intruders use to try to crack encryption is a brute force attack, which means trying every different key combination available until they find the key that works. The more bits in the key, the longer it takes for a brute force attack to find the key. An intruder using the latest hardware would take a long time to crack a 192-bit key—supposing the intruder can try 1,000,000,000,000 keys a second, it would take about 200,000,000,000,000,000,000,000,000,000,000,000,000 years to try every combination. Even if the intruder got lucky, and found the key after trying only 0.0000000001% of the available combinations, the task would still take trillions of years.

Another method intruders use for cracking encryption is to find where the key is stored and then simply read the key. How can you store the key to protect against this? The least secure method is to store the key unencrypted in a file or in the registry accessible to everyone, since if an intruder gains access to your machine, all he needs is notepad.exe to read the file or RegEdit.exe to read the registry. Hard-coding the key in the application (as the employee management system currently does) is also not a good idea since if an intruder gets a copy of your application, he could easily use a de-compiler or debugger to find the key. A better method is to encrypt the key and store it in a file that is protected by the file system so that only authorized users of the system can read it. This immediately raises the questions of where to store the key you use to encrypt the private key? Windows helps with this by providing methods for encrypting and decrypting sensitive data by using logon credentials as a key. When using these methods, there are several things to be aware of:

- **Data encrypted by one user cannot be decrypted by another user.** If several people share the same computer, each person will need to have her own separate copy of the encrypted data because one person's logon credentials can't be used to decrypt data encrypted with another person's logon credentials.
- **Directory Security.** You can make this technique even more secure by storing the encrypted data in a directory that only the current user has access to. In the following exercise, you'll store the

encrypted key in the Application Data directory, which is different for each user.

- **Installing.** If you're using this technique to install a predefined value such as a private key, consider how you will install the value in the first place. One option is to provide a key-installer program that can be run from the server to install the key. You should ensure that only authorized users of the application have permission to view or run the program that installs the key. Also, you should consider removing access to it after the key has been installed.

While these techniques are great for storing private keys, they can be used for any sensitive information such as connection strings and credit card information.

Encrypt the private key

In this exercise, you will encrypt the private key and store it in the application directory. You will also change the employee management system to retrieve the private key from the encrypted file.

1. Start Visual Basic .NET, and load the solution CH01_Encryption\InstallKey\Start\InstallKey.sln.
2. Open MainModule.vb, and insert the following code:

```
'Insert code below...
Public G_PRIVATEKEY As String = "111222333444555666777888"
Sub Main()
  'Encrypt the key and store it in the location
  'c:\Documents And Settings\<username>\Application Data\emsKey
  Settings.SaveEncrypted("EMSKey", G_PRIVATEKEY)
  MsgBox("Done")
End Sub
```

3. Open SecurityLibrary.vb, and move to the end of the file. You are about to add the necessary code to easily use the Windows *CryptProtectData* and *CryptUnprotectData* APIs. This is 120 lines of code, so it will be easiest to simply cut and paste it in. In the same directory as InstallKey.sln, you will find a text file named LoadAndSaveSettings.txt. Open this file, and copy and paste the contents at the end of SecurityLibrary.vb.
4. Press F5 to run the application. It will install a file named EMSKey.txt in the Application Data directory, which is usually c:\Documents And Settings\<username>\Application Data\EMSKey.txt.

5. Now that the key is installed, you need to change the employee management system to use the encrypted key. In Visual Basic .NET, open the solution CH01_Encryption\ EMS\Start\EMS.sln.

6. Open MainModule.vb, find the line that reads

```
Public G_PRIVATEKEY As String = "1122334455667788"
```

and change it to

```
Public G_PRIVATEKEY As String = Settings.LoadEncrypted("EMSKey")
```

7. Press F5 to run the application. Now the private key is being loaded from an encrypted file.

One final note on private keys: In your own applications, you should create a private key that is more complicated than the 1112223334445556667778888 used in this example—private keys should be a random string of characters, numbers, and punctuation.

Public Key Encryption

Public key encryption (also called asymmetric encryption) has an important difference from private key encryption. Public key encryption uses two different keys: one key for encryption and another key for decryption. Why don't they simply call this two-key encryption and call private key encryption one-key encryption? While it is well known that security experts like to invent jargon to justify their high consultancy fees, there is also a logical reason for this naming, which lies in the way the two types of encryption are used.

While private key encryption assumes that both the encrypting and decrypting parties already know the private key, public key encryption provides a method to securely issue a key to someone and have that individual send you information that only you can decrypt. It works like this: Our system creates a public/private key pair. We send the public key to someone who uses it to encrypt a message. She sends the encrypted message to us, and we decrypt the message with the private key. (Note: The private key is not the same as the key used in private key encryption.) Even if an intruder gains possession of the public key, he cannot use it to decrypt the encrypted message because only the private key can decrypt the message, and this is never given away. In contrast with private key encryption, the keys used in public key encryption are more than simple strings. The key is actually a structure with eight fields: two of the fields are used for encrypting with the public key, and six are used for decrypting with the private key. The public key is obtained by extraction from the private key, which is why the private key can be used for both encryption and decryption.

Figure 1-4 shows how public key encryption and decryption work, using the example of a system requesting a credit card number from a user.

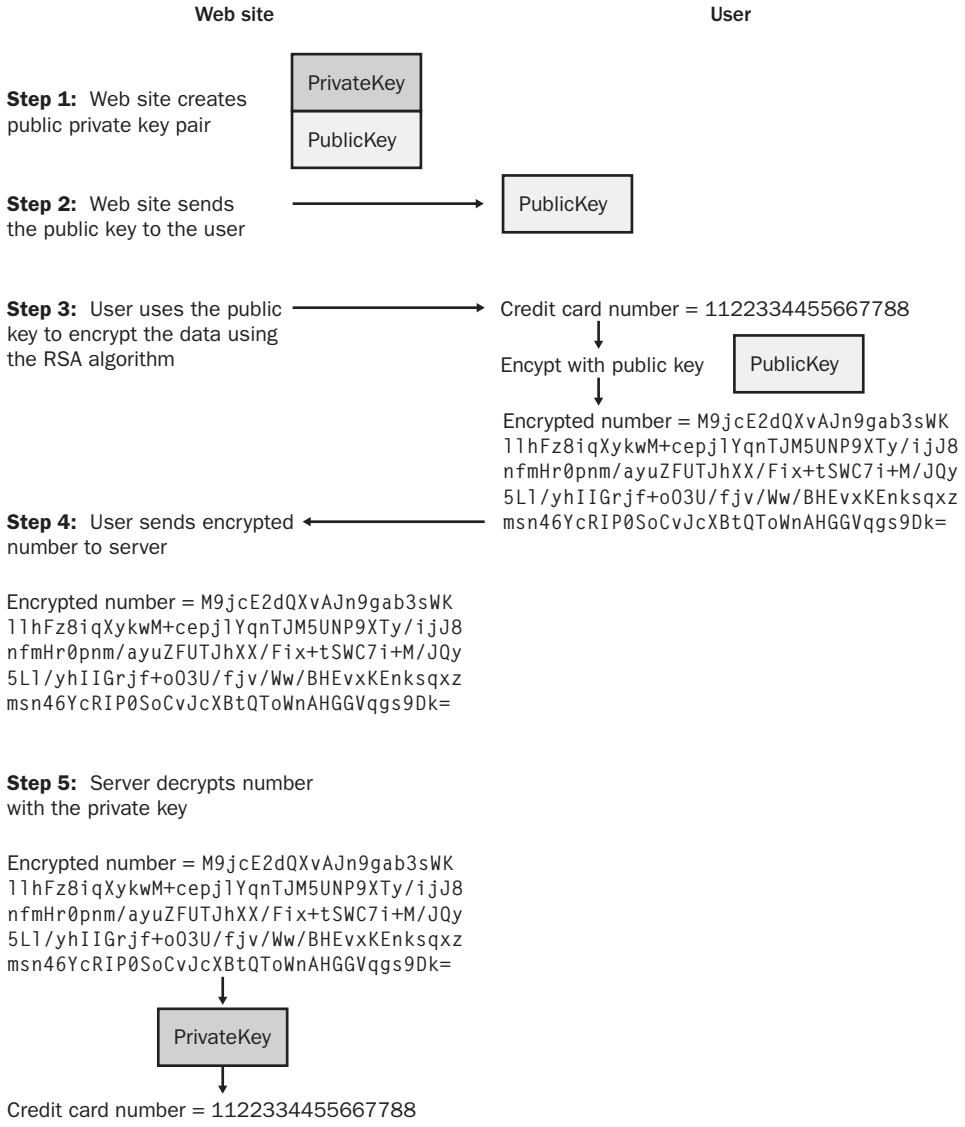


Figure 1-4 Public key encryption and decryption

Public key encryption is slower than private key encryption and cannot process large amounts of data. The RSA algorithm (RSA refers to the initials of the people who developed it: Ron Rivest, Adi Shamir, and Leonard Adleman) can encrypt a message of only 116 bytes (58 unicode characters). A common use for public key encryption is for securely passing a private key, which is then used for encrypting and decrypting other information.

Add public key encryption to the security library

In this exercise, you will add public key encryption functions to your security library.

1. In Visual Studio .NET, open the project CH01_Encryption\EMS\Start\EMS.sln.
2. Open SecurityLibrary.vb. Add the following code:

```
Namespace PublicKey
  Module PublicKey
    Function CreateKeyPair() As String
      'Create a new random key pair
      Dim rsa As New RSACryptoServiceProvider()
      CreateKeyPair = rsa.ToXmlString(True)
      rsa.Clear()
    End Function
    Function GetPublicKey(ByVal strPrivateKey As String) As String
      'Extract the public key from the
      'public/private key pair
      Dim rsa As New RSACryptoServiceProvider()
      rsa.FromXmlString(strPrivateKey)
      Return rsa.ToXmlString(False)
    End Function
    Function Encrypt(ByVal strPlainText As String, _
        ByVal strPublicKey As String) As String
      'Encrypt a string using the private or public key
      Dim rsa As New RSACryptoServiceProvider()
      Dim bytPlainText() As Byte
      Dim bytCipherText() As Byte
      Dim uEncode As New UnicodeEncoding()
      rsa.FromXmlString(strPublicKey)
      bytPlainText = uEncode.GetBytes(strPlainText)
      bytCipherText = rsa.Encrypt(bytPlainText, False)
      Encrypt = Convert.ToBase64String(bytCipherText)
      rsa.Clear()
    End Function
    Function Decrypt(ByVal strCipherText As String, _
    ByVal strPrivateKey As String) As String
      'Decrypt a string using the private key
```

```
        Dim rsa As New RSACryptoServiceProvider()
        Dim bytPlainText() As Byte
        Dim bytCipherText() As Byte
        Dim uEncode As New UnicodeEncoding()
        rsa.FromXmlString(strPrivateKey)
        bytCipherText = Convert.FromBase64String(strCipherText)
        bytPlainText = rsa.Decrypt(bytCipherText, False)
        Decrypt = uEncode.GetString(bytPlainText)
        rsa.Clear()
    End Function
  End Module
End Namespace
```

Export Restrictions on Encryption

In June 2002, the United States Bureau of Industry and Security eased restrictions for companies that export software products containing encryption. Software that uses private key encryption with keys of more than 64 bits can be exported without a license to many destinations following a 30-day review period. For full details, see the Bureau of Industry and Security encryption Web site at *http://www.bxa.doc.gov/Encryption/*.

Hiding Unnecessary Information

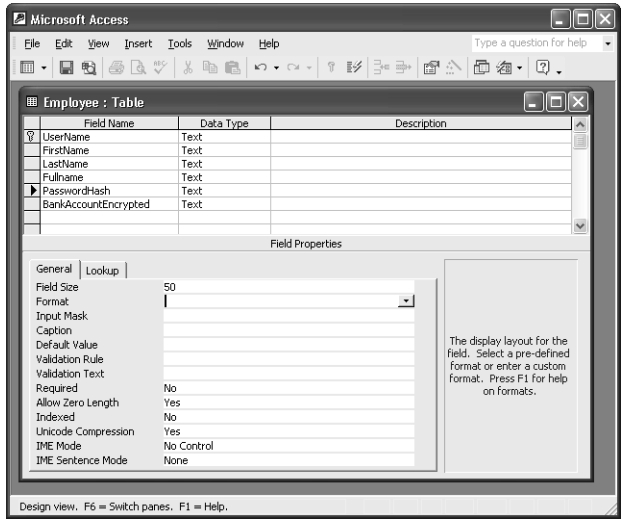
Now that you have encrypted the passwords and bank account information, you should do two more encryption-related things to further secure the employee management system: remove the unencrypted password field and the unencrypted bank account field from the Employees table, and protect the password entry field in the logon screen.

Remove the Password and BankAccount fields

The unencrypted Password and BankAccount fields are no longer needed in the EmployeeDatabase.mdb database. In this exercise, you will remove these two fields from the database.

> **Note** This is an optional exercise. Don't worry if you don't have Microsoft Access; the other exercises in this book will still work.

1. In Microsoft Access XP, open the database EmployeeDatabase.mdb.
2. In the Database window, select the table Employee and click Design on the Database Window toolbar.
3. Select the Password field's row selector, and click Delete Row on the Microsoft Access toolbar. Microsoft Access will then ask you to confirm that you really want to delete the row and all the data it contains. Click Yes.
4. Select the BankAccount field's row selector, and again click Delete Row on the Microsoft Access toolbar. Again, click Yes in the dialog box that asks you to confirm the field deletion.
5. Click the Save button on the toolbar to save the table changes. The new table design should look like the following illustration:



Hide the password entry field

If someone is looking over your shoulder while you're logging on to the employee management system, that person might be able to read your password as you type it. Windows Forms has the capability of hiding the password as you type it. The following exercise describes the steps necessary to hide the password.

1. In Visual Studio .NET, open the project CH01_Encryption\EMS\Start\EMS.sln.

2. Open the form frmLogin.vb in the Windows Forms designer.
3. In the form designer, select the password field *txtPassword*.
4. In the property browser, find the *PasswordChar* property, and change the value to *.

   After you complete these steps, the password entry field will appear as a series of asterisks instead of text, as shown here:

Encryption in the Real World

At the end of most chapters in this book, you'll find a section like this one that explores where you might use techniques learned in the chapter in your own real-world projects. Encryption has a number of uses but two main purposes:

- Securely storing sensitive information on a disk or in a database so that it can be accessed only by an authorized person or software program.
- Scrambling information so it can be transported from one trusted system to another trusted system over an insecure transport such as the Internet. Some specific examples are listed here:
  - **Authenticating passwords.** This can be done using either a hash digest or a private key. Hash digests are a good choice when the password is used only for validating the login. If, however, the password is used for connecting to a database, private key encryption is the better method because the system needs to use the unencrypted string.

- **Verifying the integrity of a file.** Because a hash digest is a unique signature, it can be used to verify that a piece of information, such as a file, is unchanged. For example, you can send an XML file through the Internet and then send the hash of the file; in this way, the recipient can verify that the file wasn't corrupted during transmission.[2]
- **Storing and retrieving sensitive information in a file, registry, or database.** Private key encryption is a good method for two-way encryption of information when both the encrypting and decrypting parties know the key.
- **Transmitting secret information over the Internet.** Private key encryption is good for passing secret information over the Internet, provided both parties already know the key. Public key encryption can also be used, but it's slower and subject to size limitations.
- **Receiving private information, such as private user information over an intranet, extranet, or the Internet.** Public key encryption is a great way to get information from someone who doesn't already possess a private key. The ultimate recipient of the information creates a key pair and sends the public key to the sender of the information. The sender encrypts the information and then submits it to the recipient, who uses the private key to decrypt it.

Summary

In this chapter, we jumped right into encryption and created a library of functions for creating hash digests as well as for encryption and decryption using private and public keys, all with a single line of Visual Basic code. In addition, we started securing the employee management system with minimal impact on usability and programming time. This illustrates an important point: if security is complicated to implement or use, people won't implement or use it. The purpose of this and future chapters is to show you techniques that are simple to bolt onto your existing applications and that have minimal impact on usability.

2. Be aware that this is not a guarantee against tampering—an intruder could modify the file and then create a hash of the modified file.