l e x a n a \ n e t

June 2005

File Encryption in .NET using *TripleDES* and *Blowfish*

By Drew Hamre

email

The concept of *symmetric encryption* is perhaps familiar from childhood: friends share a single code book, and use the codes to both encrypt and decrypt secret messages. This technique is sometimes called *shared key* encryption, because the same password and process is used both to hide secrets and reveal them.

The simplicity of symmetric encryption belies its utility. Today, businesses commonly use symmetric encryption to encode communications among internal systems, or to encrypt archival data. Below, we'll discuss file encryption using two popular symmetric algorithms: *TripleDES* (included with the .NET Framework) and *Blowfish* (a widely-available third-party algorithm).

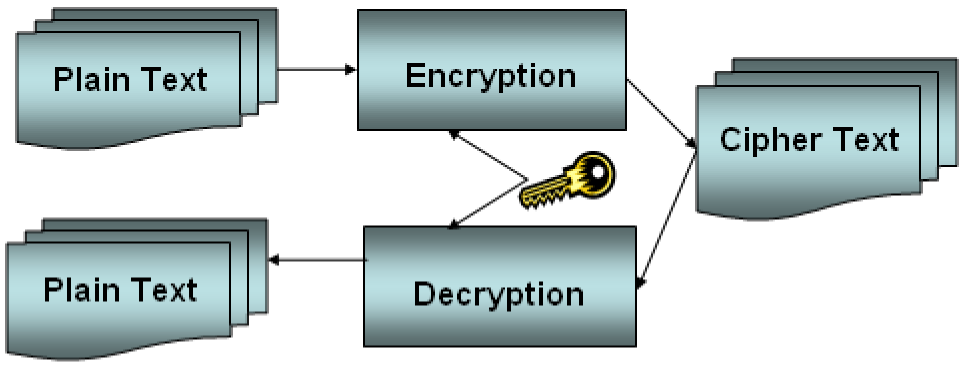


Figure 1 - Symmetric (or *shared key*) encryption

We'll also describe a simple *Blowfish*-based system that was implemented for a commercial bank. The system provided secure communications between legacy UNIX and Windows servers that were completely isolated, except for SMTP email.

Beyond .NET encryption: Speed, source-code, and interoperability

Microsoft's .NET Framework includes comprehensive, well-tested cryptographic libraries for symmetric (shared key) encryption, asymmetric (public key) encryption, and hash digests. Given the functionality built into .NET, is it worthwhile to consider non-standard cryptographic software?

Some considerations that might justify the effort and risk in extending .NET's standard cryptographic libraries: *speed*, *source code*, and *interoperability*.

Speed – Modern encryption algorithms are CPU intensive. While symmetric algorithms are generally speedier than asymmetric (as much as *10-times faster*, in many cases), there are significant differences among symmetric alternatives. The symmetric options offered by Microsoft (*DES*, *Triple-DES*, *RC2*, and *Rijndael*) are not necessarily the fastest algorithms nor fastest implementations. If massive amounts of data are encoded in near real-time, processing efficiencies become critical and the incentive is strong to look beyond the standard libraries.

Source Code –You'd expect that cryptographic secrecy would extend to source code, and that algorithms and implementations would remain a closely-guarded black box. The reverse is true: it's considered good practice among cryptographers to publish all encryption algorithms so they can be scrutinized and vulnerabilities can be assessed. (If you hear echoes of the arguments for 'open source' software, you're probably right.)

Note that publishing an encryption algorithm poses no risk. The strength of modern encryption algorithms is intrinsic in the mathematics. Encrypted data is vulnerable only to someone knowing the key, not to someone knowing the algorithm.

In addition to publishing algorithms, it's also common for implementers to make cryptographic source code available to scrutiny. Note that Microsoft has not yet followed suit. While Microsoft has released source code for its .NET crypto interfaces, its core encryption code for .NET has not been similarly published.

Source code is a boon for developers, and makes it possible (ironically) to plug gaps within Microsoft's own cryptography lineup. For example, the .NET Compact Framework doesn't support the *Security.Cryptography* namespace. However, third-party source is now available for managed code extensions, allowing applications to offer algorithms such as AES / Rijndael[i] (the default for WSE 2.0 security).

Interoperability - When trading encrypted communications (or encrypting and decrypting files) both ends of the transaction must translate every bit (literally) of data according to the same binary conventions. This may pose problems when sharing information across unlike platforms, as systems disagree about byte-orders within words (the 'big-endian' – 'little-endian' problem).

One way of assuring cross-platform interoperability is to use identical implementations of an algorithm, written in a platform-neutral language like Java or C. Toward this end, there are portable implementations of many encryption algorithms, including Blowfish (discussed below). If source code is available, then a single encryption library can be installed and used and on any platform that supports the implementation language.

We'll revisit the potential benefits of 'non-standard' .NET encryption methods, below. First, though, we'll describe using one of .NET's standard cryptographic libraries to write a custom utility to encrypt and decrypt files.

Using .NET symmetric algorithms to encrypt and decrypt files

One of the popular uses for symmetric encryption is to protect archival data[ii]. File encryption is built into several versions of Windows (including XP Pro) as the

Encrypting File System (EFS). However, EFS[1] works only on NTFS and offers no protection from someone who's gained access to your logged-on session. For this reason, hard disk encryption utilities are widely popular.

Some encryption utilities offer the same transparency of operation as EFS, while extending the breadth of covered files and offering two-factor authentication. Of course, transparent operation (where authentication derives chiefly from the session context) presents the same vulnerability as EFS.

Other 'non-transparent' utilities must be run manually on a file-by-file basis, using a password supplied at runtime. .NET's cryptographic libraries can also be used to create such a utility. Below we'll discuss a sample encryption program that prompts the user for an alphanumeric password, and then uses the password in conjunction with a symmetric encryption algorithm to encode (or decode) a selected file.

Once encrypted, the target file can only be read by someone using the same password and crypto algorithm to decrypt the file. To simplify software distribution, the decryption logic and the encrypted file's data may be packaged together as self-extracting executable.

To write custom file encryption software, you can use the .NET symmetric encryption methods as follows (a variation of an example by Richard Mansfield):

- Write a UI to solicit two user-supplied parameters: input file path and password
- Cryptographically strengthen the user-entered alphanumeric password by mathematically hashing its value into a byte array. Internally, you'll use the hash value as the key to encrypt and decrypt the data. (Don't worry: the same password and hash method will always emit the same internal key.)
- Create a .NET-provided cryptographic object (as we do below with *TripleDESCryptoServiceProvider*), and then create a corresponding .NET cryptographic stream. In creating these objects, you'll need to supply the byte array containing the hashed key plus an *initialization vector* (IV). (The IV is a sequence of random bytes appended to the front of the plaintext. Adding the IV to the beginning of the plaintext reduces the possibility of having the initial cipher text block identical for any two files.)
- Associate an output file stream with the cryptographic stream, and read/write blocks of data until the input is exhausted.

```
‘ Based on Richard Mansfield ‘s “Keeping Secrets …”
Try
     ' Declare input/output file streams and save size
      Dim fin As New FileStream(inName, FileMode.Open, FileAccess.Read)
      Dim fout As New FileStream(outName, FileMode.OpenOrCreate,
FileAccess.Write)
      fout.SetLength(0)
      Dim totalFileLength As Long = fin.Length

      ' Create cryptographic object and stream
      Dim des As New TripleDESCryptoServiceProvider()
      Dim crStream As New CryptoStream(fout, _
```

[1] EFS combines both symmetric and asymmetric encryption.

```
            des.CreateEncryptor(TheKey, Vector), CryptoStreamMode.Write)

        ' Read/write the files
        While totalBytesWritten < totalFileLength
            packageSize = fin.Read(storage, 0, 4096)
            crStream.Write(storage, 0, packageSize)
            totalBytesWritten = totalBytesWritten + packageSize
        End While
        crStream.Flush()
        crStream.Close()

Catch e As Exception
        MsgBox(e.Message)
End Try
```

The .NET cryptographic object model makes it extremely easy to swap algorithms. Note the original sample code used DES, while we've 'upgraded' to TripleDES with only a few changes (though take care to adjust any internal IV and key-size parameters, accordingly). In similar fashion, we can extend our .NET code to call a *non*-standard encryption library, such as Blowfish.

The *Blowfish* symmetric encryption algorithm

The Blowfish algorithm was designed in 1993 by mathematician/cryptographer Bruce Schneier[2], author of Applied Cryptography (the discipline's standard textbook) as well as other books on security issues. Blowfish is a symmetric block cipher that can be used as a drop-in replacement for algorithms like DES or IDEA. (Block ciphers encrypt data in discrete blocks; a chunk of plaintext is read, and then encrypted.) Unlike IDEA (which is patented), Blowfish is "unpatented and license-free, and is available free for all uses".

The Blowfish algorithm has been scrutinized by cryptographers for more than a decade, and its known weaknesses are limited to constrained cases where the number of encryption rounds is limited. Blowfish has recently been added to the Linux kernel, and the encryption technique is now so well-known that it was the subject of a code-busting subplot during the popular spy series, "24".

While Blowfish source code is available for many implementations, two resources are especially valuable to .NET developers. Markus Hahn's site provides C# source code for Blowfish encryption, along with validation instructions. The Flowgroup has extended Hahn's code around a *Stream*-derived class for ease of use.

Benchmarking the encryption utility

Encryption performance is hotly contested among both software and hardware engineers. Alternative algorithms may respond differently to seemingly subtle hardware changes. The latest generation of Wintel CPUs (especially Itanium, but also the AMD/Intel extended 64-bit designs) include design considerations that are tailored to improve cryptographic performance.

Because encryption algorithms show varying sensitivity to such design changes, cryptographic benchmarking will be an ongoing process. The Windows OS also adds measurement variability due to process scheduling, data-caching, NTFS-background

[2] Schneier is the founder and CTO of Counterpane, a leading managed-security vendor. Schneier's monthly newsletter, Crypto-gram, is one of the most valuable sources of security-related news.

processing, and CPU throttling related to power management. Suffice it to say that all cryptographic benchmarks should be deeply etched: "Your mileage may vary."

Blowfish is considered among the top performing algorithms. Benchmarks generally show Blowfish performing on par with TripleDES, though both are typically slower than AES/Rijndael. Of course the real issue is performance on our specific system, which we can test with a simple test harness:

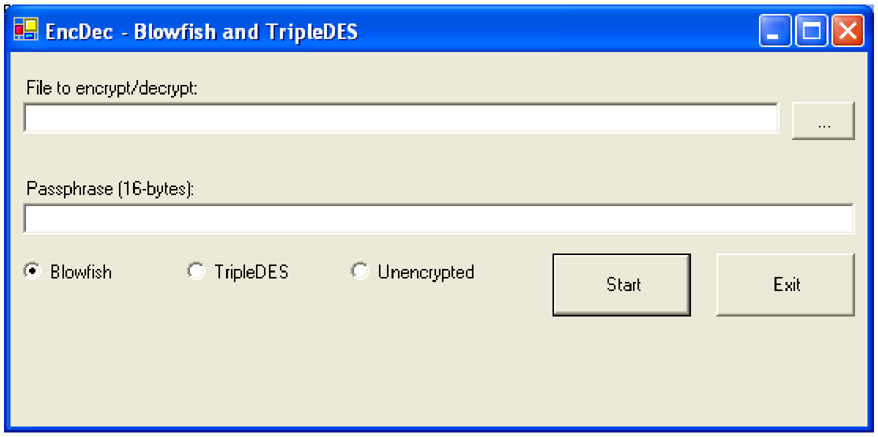
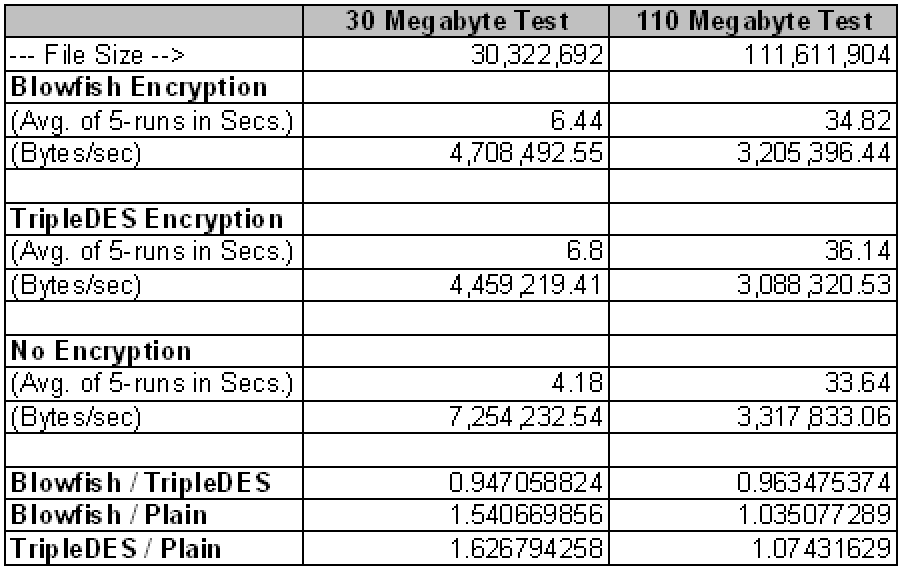


Figure 2 - Test harness for encryption utility

To assess encryption/decryption performance, we'll first embed a high-resolution timer into the code. Then we'll compare timings for straight I/O (writes to an unencrypted output file) versus encrypted I/O (writes to an encrypted output file) using both TripleDES and Blowfish. We'll use different copies of our test file on alternate runs, to lessen the effect of caching, and then average the results.

On a Pentium 4 laptop, we find the standard .NET TripleDES library consistently takes a slightly longer time to encrypt than the Hahn/Flowgroup implementation of Blowfish. However, variability from run to run was such that the Blowfish advantage was not statistically significant. As expected, straight I/O was consistently faster than encrypted I/O. The overhead stemming from encryption was far more apparent for the smaller file than the larger, perhaps because for the larger file the laptop's pronounced I/O limitations effectively masked the CPU overhead incurred by encryption.



	30 Megabyte Test	110 Megabyte Test
--- File Size -->	30,322,692	111,611,904
Blowfish Encryption		
(Avg. of 5-runs in Secs.)	6.44	34.82
(Bytes/sec)	4,708,492.55	3,205,396.44
TripleDES Encryption		
(Avg. of 5-runs in Secs.)	6.8	36.14
(Bytes/sec)	4,459,219.41	3,088,320.53
No Encryption		
(Avg. of 5-runs in Secs.)	4.18	33.64
(Bytes/sec)	7,254,232.54	3,317,833.06
Blowfish / TripleDES	0.947058824	0.963475374
Blowfish / Plain	1.540669856	1.035077289
TripleDES / Plain	1.626794258	1.07431629

Even in the worst case, the encryption ‘surcharge’ seems not a bad penalty, perhaps, for adding Blowfish or TripleDES security that is widely held to be unbreakable by current technology.

Using *Blowfish* for the automatic decryption of email attachments

The custom commercial banking software mentioned above allowed remote administration of a Windows NT domain via encrypted email:

- Domain administration requests were issued from a UNIX-based application. Custom software formatted the request as an XML document, encrypted the resulting file using Blowfish, and forwarded the file as an email attachment via SMTP to a ‘mail drop’ (special public folder) on a Microsoft Exchange Server.
- The Exchange Server mail drop was associated with scripts that activated upon message receipt. The code would retrieve the attachment, decrypt the XML document using Blowfish, and then perform the embedded administrative requests. Supported tasks included creating domain accounts, deleting accounts, and changing account passwords.

To assure byte-by-byte compatibility across UNIX and Windows platforms, a Java-based Blowfish implementation was used. This single Blowfish source code library therefore supported encryption and decryption on both platforms. Note there are also *C*-language implementations of Blowfish that would offer the same portability.

On Exchange Server 5.5, a script agent was created to hook the *Folder_OnMessageCreated* event. (Had this been Exchange Server 2000 or 2003, the folder’s *OnSave* event sink would be used.) The event script used CDO to save the encrypted to the file system, and invoke the Blowfish component.

Summary

This paper looked at alternatives for file encryption, including .NET’s built-in *TripleDES* algorithm and custom implementations of the *Blowfish* algorithm. The Blowfish alternative demonstrates reasonable encryption/decryption speeds, and also interoperates with different operating systems and processor architectures.

Resources

The Blowfish Encryption Algorithm by Bruce Schneier
http://www.schneier.com/blowfish.html

Keeping Secrets: A Guide to VB .NET Cryptography by Richard Mansfield
http://archive.devx.com/security/articles/rm0802/rm0802.asp

Platform Neutral and Transparent Encryption … by Zhenlei Cai
http://www.15seconds.com/issue/030310.htm

Coder's Lagoon (including "Blowfish.NET 1.01" and "BlowfishJ 2.14") by Markus Hahn
http://maakus.dyndns.org/software.html

Architectural Support for Fast Symmetric-Key Cryptography (Burke, McDonald, Austin)
http://www.princeton.edu/~rblee/ELE572Papers/ArchFastSymmetricKey-austin.pdf

Encrypting with the Compact Framework on a PocketPC using a "BlowFish" Stream
http://www.flowgroup.fr/tech_blowfish_us.htm

Acknowledgement

Thanks to reviewers for their time and suggestions; any inaccuracies remain mine.

Drew Hamre is a principal of lexana\net and lives in Golden Valley, Minnesota.

---

[i] In 2001, the U.S. government adopted the Rijndael algorithm as its new Advanced Encryption Standard (AES), replacing DES. *Rijndael* takes its name from its inventors (Joan Daemen and Vincent Rijmen), and is pronounced "Rhine dahl".

Cryptography is a volatile field, however, and in April 2005 – only two years after NSA approval - cryptographer D.J. Bernstein claimed his ‘cache timing attack’ could break most real-world AES implementations, including OpenSSL.

[ii] The importance of persisting information in *encrypted* form has been driven home by recent data thefts from corporations including MCI, Bank of America and Wachovia. Data protection laws in California (SB1386) stipulate that consumers must be notified if their sensitive information is stolen.

SB1386 lets companies forego notifying customers if the stolen data was encrypted.