Cryptography in SQL Server

SQL Server Technical Article

Writer: John Hicks, Microsoft Corp.

Technical Reviewers: Raul Garcia, Microsoft Corp.

Sung Hsueh, Microsoft Corp.

Il-Sung Lee, Microsoft Corp.

Editor: Diana Steinmetz

Published: July 2008

Applies To: SQL Server 2005 and SQL Server 2008

Summary: Although cryptography provides SQL Server with powerful tools for encryption and verification, these are often not well understood. This can lead to poor or incomplete implementations. This white paper presents an overview of cryptographic functionality and discusses how this applies to authentication, signed procedures, permissions, and encryption. Because the target audience is the database professional and not necessarily security experts, the focus is on practical ways to use cryptography in SQL Server.

Introduction

It often seems that cryptography is employed to solve problems for which it is not really a solution. Even in circumstances appropriate to a cryptographic solution, it is not always the case that it should be deployed by means of the database engine. This paper provides guidance on the use of cryptographic functionality generally, and then how it is implemented in Microsoft® SQL Server®.

When you tell your users that their data is cryptographically protected, you are implicitly assuring them that their data is protected against several kinds of threats. Cryptography is the world of the paranoid (see, for example, Bruce Schneier's seminal book *Applied Cryptography*). If data really must be protected, build your systems while planning on a security audit by the most paranoid person you can imagine. It is rather easy to create an incomplete cryptographic solution. If you plan to keep your data "a little bit secure," it is better to not represent to your users that their data is protected at all.

Just because you need to cryptographically protect your data does not necessarily mean that the database engine is the best place to do it, especially when scalability is a concern. Because cryptography is very CPU intensive, it may be wiser to limit the database to storing data and leave the cryptographic processing to upstream components—like the middle tier—as these are easier to scale.

SQL Server enables you to robustly secure your data—certainly as well as its competitors—and this paper shows you how.

Overview of Cryptography

Behind all cryptographic processing are essentially three functions:

· Symmetric key processing

· Asymmetric key processing

· One-way hashing

Symmetric Key Processing

The best known examples of symmetric keys are those derived from a password. When a document is encrypted with a password, exactly the same password must be supplied to decrypt the document. This is why the functionality is called symmetric. Naturally, you can also use passwords to verify identity. Until the 1970s, symmetric keys were the only tool in the cryptography toolset.

The disadvantage of symmetric keys is precisely their symmetry. If I want another user to be able to decrypt the document that I just encrypted, I must somehow communicate that key to him or her. This presents a problem: if I have a secure way to send the key, that would also be a secure way to send the document; if I do not have a secure channel (which is why I encrypted the document), I also do not have a secure way to communicate the key. (Naturally, there are workarounds, such as communicating the key on another channel with the presumption that anyone with access to one channel does not have access to the second.)

Similarly, symmetric keys are problematic for identity verification because of their symmetry. This is the point of “phishing” e-mails. Typing a password for user verification allows the verifier to know the password, which an unscrupulous host could use to impersonate that user elsewhere.

The most common problem with keys derived from passwords and password phrases is that they are predictable, or even guessable, sequences of characters. They are not really random. The keys derived from passwords are only as strong as the password used to generate it. These keys are much more vulnerable to “dictionary” attacks, in which known words, letter sequences, and combinations are systematically evaluated.

A far more secure key is one that is randomly generated. These are still theoretically vulnerable to a *brute force attack*, which is launched by systematically guessing key values. The larger the key size, the more effort is required to mount such an attack, as there are more values to guess. The common key sizes today are large enough that it is extremely unlikely that the computing power available now or in the foreseeable future would be able to successfully elicit the key value during the useful life of the data (given eternity, of course, this would succeed at some point). Note that for a key to be as strong as its key length indicates, it must be truly random.

In SQL Server 2005 and 2008, the **SymmetricKey** object is derived from a password if the “pass phrase” argument is presented; otherwise, the key is random.

Symmetric keys have two important advantages:

· They can be derived from passwords that can be remembered by people or they can be large, randomly generated values offering maximal security.

· The algorithms tend to be very fast compared to asymmetric algorithms.

These are not insignificant advantages.

Asymmetric Keys

Asymmetric keys always exist as a key pair—the keys may be related as two prime factors of another very large number, through discrete exponentiation, or an elliptic curve in a finite field—and information scrambled by one key can only be unscrambled by the other key. You cannot undo the cryptographic operation with the same key that was used to do the operation; you must use the other key. One of the member keys is defined as the public key and the other is the private key.

Asymmetric keys get around the problem of having to communicate a key to a recipient. For example, if you want to be able to receive an encrypted message from Bob, it does not matter if Jane, Shirley, and Fred are also able to encrypt messages and send them to you, it only matters that *only you* can decrypt the messages. In this scheme, the public key can be issued to anyone and everyone; the private key must not be shared with anyone.

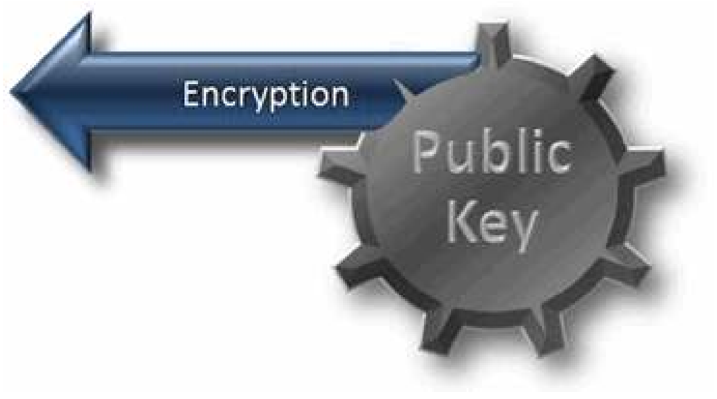

Figure 1: Encrypt with public key

This operation, scrambling a message with the public key, is called *encryption*. Anyone with access to a public key can create an encrypted message, but only the owner of the private key can decrypt and read it.

The inverse operation, scrambling bits with my private key, allows anyone with access to my public key to verify that the information was processed by my private key. This is called a *signature*. Only someone with access to the private key can encode the message such that the public key can decode it. A signature proves that the owner of the private key had access to the corresponding message, and—most importantly—that the message has not changed since then.

Figure 2: Sign with private key

When most people think of cryptography, they tend to think of encryption first. However, the functionality enabled by signatures is arguably more significant because it can handle problems with identity verification and ensure, in a way that is publically verifiable, that the signed content is has not changed subsequent to the signing.

There are three difficulties with asymmetric key cryptography. The first two are the inverse of the advantages of symmetric keys:

· The keys are very large and cannot be remembered by humans (which is also true of random symmetric keys), although they can be embedded onto key cards and such.

· The keys require significant computing resources to use.

· These cryptographic operations are vulnerable to advances in mathematical algorithms or breakthrough computational advances.

Asymmetric keys are built on the fact that some mathematical operations are relatively easy to perform, but difficult to reverse. For example, it is much easier to multiply prime numbers than to factor the result. Asymmetric keys are large enough that the processing power required to calculate the relationship between the numbers (given all the global computing power estimated to be available within the protection timeframe) vastly exceeds the useful life of the data. An advance in mathematical logic or computational power (like quantum computing?) may make calculating the key relationship efficient enough to break this protection.

SQL Server implements asymmetric keys with either **AsymmetricKey** objects or **Certificate** objects. Both use the same RSA asymmetric algorithm and both can be created by SQL Server or imported from a key file, but only certificates can be exported by SQL Server. Certificates may or may not contain private keys (certificates that do not contain a private key cannot be used to create signatures or decrypt data). *In practice, **Certificate** objects are always recommended over **AsymmetricKey** objects.* (The one exception is for Transparent Data encryption with Extensible Key Management hardware. This requires an asymmetric key.)

The SQL Server implementation is discussed in more detail later in this paper.

Cryptographic Hash

A *hash* is a number that is generated by reading the contents of a document or message. Different messages should generate different hash values, but the same message causes the algorithm to generate the same hash value.

A good hashing algorithm has these properties:

· It is especially sensitive to small changes in the document. Minor changes to the document will generate a very different hash result.

· It is impossible to reverse. There will be absolutely no way to determine what changed in a document or to learn anything about the content of a document by examining hash values. For this reason, hashing is often called *one-way hashing*.

· It is very efficient. The hashing algorithm may need to traverse very large files, so the more efficient this algorithm, the better.

Naturally, one can still determine the content of a document from a hash value by correctly guessing the original content and hashing it with the same algorithm; a hash value match indicates a very high probability that the original content was correctly guessed. It is possible for two different messages to generate the same hash, but it is extremely unlikely that two randomly selected messages will do so.

Hybrid Approaches

Because working with asymmetric keys is so computationally expensive (by roughly an order of magnitude over symmetric keys), real-life implementations generally use a several techniques to improve the performance of the operation.

In the case of signatures, you can generate a hash value for the document to be signed. The private key is then used to scramble only the hash value, which is included with the document as the signature. To validate a signature, the recipient need only re-hash the document in the same way and unscramble the included hash value with the public key. If the two hash values match, the signature is valid. The document itself is not scrambled and is readily available for anyone to read.

Likewise, a document can be encrypted with a public key, but doing so requires intensive processing resources. A common solution is to generate a large, random symmetric key and to encrypt the document using that. This takes advantage of the much greater efficiency of the symmetric key algorithms. Next, this key is encrypted by using a public key so that only the intended recipient can access the key. To decrypt the document, the owner of the private key first decrypts the part of the document containing the random encryption key, and then uses that key to decrypt the document itself.

This approach combines the advantages of asymmetric keys while minimizing processing overhead—one of the main disadvantages of public key cryptography. SQL Server 2005 and 2008 support these conventional implementations somewhat straightforwardly, by automatically generating the large, random symmetric key and allowing it to be encrypted by a certificate.

It is also not unusual to implement both a signature and encryption—but it may be less common than people new to cryptography may suppose. By creating a document that is both signed *and* encrypted, the sender can be assured that only recipient will be able to read it, while the recipient can be assured that it really came from the sender.

Certificates and Public Key Infrastructure

Technically, a valid signature only assures me that the owner of the private key once had access to the document that is signed. The important thing to note is that the owner of the private key could be anybody. Technically, I could digitally sign a document committing your company to buy a product from my company. For this reason, a signature is useless unless we can verify the identity of the owner of the private key.

The same identity problem also creates an encryption vulnerability, called the *man-in-the-middle attack*. Suppose you are secretly monitoring a terrorist cell that encrypts communications by using asymmetric keys. You discover that a new terrorist cell wants to send encrypted information to this monitored cell. To do this, the new terrorists request a public key. With a quick DNS hack, you intercept the public key response and provide your own public key instead. When the message from the new terrorist arrives, you intercept it and easily decrypt it using your private key. To avoid detection, you re-encrypt the document using the target cell's real public key and forward the message to the terrorist cell. As long as you are able to reliably intercept the messages, your activity is undetectable. In this scenario, you are the "man in the middle." Note that the man-in-the middle attack must be in place at the beginning of the conversation in order to substitute the public key. If the sender already has the correct public key, the man-in-the-middle attack is impotent.

Certificate Authorities

By having a verifiable association between your public key and your identity, you are protected against both fraudulent signatures and man-in-the-middle attacks. In the nondigital world, this identity problem is (very inadequately) solved by using a Notary Public for signatures. The notary provides assurance that the physical, written signature was created by a specific person, and that person's identity has been verified. The digital equivalent to a notary is a *chain of trust* from one or more *certificate authorities.*

The chain of trust is rather like the human "friend of a friend" certification. If someone name Joe presents himself to me and I do not know Joe, Joe can report that his friend Betty will unequivocally vouch for him and his identity. However, I do not know Betty either. Betty, however, is able to enlist Ted to vouch for her veracity. I do know Ted and I trust him. When Ted confirms that Betty is reliable, I can use Betty's certification to validate Joe.

The highest certificate authority is called a *root authority*. These are the organizations that provide public certification of an identity and public key. The root authority does this by signing your organization's public key and identity information with their own private key. This certification is valid for a specified period of time. These authorities also manage a public revocation list, in the event that something was wrongly certified or a private key was compromised. Anyone who trusts the public key of the root authority will automatically accept their signed certificates as correctly validating the identity behind a public key.

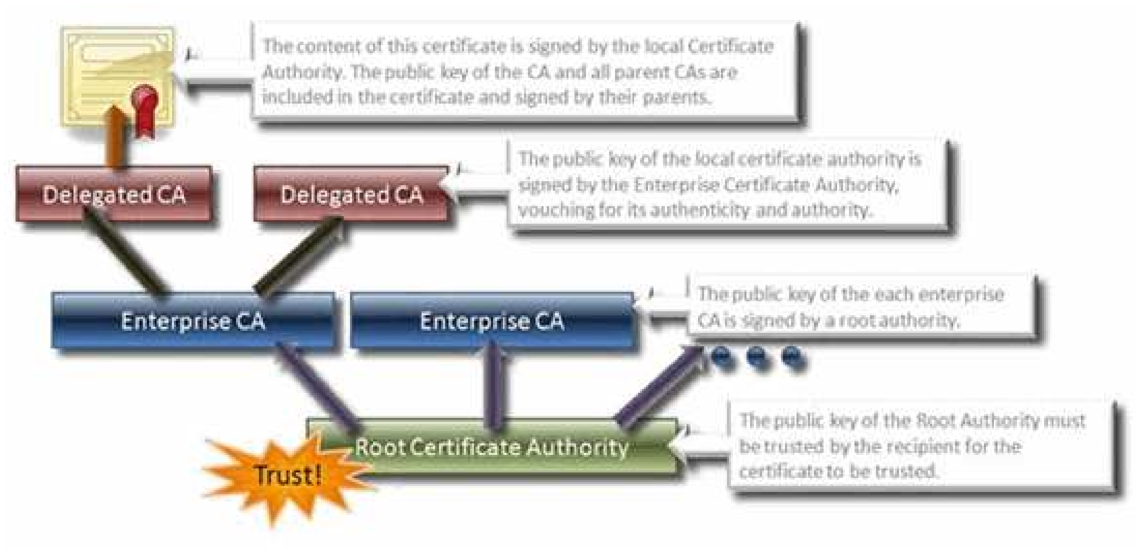
The content of this certificate is signed by the local Certificate Authority. The public key of the CA and all parent CAs are included in the certificate and signed by their parents.
Delegated CA
Delegated CA
The public key of the local certificate authority is signed by the Enterprise Certificate Authority, vouching for its authenticity and authority.
Enterprise CA
Enterprise CA
The public key of the each enterprise CA is signed by a root authority.
Root Certificate Authority
Trust!
The public key of the Root Authority must be trusted by the recipient for the certificate to be trusted.

Figure 3: Chain of trust

What prevents you from setting up your own root authority and certifying your own fraudulent signatures? Well, nothing. Except that, by default, no computer will trust your certifications. To see which root authorities your computer trusts, run certmgr.msc (type it on the command line or use the Run dialog box). To learn more, see Creating, Viewing, and Managing Certificates [http://msdn.microsoft.com/en-us/library/aa379872.aspx] on the Microsoft Developer Network (MSDN).

If your organization allows individuals to sign documents or send encrypted messages, you need a large number of certificates. Each person who is eligible to sign or receive encrypted messages should have his or her own distinct key pair (actually, a distinct key pair for each set of functionality). As it is not realistic to have each person certified by a root authority, the organization implements its own certificate authority. Generally, this local certificate authority is authenticated by a root authority; in other cases, the local certificate authority is trusted as a root authority through a group policy. In a large enterprise, this enterprise certificate authority may even authenticate other certificate authorities. The certificates issued by a certificate authority include the signatures of all parents up to the root authority. This chain of trust is valid as long as all signatures are valid, the ancestry is complete, and the first-level certifying authority is a root authority.

A valid certificate file that includes a chain of trust that ultimately resolves to a trusted authority will be implicitly trusted. Since the certificate file contains all of the public keys in the chain of trust, it is not necessary to contact all of the certificate authorities to verify the chain (unless checking for revocation). Certificate files also have purpose designations (such as e-mail encryption, authentication, and trust), expiration dates, and other properties.

Interestingly, a Windows Certificate Authority cannot be configured to archive the private key of a signature certificate. If a user's hard disk crashes, he or she may be able to continue to read encrypted messages by connecting to a certificate authority and retrieving previously issued certificates. Crucially, an authorized administrator may also be able to retrieve these archived private keys if it becomes necessary to audit a user's correspondence. However, the fact that the certificate authority does not contain a private key provides "non-repudiation" for signatures. It is simply not possible for the same administrator to retrieve a signature certificate from the certificate authority and sign messages as though they originated from another user. Only the owner of the private key can do this. If the original signature certificate is lost, it cannot be replaced by the certificate authority; but the certificate authority can issue a new signature certificate, even while the old signatures remain valid. In short, a single user can be associated with multiple signatures.

Revocation

If a certificate's private key becomes compromised, the issuing authority can add it to its Certificate Revocation List (CRL). Ideally, a signature should be checked against the CRL of the issuing authority before it is assumed to be valid. Note that certificates and CRLs are both time sensitive, so it may be critical to compare the message time with the validity dates in the certificate or CRL.

Note that SQL Server does not validate dates, check CRLs, or participate in a Public Key Infrastructure (PKI). SQL Server relies on the administrator to ensure that that any certificates loaded into the database are trustworthy. Remember that for the most part, PKI is intended to solve problems (such as validating newly presented certificates) that are ancillary to the SQL Server data storage role.

A Quick Tour of Cryptography in Windows

Although this is not a comprehensive presentation on cryptography, a brief overview of cryptographic functionality in Microsoft Windows® may provide a helpful orientation. SQL Server cryptographic functionality is ultimately rendered by the cryptographic algorithms within Windows.

The cryptographic algorithms in Windows are provided by the cryptographic APIs—also called the CryptoAPI, or just CAPI. Much of this functionality was eventually exposed to the object-oriented world in a thin COM wrapper called CapiCom. CapiCom is not recommended for server applications, and frankly I find it generally just as easy to invoke the APIs directly and avoid the dependency on an external type library. The .NET Framework 1.*x* releases had relatively poor support for the CryptoAPI, and some of the implementations contained bugs. If you are going to create an application that uses Windows cryptographic functionality, use the .NET Framework 2.0 or later. It is far more complete and robust. Microsoft Visual Studio® 2008 contains managed wrappers for the new functionality offered by Windows Vista® and Windows Server® 2008. The newer algorithms are more efficient and more secure, but they are not available within SQL Server native cryptographic functionality.

One of the biggest difficulties of the Windows cryptography documentation is that Windows cryptographic architecture is designed to be extensible. This is a great thing, but it can also add considerable confusion. For example, certificates can be stored in one of several defined certificate stores, but they can also be stored in alternate stores, such as key cards and hardware devices. (Extensible Key Management (EKM), new in SQL Server 2008, also offers this functionality.) The cryptographic algorithms, features, and functionality of the CryptoAPI also depend on which cryptographic provider is in use. This means that the core cryptographic behavior is very difficult to document because it varies depending on the provider. You can even use third-party providers.

This is why the documentation may seem to stop short of actually describing how to implement the functionality you need. Over time, the documentation has improved.

Furthermore, different versions of Windows have access to different providers and use different default providers. The most recent releases generally have more secure providers, but using them may mean that certificates, signatures, or encryption that is generated by newer versions of Windows cannot be decrypted in older versions of Windows. This is why the cryptographic algorithms available in SQL Server depend somewhat upon which Windows version it runs on.

Cryptography in SQL Server Authentication

As the main point of this discussion is how to use cryptography to protect data stored in SQL Server, this section may be somewhat of a detour. You may find this information useful, however, and if you really need to cryptographically protect your data, you also need robust authentication.

Security in SQL Server has two parts: *authentication* and *authorization* (permissions). Authentication consists of verifying the identity of a user, or the *security principal*. Both encryption and signatures can play a role in this. Authorizations or permissions are the rights granted or denied to the user. Signatures can also play a role in permissions, as you can assign permissions to a signature in SQL Server.

Windows Authentication

Ideally, an SQL Server instance should validate user identities by using Windows Authentication exclusively. Windows Authentication works with the Active Directory (Kerberos), or without it (NTLM). On domains with an Active Directory, Integrated authentication is:

· Easier to manage—there is no need to create individual logins or manage passwords. Logins can be based on Windows group membership.

· More flexible—users can be validated by passwords, smart cards, biometric devices, and so on.

· More secure—Kerberos provides immunity to certain attacks that require additional work to avoid with SQL Server authentication.

However, when a user connects to SQL Server by using Windows Authentication on a domain with an Active Directory, the SQL Server instance must connect to a domain controller to retrieve the corresponding client *security token*. A security token contains the user identity, group membership, and Windows privileges. This enables SQL Server permissions to be granted to Windows groups. The identity of the token retrieved from the domain controller is compared to the connection information submitted by the user; if they match, the identity is valid. By validating the user’s credentials with the domain controller, Windows Authentication is not vulnerable to attacks such as man-in-the-middle.

Because SQL Server requires access to a domain controller to retrieve these tokens from the Active Directory (assuming that this is not an NTLM domain), an intermittent or high-latency connection to the domain controller may cause connections to fail. To ensure fast user connectivity, *ensure that SQL Server connectivity to the domain controller is as fast as possible.* (Of course, you must also ensure that the SQL Server service account has permission to query the domain).

Integrated authentication defaults to Kerberos but fails back to an NTLM handshake if necessary. When a Windows user connects to SQL Server, authentication is first performed by Windows itself, prior to receipt by the SQL Server instance. This means that if Windows Server refuses a user’s connection, the user cannot establish a connection to SQL Server no matter what permission he or she has.

SQL Server Authentication

SQL Server Authentication is a good choice in several scenarios such as when SQL Server administrators have no ability to manage Windows groups, users are establishing connections from a non-Windows platform, or when users need to connect from an untrusted domain. For example, it is not uncommon for the production server domain to not trust the desktop user domain where the database administrators (DBAs) work—but this is not necessarily a best practice. Another advantage of using SQL Server Authentication is that when connecting to linked servers, it avoids the delegation requirement for double hops.

In SQL Server 2000, the login handshake was not encrypted unless the administrator explicitly provided an SSL certificate; otherwise, the password and login name were passed as plain text. In the case of SQL Server 2005 and 2008, however, if no certificate is specified, SQL Server generates and uses a self-signed certificate. Self-signed certificates are vulnerable to man-in-the-middle attacks, but this is still far better than plain text logins. Ideally, SQL Server 2000, 2005, or 2008 should have a valid certificate signed by a certificate authority for the SSL encryption. The process for installing this is the same for both versions and is described in How SQL Server uses a certificate when the Force Protocol Encryption option is turned on [http://support.microsoft.com /default.aspx?scid=kb;en-us;318605] .

At the time of initial connection, SQL Server does not know whether Windows Authentication or SQL Server Authentication will be used, so SQL Server 2005 and 2008 always use an SSL encrypted authentication. This is not necessary with Windows Authentication, but the SSL channel is used anyway. Not all clients support authentication using SSL encryption and in these cases the user name and password *will be sent as plain text!* SQL Server can be configured to allow only clients that support login encryption via the **ForceEncryption** property under Network Configuration in Configuration Manager.

The password complexity policy is the same as the password complexity on the host server (provided the host server is running Windows 2003 or later). In fact, the complexity is actually validated by a Windows API that is called by SQL Server. It is highly recommended that you leave this on, but you can turn it off by using the CHECK_POLICY option of the CREATE LOGIN statement.

Real-World Problem

In environments where the server is on a different (untrusted) domain from the world of desktop users and, consequently, SQL Server Authentication is used to allow user access, there is still sometimes a need for users throughout the enterprise to access the data.

A common approach is to give all the users the same SQL Server login; this essentially becomes a public login. For those of you not paying attention: this is a bad idea. This public login can be restricted to appropriate permissions, but as anyone with experience knows, some users misbehave and we must manage who is doing what without simultaneously affecting all users.

The correct alternative is to create a SQL Server login for each user who needs access. This allows auditing, the ability to revoke access, the ability to grant additional permissions, and the ability to determine who is using your data. The downside to this approach is that DBAs in this situation may often feel that he or she spends too much time creating new logins.

One solution is to have a Web page where users go to request access. Ideally, this page is hosted on the desktop domain and you can use ASP.NET to determine users' domain names, which become their SQL Server login as well. In this case, the user need only provide a password for baseline-level access. The application can then connect to the database server and create the account.

The core security problem to this approach is that the permissions necessary to create a new login vastly exceed the permissions that should ever be granted to a database login belonging to a Web application.

A workaround, which I implemented in SQL Server 2000, is to create a table that contains login requests, The Web application has permission only to execute a stored procedure that writes to that table (that is, minimal permissions). An independent SQL Server Agent job is also set up to periodically poll the table and it creates logins and database permissions for any requests that it finds. Naturally, the SQL Server Agent job requires elevated permissions to create logins and grant permissions, but there is no danger of these permissions being exposed to the Web application login. In this way, the Web application does not require any advanced database permissions, but the general login creation (or password change) process can be completely self-service.

A better solution to the problem of allowing elevated permissions to execute a specific task (like creating a login) to a user account with minimal permissions is to use *signed stored procedures* and give the signature the elevated permissions. This is discussed in the next section.

Using Signatures on Procedures to Grant Permissions

A new—and, in my experience, little known—feature in SQL Server 2005 is the use of a *signing certificate* to sign stored procedures, assemblies, views, and functions. Administrators can assign permissions to the signing certificate itself and can then be sure that the permissions granted cannot be inadvertently modified by changing the stored procedure, assembly, and so on. Signed executable code cannot be changed without breaking the signature, which consequently invalidates any permissions granted to the signature—that is, unless the changed procedure, assembly, view, or function is re-signed with that signature.

The syntax for adding a signature is simple and includes the ability to add an existing signature (presumably created on another server) with a certificate containing only a public key and to specify a password if the certificate is password protected:

```
ADD SIGNATURE TO <module_name> BY CERTIFICATE <key_name>

[{ WITH PASSWORD = ‘password’ | WITH SIGNATURE=binary_signature}]
```

The need for signing code may seem rather exotic. In most production environments, administrators do not need to worry about someone other than administrators changing procedures. However, there are important scenarios where this could be very useful:

· A signature enables a low-privileged account to execute a specific, pre-approved task that the account would not otherwise have permissions to perform.

· ISVs may find signing certificates useful for ensuring that the stored procedures, assemblies, and so on, that are released with their product are not changed.

· Signatures solve a difficult problem with cross-database authentication.

The ability to deploy signed stored procedures, assemblies, functions, and views can allow an ISV to effectively prevent code tampering by a customer's database administrator. This can help prevent local changes that would otherwise become support issues. Signed procedures are easy to include in any deployment scenario: the signatures can be backed up and restored, attached, or scripted as a blob with the ADD SIGNATURE Transact-SQL statement. Technically, a system administrator may be able to temporarily work around this by implementing his or her own certificate of the same name and signing all of the same procedures, but this would not be trivial. It would also be detectable by the ISV.

The real power of signatures lies in cases where data for a particular query is spread among multiple databases on the server instance and there are different security models (or database owners) among the databases.

One way to address this is to ensure that the server logins are mapped as users in both databases and that the users have valid permissions in both databases. Every user granted permission to execute a query in the first database must also be granted appropriate permissions in the second database. Needless to say, this can quickly become unsupportable, especially when the databases are owned by different groups.

Another solution is to enable *cross-database ownership chaining* on the server. Although this may be a convenient solution, it is a huge security risk and for that reason was long ago disabled by default. A related alternative is to use the EXECUTE AS impersonation statement, which allows the stored procedure in the first database to impersonate a specific valid user in the second database. In order for the impersonating user to successfully query the target database, however, the TRUSTWORTHY property must be set for the source database. The TRUSTWORTHTY property allows impersonated users to query the target database, but it also opens a security hole similar to cross-database ownership chaining. Both settings are generally bad ideas unless you have a *very* clear understanding of how the database permissions will work between the databases.

To illustrate, suppose I want to maliciously take control of your database, but I have no permissions in your database. I have my own database residing on the same server instance. The first thing I do is to add one or more logon accounts to my database for users who have elevated permissions in *your* database. I cannot log in as one of these high-privileged users because I do not know the password, but as the owner of my own database, I can grant myself permission to impersonate this user by using the EXECUTE AS statement. Next, I concoct a scenario with which I convince a server administrator to mark my database as Trustworthy. Now my impersonated login context extends to your database, giving me all the power of this high-privileged user in your database. So, flagging a database as Trustworthy really means that the owners of the database must be trusted —which is essentially only the case if you are the owner of both databases.

For more information about the TRUSTWORTHY bit, see this SQL Server Security blog entry [http://blogs.msdn.com/sqlsecurity/archive/2007/12/03/the-trustworhy-bit-database-property-in-sql-server-2005.aspx] .

Signing certificates solve this problem even when the foreign database is not Trustworthy. For example, if a user wants to query my database as a part of their stored procedure, I simply:

1. Verify that the stored procedure is acceptable.

2. Create a signature for it by using a signing certificate.

3. Grant the appropriate permissions to the signing certificate.

4. If the other database owner does not already have it, I provide a copy of my certificate (containing the public key only).

The certificate and signature are added to the database that I do not trust; because I trust the signature, the procedure can execute against my database. If the owner of the other database tries to modify the procedure to do something other than what I have signed, the signature becomes invalid.

To grant permissions to a certificate, you must create a user (database scoped) or login (server scoped) mapped to it:

```
USE myDB

CREATE CERTIFICATE <certname> FROM FILE = 'cert_name.cer'

CREATE USER <username> FOR CERTIFICATE <certname>
```

or

```
USE master

CREATE CERTIFICATE <certname> FROM FILE = 'cert_name.cer'

CREATE LOGIN <loginname> FOR CERTIFICATE <certname>
```

Generally, you will reduce confusion if the user name or login name is the same as the certificate name. For example:

```
CREATE CERTIFICATE SupervisorCert FROM FILE = 'SupervisorCert.cer'

CREATE USER SupervisorCert FOR CERTIFICATE SupervisorCert

GRANT SELECT, INSERT ON dbo.Employees TO SupervisorCert
```

Technical Details

There are two ways in which a signature can allow a procedure, assembly, function, or view to execute in a database:

· **As a secondary identity.** Earlier in this discussion we talked about security tokens, which represent a user or login's primary identity and secondary identities, which are obtained through membership in SQL Server roles and Windows groups. Users are typically granted permissions based on their secondary identities (because it is generally easier to manage groups); for example, I may have permission to run a stored procedure because of my membership in the Supervisors group. When code is protected by a signature, the mapped signature user or login account becomes yet another secondary identity during the execution of the signed procedure. Permissions can then be granted to the signing certificate user account, just as they would be to a role or a Windows group. Any permission checks evaluate the primary identity and all secondary identities, including the signing certificate. If permission is granted to any of the identities (user account, SQL Server roles, Windows groups, or certificate user accounts), the check succeeds. Calling from a signed procedure to an unsigned procedure causes the secondary identity to be removed during the execution of the unsigned code. (One caution is that the creation of new objects in a cross-database scenarios may cause undesired implicit user creation in the target database. If new database objects are to be created, use the next option.)

· **As an identity validator.** Alternatively, signatures can be used to validate an impersonation attempt. When an EXECUTE AS impersonation statement is used within a signed procedure, assembly, function, or view, the impersonation is trusted for all databases in which the signing certificate user account has the AUTHENTICATE permission. If the signing certificate login account (server scope) has the AUTHENTICATE SERVER permission, signed impersonation statements are trusted on every database within the instance.

Signing certificates, then, enables a very controlled escalation of permissions between databases. It is possible to use the certificate both as a secondary identity and to authenticate EXECUTE AS impersonation. Explicitly granting permissions to the corresponding certificate user allows any user to inherit this permission when running signed code. Granting AUTHENTICATE or AUTHENTICATE SERVER permission to the corresponding certificate user account allows the certificate to impersonate any user specified in a signed EXECUTE AS statement.

In either case, the administrator is assured that the activity of the procedure, assembly, and so on, cannot be changed to take malicious advantage of the elevated permissions. Assuming that the certificate contains a private key, only users with control permission on the signing certificate can create a signature.

One important caveat is that signatures can be copied like other database objects. A signed procedure, assembly, function, or view whose certificate is granted elevated permissions may be copied to my database and, if I can also load the public key certificate, I should be able to execute the signed code with the same permissions as the original database.

Furthermore, it should be obvious that it is never advisable to sign a procedure containing dynamic SQL. However, if dynamic SQL is required, you can put the dynamic SQL into its own, unsigned procedure, then call it from the signed procedure. When this new procedure is called by the signed procedure, the extended security identity provided by the certificate is removed during the execution of the unsigned procedure and resumed when execution returns to the signed procedure. If you do not trust the code, do not sign it.

SQL Server and the Key Hierarchy

SQL Server 2005 introduced several new security objects. Most of these are stored within specific databases (the databases in which they are to be used), in a sort-of security hierarchy. When the security hierarchy is used, access to a parent key can be used to decrypt a child key. While the hierarchy enables convenient access to keys, each key is only as secure as its parent. The hierarchy can be circumvented at any level and it need not be used at all.

Certificates (and asymmetric keys) and symmetric keys are *securable objects*, which means that the SQL Server permissions architecture allows the administrator to grant or deny permissions on them, just as for a table or stored procedure. Consequently, access to these objects is protected in two ways: through the standard permissions architecture and through the need to access the key that encrypted them.

Because the SQL Server key hierarchy is flexible, it can be somewhat challenging to understand how it works. The first important decision is whether you want to continue to make use of the key hierarchy. It offers convenience at the cost of some

security.

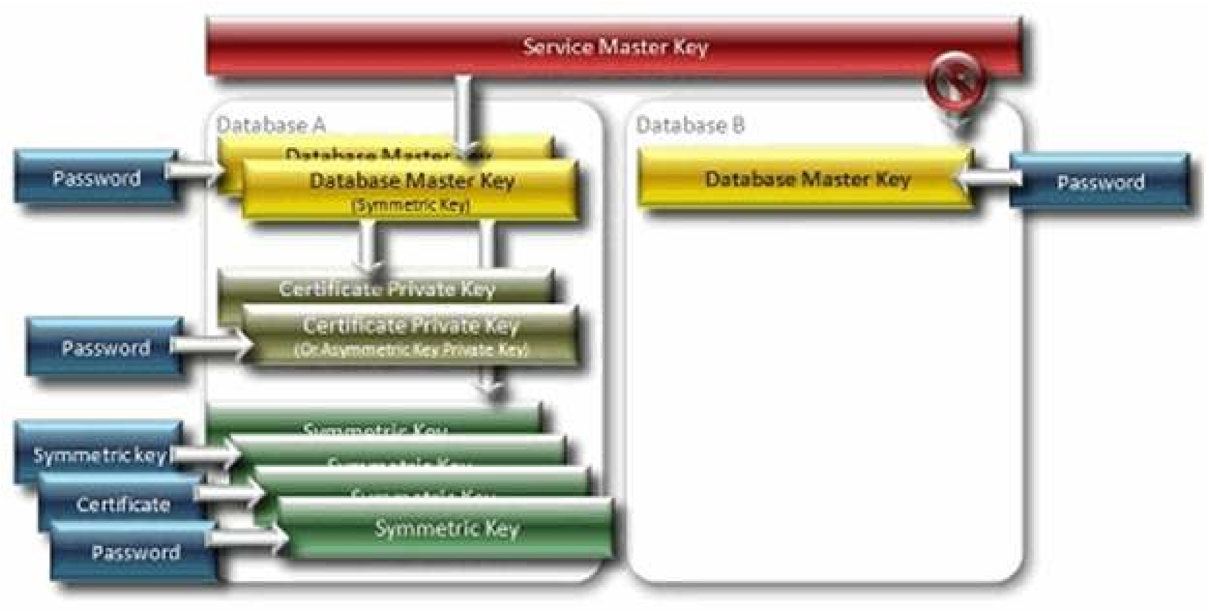

Figure 3 : Key Hierarchy

Service Master Key

The top level of the key hierarchy is the *Service Master Key*. This is a server-wide key that is used to protect the Database Master Keys (and other sensitive server data, such as linked server logins). It is automatically created and it cannot be dropped; it can, however, be ALTERed.

Since the Service Master Key is not stored in one of the user databases, it is important that this key be backed up and stored in a safe and secure place. There is a distinct set of backup commands for this.

The Service Master Key is protected by the DPAPI and saved both at the machine and user (service account) level. Consequently, it is possible to change the service accounts on the same computer without losing the Service Master Key, and it is also possible to change servers (as long as the same domain account is used on the new server) without losing the Service Master Key. If the computer *and* service account changes at the same time, the Service Master Key will be lost and must be restored from the Service Master Key backup.

Database Master Key

The *Database Master Keys* exist at the database level and are stored in the database and are backed up with the database. They must be explicitly created and they can be dropped if they are not needed. These keys are used to protect database-level objects, such as private keys and symmetric keys.

By default, two instances of the Database Master Key are created; one is encrypted by the Service Master Key and another is encrypted by the password supplied when the key is created. Consequently, access to the Database Master Key is either through access to the Service Master Key or through the password. (Note that this should not be confused with a single instance of the Database Master Key that is protected by both the Service Master Key and a password, which would require access to both to decrypt the key. Rather, the Database Master Key is encrypted twice, once by the password and independently by the Service Master Key.) This means than anyone with access to either the Service Master Key, such as sysadmins, or the password also has access to the Database Master Key.

You can create additional instances of the Database Master Key, each protected by another password. This allows *key escrow* scenarios, where each DBA has access to the Database Master Key using their own distinct password. When a DBA leaves the team, their instance of the Database Master Key can be removed without having an impact on the others.

To try to protect the database keys from sysadmins, you can remove the instance encrypted by the Service Master Key (by using the ALTER MASTER KEY statement). In this case, an OPEN MASTER KEY statement and the corresponding password will always be required to open and use the key. However, this offers only very limited protected against a determined sysadmin. If this is a problem, you should host your data elsewhere.

Certificates and Asymmetric Keys

You can create a Certificate object (or asymmetric key) by loading a certificate file that is created externally—such as through a certificate authority—or SQL Server can create the certificate. Naturally, when SQL Server creates the certificate it is self-signed. There is little difference between a SQL Server certificate and a SQL Server asymmetric key; the primary difference is that the asymmetric key cannot be exported. For this reason, they are treated as the same thing in this discussion. Create **Certificate** objects unless you have a reason to prefer an asymmetric key.

Certificate files with private keys commonly require passwords to open them. This password must be provided separately (as the DECRYPTION BY PASSWORD argument) and is distinct from the password that might be used to encrypt the private key on the server.

When you create a **Certificate** object (or asymmetric key) you must decide whether to protect the private key with a password or with the Database Master Key. This choice determines how the key must be opened for use. Only one encryption can be in use at a time—the Database Master Key or a password. This setting can be changed by using the ALTER CERTIFICATE statement.

Because the certificate public/private key pair is designed to solve the problem of communicating a key, you would expect that data encrypted with an SQL Server certificate could be decrypted by using the same certificate key pair on an external box (or vice versa). While technically possible, this is not supported.

Symmetric Keys

The **SymmetricKey** object in SQL Server can be derived from a password or password phrase, or from a random set of bytes. If you specify a password as the KEY_SOURCE argument when you create the symmetric key, the resulting symmetric keys are based upon that password seed; otherwise, the resulting symmetric key is based on a random set of values.

It is generally best to *not* specify the KEY_SOURCE argument, as symmetric keys derived from a password are almost certainly much less secure than those generated randomly. The primary reason for using a KEY_SOURCE is when a key must be created or recreated in multiple places.

Confusingly, the symmetric key can also be secured with a password. This password is not the same thing as the password seed (the KEY_SOURCE argument). When secured with a password, the key itself is random, but access to it requires a password. The key is still only as secure as this password, but this protecting password can change without changing the key—and without the subsequent need to re-encrypt the data.

Unlike certificates, SQL Server implicitly stores and maintains the multiple versions of the same symmetric key, each with their own encryption. This means that that the same symmetric key can be protected by more than one password, other symmetric keys, multiple certificates, and/or the Database Master Key.

Since symmetric keys are generally used in encryption, the fact that we can store the same key multiple times means that multiple users can access the same key without also sharing their certificate or password. For example, I can encrypt data with "key X" and provide a copy of "key X" to users Joe, Sue, and Bob, each with their own distinct password (or certificate). If Bob leaves the company, I can drop his copy of the key without having an impact on Joe and Sue. Symmetric keys must be opened before use with the OPEN SYMMETRIC KEY statement. The decryption process automatically determines the correct key from the set of open keys.

In summary, the default architecture—the key hierarchy—in SQL Server allows relatively convenient access to any keys to which the user has permission. Generally, this is a good solution, as the permissions in SQL Server are robust and should provide adequate security. The primary reason to abandon this approach is when those who have implicit permissions on the keys (such as sysadmins, or those who can grant themselves permissions on the keys, such as security administrators) should not have use of the keys.

A reasonable argument can be made that removing the key hierarchy will also improve security by removing inherited permissions and instead requiring a password or other key. However, in practice the password must be presented in some way and ensuring that a skillful administrator could not capture it would be very difficult. If this is a concern, it is better to encrypt and decrypt the data externally.

Protecting SQL Server Data by Using Encryption

The first question to ask is "Why do you want to encrypt data on the server?" In many cases, controlling access to a table by using SQL Server robust permissions architecture is protection enough. As a general rule, if the storage media (SAN, disks, backup tapes, and so on) are completely secured, it is technically not necessary to encrypt the data. Encryption offers little help in controlling access to data through the database engine; if an attacker is able to defeat your permissions architecture, he or she probably can also access your encryption keys with only a little more trouble.

However, systems are not perfect and if an attacker were to gain access to the file system, encryption does offer "defense in depth."

The case for encryption is strongest in the following scenarios:

· When the storage media is vulnerable, as when a portable computer can be lost or a backup tape can be compromised.

· When the data must be protected from sysadmins. Practically, in this scenario it would be best not to use

SQL Server internal encryption and far better to rely exclusively on the middle tier to perform encryption and decryption. It would be very difficult to protect your data from a determined sysadmin if your keys are stored in SQL Server.

· When there is a need to protect sets of records (rows) within the same table where users have rights to query the table directly. In this case, users with access to specific cryptographic keys can see the corresponding records, but not have access to the other records in the same table. Without encryption, this could be much harder to enforce. (When you merely need to limit the columns that are accessible to the user, this is better done with permissions on views.)

· When regulations require that data be stored in encrypted form.

When one or more of these conditions do not apply, encryption is probably not necessary.

When access to disk storage is storage is secure, but offsite tape storage may be less secure, you may want to consider third-party backup solutions that are capable of encrypting the backup files.

Database-Level or Volume-Level Encryption

With SQL Server 2005, Microsoft introduced encryption through the database engine. This is still the most flexible, granular, and robust encryption option. The biggest disadvantage is that the database must be essentially built with this encryption architecture in mind. This means that column data types must probably be changed (to **varbinary**) and data access procedures must explicitly decrypt the data with a valid key. There may also be a substantial performance impact when searching encrypted columns.

Windows Server 2008, some versions of Windows Vista, and SQL Server 2008 Enterprise Edition and Developer Edition provide some new options. Windows Server 2008 and Windows Vista add BitLocker (and the Encrypting File System, which is not recommended for SQL Server). SQL Server 2008 Enterprise Edition has Transparent Data Encryption (TDE).

BitLocker and TDE are remarkably similar. Both work as layers between the SQL Server buffer pool and the underlying storage, decrypting and encrypting as data is read and written to disk. The primary advantage to either approach is encryption can be added to the disk storage media without any changes to the database. In both cases, the encryption is transparent.

As a general rule, it probably makes more sense to use BitLocker on portable computers (even those without SQL Server) as it protects the entire volume. If laptop data is compromised, presumably this happens due to physical access to the media and BitLocker protects against this. On a server, however, it probably makes more sense to use SQL Server TDE, as this encrypts the database file, log file, **tempdb**, and the backup files. If a server storage volume is compromised, it is typically through network access and BitLocker does not protect against this. With BitLocker, backup files are not encrypted unless they are stored on an encrypted volume. With TDE the backup files are also encrypted, but the new WITH COMPRESSION backup option does not provide useful compression—which is always the case when you try to compress encrypted information. (The new SQL Server 2008 Page Compression feature will still be effective.)

Note that neither BitLocker nor TDE encrypt data in memory. This can provide a substantial performance benefit over the encryption offered in SQL Server 2005, including the use of indexed searches (discussed later). But this also means that a system administrator with access to this memory can read the unencrypted data. All users with database permissions to access data will see unencrypted data. Because TDE is limited to database specific files, unencrypted data might leak to disk if the engine is forced to page data from memory. Also, FILESTREAM objects are not encrypted by TDE, as they might be with BitLocker.

It is possible to combine any or all of these techniques to achieve defense in depth. For more information about these encryption tradeoffs, see Database Encryption in SQL Server 2008 Enterprise Edition [http://msdn.microsoft.com/en-us/library/cc278098(SQL.100).aspx] by Sung Hsueh.

Cell-Level Encryption

To some extent, BitLocker and TDE offer data encryption with minimal performance and administrative overhead. This may be enough. As discussed earlier however, anyone with read access to the tables will see plain text (unencrypted) data. It is also possible for these types of encryption to "leak" unencrypted information under certain scenarios such as memory dumps or paging in the case of TDE, and remote volume access and unencrypted backups in the case of BitLocker. A truly robust data security strategy offers defense in depth.

Cell-level encryption remains encrypted in memory and users with read access to the tables see encrypted information unless they have access to the key and explicitly decrypt the data. Unlike TDE, this functionality is also available in SQL Server 2005 in all editions.

To review what we discussed earlier: for performance reasons you should generally not use a public/private key pair (asymmetric keys or certificates) for data encryption and decryption. The performance of asymmetric key algorithms is very poor. The real advantage of using a key pair is that anyone can encrypt, but only the owner of the private key can decrypt. In other words, asymmetric keys and certificates solve a communication problem between a sender and receiver—a problem that probably does not apply to the database engine. In the case of a database system like SQL Server, it would almost certainly be the same agent doing both the encryption and the decryption (the engine or an agent service account). This makes the use of asymmetric keys pointless, and the result of this is unnecessarily poor performance.

In the rare case where either encryption or decryption is not handled locally or must be handled by different agents, it might make sense to use certificates. In that case, best practice dictates that you encrypt the data with a large, random symmetric key, and then use the certificate to encrypt only this encryption key. (The .NET cryptographic symmetric key classes have the ability to generate a random symmetric key.) The decryption agent can use its private key to decrypt only the symmetric key, and then use the much more efficient symmetric key to decrypt the rest of the message.

Data encrypted by SQL Server must be stored in a **varbinary** column and consequently the encrypted value cannot exceed 8,000 bytes. Note that because encrypted values tend to be larger than the source plain text, the maximum size of encryptable plain text data is slightly smaller than 8,000 bytes. The impact of encryption on storage size varies depending on the algorithm used. After you have some experience with a particular algorithm, you should be able to predict this reasonably accurately. If you need to know this exactly, run some tests. The SQL Server 2005 Encryption: Encryption and data length limitations [http://blogs.msdn.com/yukondoit/archive/2005/11/24/496521.aspx] blog entry is a good place to start.

It may be possible to use the length of the encrypted data to infer other information about the source data. For example, a disreputable insurance company might be interested in the fact that a potential subscriber's encrypted medical history has thousands of bytes of data, even if the content is inaccessible. However, because encryption algorithms generally have fixed block sizes, it is generally not feasible to evaluate the encrypted value's length for clues to the plain text value's length (only large text fields are generally vulnerable to this).

Maliciously Using Encrypted Data without Breaking the Encryption

Besides what can be inferred from the general size of the encrypted data, *naively* encrypted data is vulnerable to several attacks that do not require access to the decryption keys.

· Suppose a malicious user has access to personnel records. Knowing that the CEO's salary is considerably higher than his or her own salary even though not knowing the precise amount, the user could simply update the content of their own record with the encrypted value of the CEO's salary.

· A malicious user could also update his or her personal record with specific information and capture the encrypted result. Subsequently, the user could search the table for other records containing the same encrypted result; any matching encrypted values must have contained the same plain text value. In this way, the user can determine the value of an encrypted field simply by confirming an earlier guess.

Fortunately, SQL Server internal implementation protects against both these attacks. Database-level or volume-level encryption is not affected by these problems, as any user with read access to the tables will not see encrypted data.

To prevent discovery of plain text content by comparing encrypted values (the second attack), most encryption algorithms include a *salt value*. Specifying a different salt value generates a very different encrypted output. When using the .NET cryptography classes, you can specify the salt as the *initialization vector* argument. In SQL Server, a random salt value is always applied to the encryption.

Because a salt value is implicitly generated and included with the encrypted output, you cannot compare values encrypted with SQL Server internal encryption functionality to see if they are equal. However, this is possible with data encrypted by the .NET classes as long as the same initialization vector is used. It is not possible to specify the salt value in SQL Server.

To protect against the other attack, SQL Server encryption functions accept an *authenticator,* also known as a *data integrity parameter*. Ideally, this should correspond to a primary key for the record or some other field that is unique for any given row. Upon decryption, the same authenticator value must be provided; if a mismatch occurs, the encryption is aborted and an error is raised. This architecture makes is impossible to execute a "copy and paste" attack, as the encrypted value in the new record would contain the wrong authenticator value.

The SQL Server authenticator argument is a **varbinary**, so the value can contain integers, **uniqueidentifiers** (GUIDs), or even strings. The initialization vector in the .NET cryptography classes is also a binary array; one could use this argument in the .NET classes to similar ends. Using identity columns may seem natural for this, but the identity value would have to be known in advance of the encryption operation. It is not clear to me how this would happen when a record is inserted. Frankly, this seems to be another instance where a **uniqueidentifier** (GUID) might be far more useful as the primary key of this table. Additionally, by using GUIDs you can perform the encryption on another server by using the .NET encryption classes and the GUID primary key and the encrypted values can be inserted into the table without fear of primary key conflicts.

Searching Salted Fields

It may seem to be a good idea to encrypt social security numbers (U.S. taxpayer numbers), but what happens when you need to find a record by social security number? Some merchants allow you to return an item without a receipt because they can look up the purchase record based on your credit card number. What happens when credit card numbers are encrypted?

Life is good as long as the SQL Server engine is simply used to store and retrieve encrypted information. When the engine must search, sort, or compare this data, serious performance problems arise. Unfortunately, the very salting technique that prevents users from comparing values also prevents the SQL engine from doing the same thing! To perform a lookup, the engine must individually decrypt every record as it looks for the matching record. While there are good reasons for encrypting credit card numbers and social security numbers, the fact that these encrypted values are salted effectively precludes the ability to perform these searches without individually decrypting the values.

Several approaches can make searching encrypted information more performant, but they also make the protected information less secure to varying degrees. Note that this is also true of database vendors who claim that their encrypted data can be indexed: they still leak information.

Method One

Encrypt the plain text without using a salt value (or use a consistent salt value). When executing a search, encrypt the search term by using the same salt value and compare the binary output with the encrypted values in the database. Identical plain text should generate identical encryptions. Note that unsalted data would have increased exposure to several attacks and allow users to confirm correctly guessed plain text values. Likewise, the data would also not be protected against copy-and-paste attacks by an authenticator argument.

Of course, this method could only be implemented by using the .NET classes, as SQL Server cryptographic functionality always salts encrypted data. You could use the SQL-CLR for this. Searches would not be case-insensitive.

Method Two

Use the SQL Server internal encryption function (always use the **EncryptByKey** function unless you are encrypting another key) and provide the authenticator argument. This creates fully salted encrypted values that are robustly protected against attack.

Additionally, create a column that contains a hash value of the original plain text value. If you want searches to be case-insensitive, first use the **lowercase()** function on the plain text value. When executing a search, create a hash of the (lowercase) search value and compare it to the values in the new column. The hash comparisons allow the search to execute very quickly.

Using the **HashBytes** function and specifying the SHA-1 algorithm gives you a 20-byte hash result. It would probably make sense to cast this output to a **binary(20)** and ensure that the column itself uses a matching data type. A fixed-size column may perform slightly better than a **varbinary(20)**. Specifying the MD4 or MD5 algorithms for the **HashBytes** function returns a 16-byte result, which can also cast as a **uniqueidentifier**.

Note that it is possible (though very unlikely) that multiple records could match the hash, so any records that match the hash should be decrypted and compared to the plain text search term. With judicious SQL, this can be done in a single query.

This approach still leaves the encryption vulnerable to a user guessing an encrypted value and confirming it by comparing the hashes, but a malicious user could not successfully copy the encrypted value into another record. Because the hashed values are not salted, this is also vulnerable to a "rainbow attack," which generally requires only time and storage to uncover the plain text from the hash, and it is also vulnerable to frequency analysis.

To work around some of these problems, Raul Garcia has an excellent blog entry [http://blogs.msdn.com/raulga/archive/2006/03/11/549754.aspx] detailing how to encrypt the hash itself by using a keyed-hash message authentication code (HMAC). This solves the problem of rainbow attacks against the hash value and also somewhat limits malicious users from confirming correct guesses by comparing hashes. Unfortunately, if the malicious user is able to write to the table (say, by updating their own record), he or she might still be able to generate values that are confirmed through hash matches. Because the hashes are unsalted, it may also be possible to determine encrypted values based upon their frequency. For example, if my table contains a list of cold war spies, I may not be able to decrypt the city column, but it would not be hard to determine which value represents "Moscow."

Method Three (recommended)

The third approach is quite like the previous one (salt encrypted values and add an additional unsalted hash column), but the hash values are truncated, or are of limited numeric precision. Whereas the previous method must handle the *possibility* of duplicate hash matches, this method addresses the *likelihood* of duplicate matches. The advantage of this approach is that a malicious user cannot confirm correctly guessed plain text values with certainty because the low resolution of the hash values make it possible that any two random plain text values would generate the same hash. The hashes are also much less vulnerable to frequency analysis.

Remember that cryptographic hashes are largely designed for signatures: their length provides a very high level of confidence that two messages will not generate the same hash value. In fact, it is improbable (although not impossible) that another message would ever be created that would generate the same hash. In our case, this is a feature we do not want, because it allows an attacker to confirm their guess of our plain text value.

As an extreme example of this approach, suppose our hash were reduced to one byte (**tinyint**), or 256 distinct values. Our search through the salted encryption records would still be substantially faster because the number of records requiring individual decryption would be about 256 times smaller (assuming an even distribution). However, any attacker trying to verify a guess of encrypted values would have 1:256 odds of a hash match by sheer chance, which offers very limited confirmation of a correct guess.

Naturally, a table containing a few hundred rows would generally have one to three hash matches when a **tinyint** hash column is used, which offers good performance and leaks almost no data about plain text source values. On the other hand, a table with a million rows would likely offer 3,900 matching hash records, each of which must be decrypted to complete the search. This would probably not result in good performance. In this case, increasing the resolution of the hash to two bytes (**smallint**), cuts the matching set of hashes to about 15 records, but it also offers an attacker 1:65,535 odds that a guessed value is correct.

You can calibrate the numeric precision of the hash result by using the Bitwise-And (&;) operator to filter unwanted high-order bits. This approach also enables you to specify the hash resolution precisely, as you can specify precisely the number of bits you want. By using this approach, it is also possible to perform an offline change to the precision of the hash.

An alternative approach to creating these "buckets" of hash matches is to truncate or reduce the precision of the plaintext value, and then create an unsalted hash from this value. This may offer more predictable control over the distribution and precision of the hash matches.

It might be a good idea to create triggers on the table that automatically calculates the hash value and populates the hash column. Properly implemented, this approach offers limited (and controlled) vulnerability to malicious attack, while still offering good search performance.

There is clearly a balance between the security of the data, the size of the table, and the performance of the searches. Increasing the numeric precision (number of bits) of the hash offers an attacker more confirmation of a correct guess, but also improves the performance of searches. There is no "one size" solution, as the correct answer depends on the size of the table, the frequency of the searches, and the criticality of the need to keep the plain text data secret.

.NET Classes vs. SQL Server Internal Cryptography

Both SQL Server internal cryptographic functionality and the .NET cryptographic classes use the Windows Crypto API, so do not expect the performance or security of one to be substantially better than the other. In general terms, SQL Server internal cryptographic functionality is much simpler to use and .NET classes are more flexible.

Note that there is little you can do with the .NET flexibility to make the data more secure than what SQL Server internal functions provide; SQL Server functions are already very robust. The exception is that you have access to newer, platform-dependent algorithms when using the .NET classes. For example, if you want to use the elliptic field algorithms that attend Vista and Windows Server 2008, you can do this with the current .NET classes, but not with SQL Server functions. If SQL Server were to allow this, you may not be able to decrypt your data if you moved to a backup server or mirror.

.NET Classes Advantages

· Cryptographic functionality is available both within SQL Server (via SQL-CLR integration) and middle-tier or even client machines (Be very careful about decrypting on client computers that are not secure, as that may expose the keys themselves.) This means that the cryptographic CPU load can be moved to or shared with other servers.

· Since multiple computers can participate in the cryptographic processing, encrypted data can be streamed while still encrypted, and then decrypted in a downstream process.

· Depending on the platform, more and newer algorithms are available and you have explicit control over all of the arguments.

· The .NET classes can be extended to participate in a PKI infrastructure. For example, certificates could be tested for their presence on a Certificate Revocation List.

SQL Server Internal Cryptography Advantages

· The functionality is natively available within SQL Server. (Although cryptographic functions created via the SQL-CLR can also be called via SQL, this would have to be installed on every server that required it.)

· The cryptographic functionality is simple and robust, whereas in the case of the .NET classes, certain arguments and practices must be explicitly used to provide an equivalent level of security.

· Data can be moved to another server, such as a backup server or mirror, and it can still be decrypted—even if hosted on an older OS platform.

· The environment leverages the built-in permissions architecture to control access to keys. Contrast this with .NET classes, where keys must be loaded, persisted, and secured in some external store, which may create access control issues.

Conclusion

The most common problem with cryptographic functionality is that it is used to address problems it was not designed to solve. Common examples include:

· Using asymmetric key pairs in situations when key communication is not an issue, such as with data stored in a SQL Server database.

· Using encryption to control access to data. SQL Server permissions are robust and designed for this. Unless access to the storage media can be compromised (via lost laptop drives, unencrypted backup tapes, shared storage, and so on), encryption does add meaningful security.

In the case of signatures, it is not widely appreciated how this functionality solves an important problem.

The best part of SQL Server cryptographic functionality is that it is relatively simple to create robust security. The native functionality offers key storage and access control and the built-in encryption functionality offers robust security (including random salt, authenticator values, and so on). The simplicity of the implementation helps ensure that those who depend upon it to protect their data or processes will not be disappointed.

About the Author

John Hicks is an Architect in the Industry Solutions Group at Microsoft. You can reach him at john.hicks@microsoft.com or through his blog at http://blogs.msdn.com/johnhicks [http://blogs.msdn.com /johnhicks] .

For more information:

SQL Server Web site: http://www.microsoft.com/sqlserver/ [http://www.microsoft.com/sqlserver/]

SQL Server TechCenter: http://technet.microsoft.com/en-us/sqlserver/default.aspx [http://technet.microsoft.com/en-us/sqlserver/default.aspx]

SQL Server DevCenter: http://msdn2.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.

Tags:

Community Content