Microsoft® SQL Server™ 2008

1 Database Encryption in SQL Server 2008 Enterprise Edition

SQL Server Technical Article

Writers: Sung Hsueh

Technical Reviewers: Raul Garcia, Sameer Tejani, Chas Jeffries, Douglas MacIver, Byron Hynes, Ruslan Ovechkin, Laurentiu Cristofor, Rick Byham, Sethu Kalavakur

Published: February 2008

Applies To: SQL Server 2008 (pre-release version)

Summary: With the introduction of transparent data encryption (TDE) in SQL Server 2008, users now have the choice between cell-level encryption as in SQL Server 2005, full database-level encryption by using TDE, or the file-level encryption options provided by Windows. TDE is the optimal choice for bulk encryption to meet regulatory compliance or corporate data security standards. TDE works at the file level, which is similar to two Windows® features: the Encrypting File System (EFS) and BitLocker™ Drive Encryption, the new volume-level encryption introduced in Windows Vista®, both of which also encrypt data on the hard drive. TDE does not replace cell-level encryption, EFS, or BitLocker. This white paper compares TDE with these other encryption methods for application developers and database administrators. While this is not a technical, in-depth review of TDE, technical implementations are explored and a familiarity with concepts such as virtual log files and the buffer pool are assumed. The user is assumed to be familiar with cell-level encryption and cryptography in general. Implementing database encryption is covered, but not the rationale for encrypting a database.

Copyright

Table of Contents

Introduction: Encrypting at the Database Level

Transparent data encryption (TDE) is a new encryption feature introduced in Microsoft® SQL Server™ 2008. It is designed to provide protection for the entire database at rest without affecting existing applications. Implementing encryption in a database traditionally involves complicated application changes such as modifying table schemas, removing functionality, and significant performance degradations. For example, to use encryption in Microsoft SQL Server 2005, the column data type must be changed to **varbinary**; ranged and equality searches are not allowed; and the application must call built-ins (or stored procedures or views that automatically use these built-ins) to handle encryption and decryption, all of which slow query performance. These issues are not unique to SQL Server; other database management systems face similar limitations. Custom schemes are often used to resolve equality searches and ranged searches often cannot be used at all. Even basic database elements such as creating an index or using foreign keys often do not work with cell-level or column-level encryption schemes because the use of these features inherently leak information. TDE solves these problems by simply encrypting everything. Thus, all data types, keys, indexes, and so on can be used to their full potential without sacrificing security or leaking information on the disk. While cell-level encryption cannot offer these benefits, two Windows® features, Encrypting File System (EFS) and BitLocker™ Drive Encryption, are often used for the same reasons as TDE—they provide protection on a similar scale and are transparent to the user.

Microsoft SQL Server Encryption

Microsoft SQL Server offers two levels of encryption: database-level and cell-level. Both use the key management hierarchy.

Cryptographic Key Hierarchy

At the root of encryption tree is the Windows Data Protection API (DPAPI), which secures the key hierarchy at the machine level and is used to protect the service master key (SMK) for the database server instance. The SMK protects the database master key (DMK), which is stored at the user database level and which in turn protects certificates and asymmetric keys. These in turn protect symmetric keys, which protect the data. TDE uses a similar hierarchy down to the certificate. The primary difference is that when you use TDE, the DMK and certificate must be stored in the master database rather than in the user database. A new key, used only for TDE and referred to as the *database encryption key (DEK)*, is created and stored in the user database.

This hierarchy enables the server to automatically open keys and decrypt data in both cell-level and database-level encryption. The important distinction is that when cell-level encryption is used, all keys from the DMK down can be protected by a password instead of by another key. This breaks the decryption chain and forces the user to input a password to access data. In TDE, the entire chain from DPAPI down to the DEK must be maintained so that the server can automatically provide access to files protected by TDE. In both cell-level encryption and TDE, encryption and decryption through these keys is provided by the Windows Cryptographic API (CAPI).

The following figure shows the full encryption hierarchy. The dotted lines represent the encryption hierarchy used by TDE.

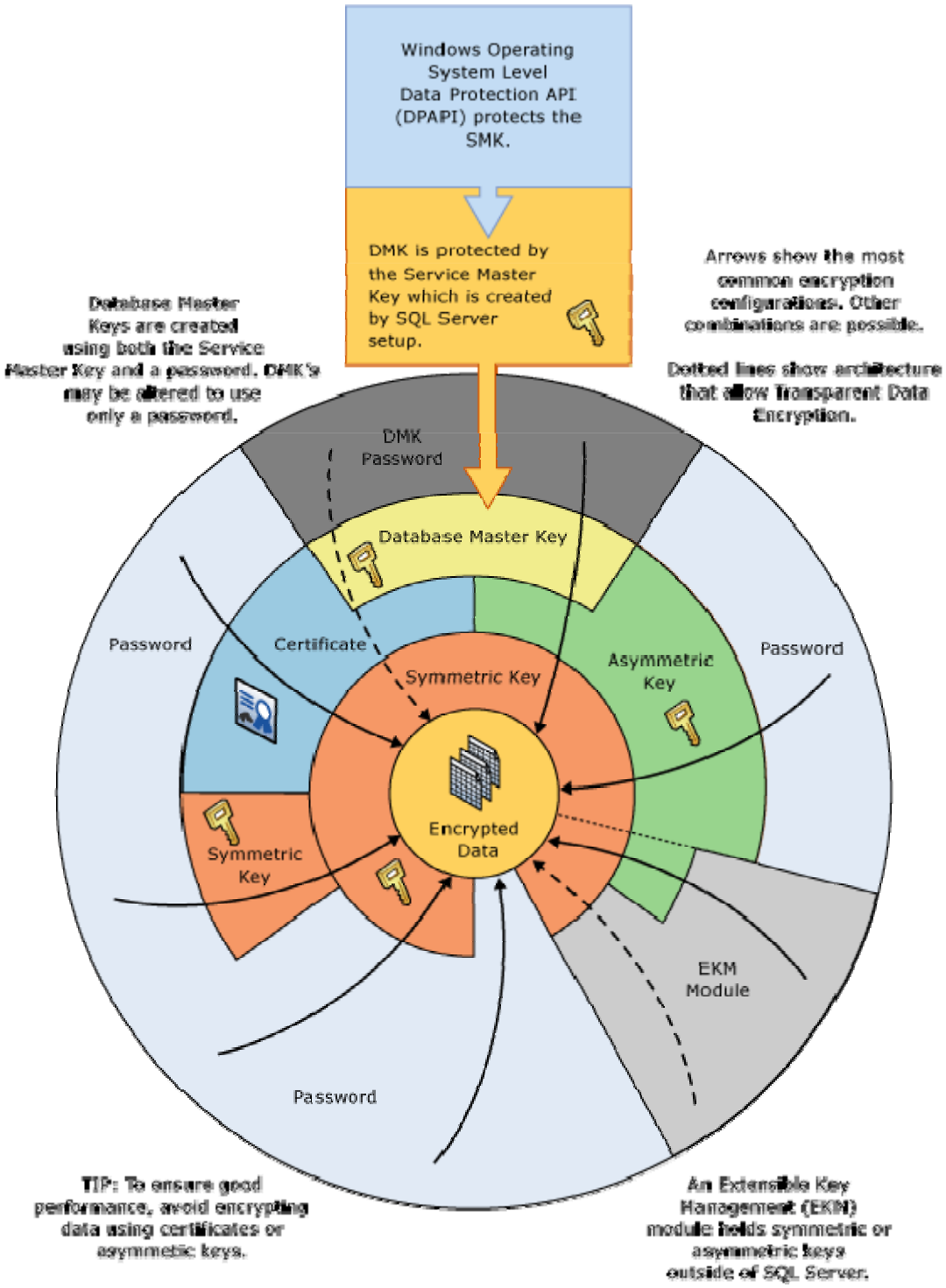

Figure: SQL Server encryption key hierarchy with TDE and EKM

TDE

Transparent data encryption is the new database-level encryption feature introduced in SQL Server 2008.

How to Enable TDE

To enable TDE, you must have the normal permissions associated with creating a database master key and certificates in the **master** database. You must also have CONTROL permissions on the user database.

To enable TDE

Perform the following steps in the **master** database:

1. If it does not already exist, create a database master key (DMK) for the master database. Ensure that the database master key is encrypted by the service master key (SMK).

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'some password';
```

2. Either create or designate an existing certificate for use as the database encryption key (DEK) protector. For the best security, it is recommended that you create a new certificate whose only function is to protect the DEK. Ensure that this certificate is protected by the DMK.

```
CREATE CERTIFICATE tdeCert WITH SUBJECT = 'TDE Certificate';
```

3. Create a backup of the certificate with the private key and store it in a secure location. (Note that the private key is stored in a separate file—be sure to keep both files). Be sure to maintain backups of the certificate as data loss may occur otherwise.

```
BACKUP CERTIFICATE tdeCert TO FILE = 'path_to_file'
   WITH PRIVATE KEY (
         FILE = 'path_to_private_key_file',
         ENCRYPTION BY PASSWORD = 'cert password');
```

4. Optionally, enable SSL on the server to protect data in transit.

Perform the following steps in the user database. These require CONTROL permissions on the database.

5. Create the database encryption key (DEK) encrypted with the certificate designated from step 2 above. This certificate is referenced as a *server certificate* to distinguish it from other certificates that may be stored in the user database.

```
CREATE DATABASE ENCRYPTION KEY
   WITH ALGORITHM = AES_256
   ENCRYPTION BY SERVER CERTIFICATE tdeCert
```

6. Enable TDE. This command starts a background thread (referred to as the *encryption scan*), which runs asynchronously.

```
ALTER DATABASE myDatabase SET ENCRYPTION ON
```

   To monitor progress, query the **sys.dm_database_encryption_keys** view (the VIEW SERVER STATE permission is required) as in the following example:

```
SELECT db_name(database_id), encryption_state
   FROM sys.dm_database_encryption_keys
```

How Data is Encrypted

When TDE is enabled (or disabled), the database is marked as encrypted in the **sys.databases** catalog view and the DEK state is set to Encryption In Progress. The server starts a background thread (called the *encryption scan* or *scan*) that scans all database files and encrypts them (or decrypts them if you are disabling TDE). While the DDL executes, an update lock is taken on the database. The encryption scan, which runs asynchronously to the DDL, takes a shared lock. All normal operations that do not conflict with these locks can proceed. Excluded operations include modifying the file structure and detaching the database. While normal database writes to disk from the buffer pool are encrypted, log file writes may not be. The scan also forces a rollover for the virtual log file (VLF) to ensure that future writes to the log are encrypted. This is discussed in more detail later in this white paper.

When the encryption scan is completed, the DEK state is set to the Encrypted state. At this point all database files on disk are encrypted and database and log file writes to disk will be encrypted. Supported encryption algorithms are AES with 128-bit, 192-bit, or 256-bit keys or 3 Key Triple DES. Data is encrypted in the cipher block chaining (CBC) encryption mode. The encrypted database files that are written to disk are the same size as the unencrypted files because no extra padding is required and the initialization vector (IV) and encrypted DEK are stored within the existing space. Because the log is padded to the next VLF boundary, the log will grow in size. Note that while the database state is marked as Encryption enabled, the actual state of the encryption should be monitored through the DEK state. When the background scan is complete the DEK state is set to Encrypted. At this point, future writes to the log and to disk are protected. This is explained in more detail later in this paper.

What Is Encrypted

TDE operates at the I/O level through the buffer pool. Thus, any data that is written into the database file (*.mdf) is encrypted. Snapshots and backups are also designed to take advantage of the encryption provided by TDE so these are encrypted on disk as well. Data that is in use, however, is not encrypted because TDE does not provide protection at the memory or transit level. The transaction log is also protected, but additional caveats apply.

For data that is in use, all pages are decrypted as they are read and stored into the buffer pool and are in clear text in memory. The operating system may page data out of memory as part of memory management. In this process, decrypted data may be written to disk. Windows and SQL Server can be configured to prevent memory from being paged to disk, although the performance cost of this can be high. Other OS actions such as hibernation and crash dumps can also cause memory to be written to disk. Data in transit is not protected because the information is already decrypted before it reaches this point; SSL should be enabled to protect communication between the server and any clients.

When the database page file is written to disk, the headers are not encrypted with the rest of the data because this information is necessary for the page to be reloaded. The header contains status details such as the database compatibility level, database

version, mirroring status, and so forth. Any important data (such as the DEK) is encrypted before it is inserted into the header. The header also includes a data corruption checksum (CRC). Users can have both a checksum on the plaintext and a checksum on the encrypted text. This is not a cryptographic checksum; it detects only data corruption (checking to see if the data is readable) and not data integrity (checking to see if the data was modified). All other user data that is stored in the database page is encrypted, including any unused (or previously deleted) sections of data, to avoid information leakage. When TDE is enabled on any user database, encryption is also automatically enabled for the temporary database (**tempdb**). This prevents temporary objects that are used by the user database from leaking to disk. System databases other than **tempdb** cannot currently be encrypted by using TDE.

Encrypting at the I/O level also allows the snapshots and backups to be encrypted, thus all snapshots and backups created by the database will be encrypted by TDE. The certificate that was used to protect the DEK when the file was written must be on the server for these files to be restored or reloaded. Thus, you must maintain backups for all certificates used, not just the most current certificate.

The transaction log is more complicated. Because the transaction log is designed as a write-once fail safe, TDE does not attempt to encrypt portions of the logs that are already written to disk. Similarly, the log header cannot be re-written because of this write-once principle so there is no guarantee that data that is written to the log even after TDE is enabled will be encrypted. The TDE background scan forces the log to roll over to the next VLF boundary, which allows for the key to be stored in the header. At this point, if the file scan is also complete, the DEK state changes to Encrypted and all subsequent writes to the log are encrypted.

Impact on the Database

TDE is designed to be as transparent as possible. No application changes are required and the user experience is the same whether using a TDE-encrypted database or a non-encrypted database.

While TDE operations are not allowed if the database has any read-only filegroups, TDE can be used with read-only filegroups. To enable TDE on a database that has read-only filegroups, the filegroups must first be set to allow writes. After the encryption scan completes, the filegroup can be set back to read only. Key changes or decryption must be performed the same way.

The performance impact of TDE is minor. Because the encryption occurs at the database level, the database can leverage indexes and keys for query optimization. This allows for full range and equality scans. In tests using sample data and TPC-C runs, the overall performance impact was estimated to be around 3-5% and can be much lower if most of the data accessed is stored in memory. Encryption is CPU intensive and is performed at I/O. Therefore, servers with low I/O and a low CPU load will have the least performance impact. Applications with high CPU usage will suffer the most performance loss, estimated to be around 28%. The primary cost is in CPU usage, so even applications or servers with high I/O should not be too adversely affected if CPU usage is low enough. Also, because the initial encryption scan spawns a new thread, performance is most sharply impacted at this time; expect to see queries perform several orders of magnitude worse. For disk space concerns, TDE does not pad database files on disk although it does pad transaction logs as previously noted in How Data is Encrypted.

TDE is not a form of access control. All users who have permission to access the database are still allowed access; they do not need to be given permission to use the DEK or a password.

TDE is available only in the Enterprise and Developer editions of SQL Server 2008. Databases used in other editions cannot be encrypted by using TDE and TDE-encrypted databases cannot be used in other editions (the server will error out on attempts to attach or restore).

Database Backups

When TDE is enabled on a database, all backups are encrypted. Thus, special care must be taken to ensure that the certificate that was used to protect the DEK (see How to Enable TDE) is backed up and maintained with the database backup. If this certificate (or certificates) is lost, the data will be unreadable. Back up the certificate along with the database. Each certificate backup should have two files; both of these files should be archived (ideally separately from the database backup file for security). Alternatively, consider using the extensible key management (EKM) feature (see Extensible Key Management) for storage and maintenance of keys used for TDE.

Other Features that Write to Disk

If a feature writes to disk through the buffer pool, data is protected. Features that write directly to files outside of the buffer pool must manually manage encryption and decryption. Thus, older versions of Full-Text Search and even the new Filestream features are not protected by TDE.

Cell-Level Encryption

SQL Server offers encryption at the cell level. Cell-level encryption was introduced in Microsoft SQL Server 2005 and is still fully supported. Cell-level encryption is implemented as a series of built-ins and a key management hierarchy. Using this encryption is a manual process that requires a re-architecture of the application to call the encryption and decryption functions. In addition, the schema must be modified to store the data as **varbinary** and then re-cast back to the appropriate data type when read. The traditional limitations of encryption are inherent in this method as none of the automatic query optimization techniques can be used.

Comparison with TDE

Cell-level encryption has a number of advantages over database-level encryption. It offers a more granular level of encryption. In addition, data is not decrypted until it is used (when a decryption built-in is called) so that even if a page is loaded into memory, sensitive data is not in clear text. Cell-level encryption also allows for explicit key management. Keys can be assigned to users and protected by passwords to prevent automatic decryption. This offers another degree of control (users can, for example, have individual keys for their own data); however, the administrator is further burdened with maintaining the keys (although Extensible Key Management, described later in this paper, can also be used for easier administration). Because cell-level encryption is highly configurable, it may be a good fit for applications that have targeted security requirements.

The primary disadvantages of cell-level encryption are the application changes needed to use it, the performance penalties, and the administration cost. As noted previously, encryption and decryption requires that you use built-ins. This is an entirely manual process and requires the **varbinary** data type; this means columns must be changed from their original data type to **varbinary**. For security, the encryption is always salted so the same data will have a different value after encryption. As a result, referential constraints such as foreign keys, and candidate keys such as primary keys do not provide any benefit on these encrypted columns. This also affects query optimization—indexes on the encrypted columns offer no advantage so range and equality searches turn into full table scans. TDE allows full use of indexes and other traditional query optimization tools as well as performing the encryption in bulk.

As a rough comparison, performance for a very basic query (that selects and decrypts a single encrypted column) when using cell-level encryption tends to be around 20% worse. This inversely scales with workload size resulting in performance degradations that are several magnitudes worse when attempting to encrypt an entire database. One sample application with 10,000 rows was four times worse with one column encrypted, and 20 times worse with nine columns encrypted. Because cell-level encryption is custom to each application, performance degradation will vary depending on application and workload specifics. As noted in Impact on the Database, this compares to 3-5% for TDE on average and 28% in the worst case (assuming the encryption scan is not running).

Although these performance concerns for cell-level encryption can be mitigated by explicit application design, more care must be exercised to prevent the accidental leakage of data. For example, consider a quick scheme to enable fast equality searches by using hashes of the sensitive data. If these hashes are stored in a column along with the encrypted data, it quickly becomes obvious if two rows have identical values because the hashes will be the same. Extra security reviews must be used to ensure that unintended data disclosures do not occur so both the database and application must be security aware. TDE prevents these data leak scenarios by encrypting at the broadest scope. In both cell-level encryption and database-level encryption, information is decrypted on the server; decrypted data is sent to clients in plaintext. SSL is recommended to protect this channel.

Recommended Usage with TDE

Cell-level encryption can be used for defense in depth both for a database encrypted by TDE and for limited access control through the use of passwords. That way, even if either TDE or authorization is subverted, data might still be safe if it is encrypted at the root by a password so that it cannot be as easily accessed. While all the disadvantages of using cell-level encryption apply, using both cell-level encryption and TDE may be useful for a subset of highly sensitive data.

In general, TDE and cell-level encryption accomplish two different objectives. If the amount of data that must be encrypted is very small or if the application can be custom designed to use it (or if the application has custom design requirements) and performance is not a concern, cell-level encryption is recommended over TDE. Otherwise, TDE is recommended for encrypting existing applications or for performance sensitive applications. Additionally, cell-level encryption is available in all SQL Server editions while TDE is available only in SQL Server 2008 Enterprise Edition and SQL Server 2008 Developer Edition.

Extensible Key Management

Extensible Key Management (EKM) is another new feature in SQL Server 2008. It enables parts of the cryptographic key hierarchy to be managed by an external source such as Hardware Security Module (HSM), referred to as a *cryptographic provider*. Encryption and decryption operations using these keys are handled by the cryptographic provider. This allows for flexibility and choice in cryptographic providers as well as common key management. TDE supports asymmetric keys that are provisioned by EKM. No other form of asymmetric key is supported by TDE and database certificates cannot currently be provisioned through EKM. EKM is supported for cell-level encryption through symmetric and asymmetric keys. It is highly recommended that you use EKM with both database- and cell-level encryption for more comprehensive key management and hardware-based cryptography (if available through the HSM).

Windows File Encryption

Depending on which version of Windows is installed, Windows offers two granularities of file protection. In most releases of Windows 2000 and later (including Windows Vista®), the Encrypting File System (EFS) is available. EFS encrypts data at the file level. BitLocker is a new technology that encrypts data at the volume level. It is available in Windows Vista Enterprise Edition, Windows Vista Ultimate Edition, and all editions of Windows Server® 2008.

Encrypting File System

EFS is a file encryption feature introduced in Windows 2000. Like encryption in SQL Server, EFS relies on the Windows Cryptographic API (CAPI). Both files and folders can be marked as encrypted, although the encryption actually occurs only at the file level. Each file is encrypted by an individual File Encryption Key (FEK) much as each database is encrypted with an individual DEK in TDE. The FEK is protected by the user's certificate, similar to how the DEK is protected by a certificate. The EFS certificate is assigned to a user while the TDE certificate is conceptually a server-wide object. Multiple certificates can be used to encrypt the FEK, which allows for more than one user to access a file. When using EFS with SQL Server, the database server service account must have access to the file encryption keys encrypting any database file so that it can read the file. This cannot be used as a form of access control—the service account is used to read database files regardless of the login account.

For more general information on EFS, see How it Works on Microsoft TechNet.

For more technical details on EFS, see the Encrypting File System Technical Reference on TechNet.

Comparison with TDE

As an operating system level encryption feature, EFS has some advantages. Whereas TDE is restricted to database files only, EFS allows for non-database and even folder-level encryption, which allows for broader encryption coverage. Key management is abstracted to the operating system, which enables users to leverage the Windows certificate store. EFS offers a data recovery path if keys are lost, while TDE does not currently have a similar solution.

The disadvantages to using EFS over TDE, are primarily in performance and administration. EFS is not designed for high-concurrency random access (it does not support prefetch or asynchronous I/O). Therefore, I/O operations may become bottlenecked and serialized. While this has a minimal impact in a normal user scenario, it has been a cause for concern in database usage. For more information on using EFS with Microsoft SQL Server, see You may experience decreased performance in some features of SQL Server 2005 when you use EFS to encrypt database files on the Microsoft Help and Support site. EFS requires file administration privileges on the OS level, which the database administrator (DBA) might not have. Because protection is tied to EFS, detaching the database, backing up the database, or adding filegroups may not be protected if these files are in locations not protected by EFS. Also, the implementation of EFS may change from release to release. This normally is not an issue because EFS is primarily used on a single computer for a single user, but it is something to consider. For more information, see this EFS and Vista... and XP blog entry.

Recommended Usage with TDE

EFS is best used on a mobile PC, desktop, or workstation where the database is primarily used by a small set of users. Because of performance concerns, EFS is generally not recommended for use with TDE although nothing prohibits EFS from working with TDE (or with SQL Server in general). Using EFS with TDE is a viable option when performance is not an issue and when defense in depth is desired. EFS can also be used in place of TDE for filegroup level granularity. Also, because it protects at the folder level, EFS can be used to protect some corner cases where data is temporarily written to disk. In environments where performance is a major concern, EFS is not recommended for use with SQL Server.

BitLocker Drive Encryption

BitLocker is a volume encryption feature included in Windows Vista Enterprise Edition, Windows Vista Ultimate Edition, and all editions of Windows Server 2008. By default, BitLocker takes advantage of a Trusted Platform Module (TPM) if one is available, to provide boot integrity protection.

For more information on BitLocker, see BitLocker Drive Encryption on TechNet.

BitLocker and EFS

It is beyond the scope of this article to compare EFS and BitLocker as both are complex technologies that require more technical detail than is covered here. Generally, EFS is targeted at protecting user data while BitLocker is designed to protect volume and system data. In terms of Microsoft SQL Server performance, BitLocker has lower latency on disk reads and writes without the concurrency issues EFS has.

For more information on data protection options at the OS level, see Data Encryption Toolkit for Mobile PCs.

Comparison with TDE

BitLocker and TDE both primarily protect against offline attacks. BitLocker protects at the volume level so when the server is online, the volume is unlocked, though not

decrypted. Like EFS, BitLocker has a data recovery mechanism, which TDE does not yet have. The advantages of using BitLocker are ease of administration, abstraction of key management, and transparency. The disadvantage is that the protection only extends to the volume. Detaching or backing up the database to a different volume that is not protected by EFS or BitLocker causes any protection the file currently has to be lost. The other disadvantage of BitLocker is the broad scope of protection. Because the entire volume is unlocked, any user with access to a computer that can access the files on disk can access the data in plaintext. Similarly to TDE, BitLocker relies on other mechanisms for access control (such as the database permissions used in TDE and the Windows file permissions used by BitLocker). As with EFS, the database administrator might not have the necessary privileges to administrate BitLocker.

Recommended Usage with TDE

BitLocker does not have the same performance concerns associated with EFS so it is recommended that you use BitLocker with TDE for defense in depth. As discussed previously, situations exist where memory can be written to disk, such as hibernation or crash dumps. Additionally, other features may write to the disk outside of TDE. BitLocker can be used to mitigate these scenarios.

Conclusion

Those who are looking for database-level encryption have many options in SQL Server and Windows. These options are not mutually exclusive. The different levels of encryption available in SQL Server and Windows can be leveraged to provide defense in depth and greater overall security. Transparent data encryption provides a good blend of ease of administration, ease of use, performance, and security. TDE also provides a comprehensive defense because the encryption stays with the database even when it is moved to different locations. Both backups and snapshots are protected without requiring support from the server administrator. EFS and BitLocker are also valid solutions either in conjunction with TDE or as standalone encryption systems. TDE is not designed to replace these solutions. Cell-level encryption provides much more granular control including explicit key management although at a cost to performance, ease of use, and administration. BitLocker and EFS provide protection in situations that TDE does not such as crash dump information or hibernation files (if protecting the system volume or system folders). BitLocker and EFS (and to a much more limited degree, cell-level encryption) can be used to protect system databases (**master**, **model**, **resource**, and **msdb**), which cannot currently be encrypted by TDE. EFS is also more generally available as TDE is restricted to SQL Server 2008 Enterprise Edition or SQL Server 2008 Developer Edition and BitLocker is available only with Windows Vista Enterprise, Windows Vista Ultimate, or Windows Server 2008.

For more information:

http://technet.microsoft.com/en-us/sqlserver/default.aspx

http://msdn2.microsoft.com/en-us/sqlserver/default.aspx

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of white papers we release. Send feedback.