Microsoft SDL Cryptographic Recommendations

October 2016

Table of Contents

Introduction

This document contains recommendations and best practices for using encryption on Microsoft platforms. Much of the content here is paraphrased or aggregated from Microsoft's own internal security standards used to create the Security Development Lifecycle. It is meant to be used as a reference when designing products to use the same APIs, algorithms, protocols and key lengths that Microsoft requires of its own products and services.

Developers on non-Windows platforms may also benefit from these recommendations. While the API and library names may be different, the best practices involving algorithm choice, key length and data protection are similar across platforms.

Security Protocol, Algorithm and Key Length Recommendations

SSL/TLS versions

Products and services should use cryptographically secure versions of SSL/TLS:

- TLS 1.2 should be enabled

- TLS 1.1 and TLS 1.0 should be enabled for backward compatibility only
- SSL 3 and SSL 2 should be disabled by default

Symmetric Block Ciphers, Cipher Modes and Initialization Vectors

Block Ciphers

For products using symmetric block ciphers:

- Advanced Encryption Standard (AES) is recommended for new code.
- Three-key triple Data Encryption Standard (3DES) is permissible in existing code for backward compatibility.
- All other block ciphers, including RC2, DES, 2-Key 3DES, DESX, and Skipjack, should only be used for decrypting old data, and should be replaced if used for encryption.

For symmetric block encryption algorithms, a minimum key length of 128 bits is recommended. The only block encryption algorithm recommended for new code is AES (AES-128, AES-192, and AES-256 are all acceptable, noting that AES-192 lacks optimization on some processors). Three-key 3DES is currently acceptable if already in use in existing code; transition to AES is recommended. DES, DESX, RC2, and Skipjack are no longer considered secure. These algorithms should only be used for decrypting existing data for the sake of backward-compatibility, and data should be re-encrypted using a recommended block cipher.

Cipher Modes

Symmetric algorithms can operate in a variety of modes, most of which link together the encryption operations on successive blocks of plaintext and ciphertext.

Symmetric block ciphers should be used with one of the following cipher modes:

- Cipher Block Chaining (CBC)
- Ciphertext Stealing (CTS)
- XEX-Based Tweaked-Codebook with Ciphertext Stealing (XTS)

Some other cipher modes like those included below have implementation pitfalls that make them more likely to be used incorrectly. In particular, the Electronic Code Book (ECB) mode of operation should be avoided. Reusing the same initialization vector (IV) with block ciphers in "streaming ciphers modes" such as CTR may cause encrypted data to be revealed. Additional security review is recommended if any of the below modes are used:

- Output Feedback (OFB)
- Cipher Feedback (CFB)
- Counter (CTR)
- Counter with CBC-MAC (CCM)
- Galois/Counter Mode (GCM)
- Anything else not on the "recommended" list above

Initialization Vectors (IV)

All symmetric block ciphers should also be used with a cryptographically strong random number as an initialization vector. Initialization vectors should never be a constant value. See Random Number Generators for recommendations on generating cryptographically strong random numbers.

Initialization vectors should never be reused when performing multiple encryption operations, as this can reveal information about the data being encrypted, particularly when using streaming cipher modes like Output Feedback (OFB) or Counter (CTR).

Asymmetric Algorithms, Key Lengths, and Padding Modes

RSA

- RSA should be used for encryption, key exchange and signatures.
- RSA encryption should use the OAEP or RSA-PSS padding modes. Existing code should use PKCS #1 v1.5 padding mode for compatibility only.
- Use of null padding is not recommended.
- Keys >= 2048 bits are recommended

ECDSA

- ECDSA with >= 256 bit keys is recommended
- ECDSA-based signatures should use one of the three NIST-approved curves (P-256, P-384, or P521).

ECDH

- ECDH with >= 256 bit keys is recommended
- ECDH-based key exchange should use one of the three NIST-approved curves (P-256, P-384, or P521).

Integer Diffie-Hellman

- Key length >= 2048 bits is recommended
- The group parameters should either be a well-known named group (e.g., RFC 7919), or generated by a trusted party and authenticated before use

Key Lifetimes

- All asymmetric keys should have a maximum five-year lifetime, recommended one-year lifetime.
- All symmetric keys should have a maximum three-year lifetime; recommended one-year lifetime.
- You should provide a mechanism or have a process for replacing keys to achieve the limited active lifetime. After the end of its active lifetime, a key should not be used to produce new data (for example, for encryption or signing), but may still be used to read data (for example, for decryption or verification).

Random Number Generators

All products and services should use cryptographically secure random number generators when randomness is required.

CNG

- Use BCryptGenRandom with the BCRYPT_USE_SYSTEM_PREFERRED_RNG flag

CAPI

- Use CryptGenRandom to generate random values.

Win32/64

- Legacy code can use RtlGenRandom in kernel mode
- New code should use BCryptGenRandom or CryptGenRandom.
- The C function Rand_s() is also recommended (which on Windows, calls CryptGenRandom)
- Rand_s() is a safe and performant replacement for Rand(). Rand() should not be used for any cryptographic applications, but is ok for internal testing only.
- The SystemPrng function is recommended for kernel-mode code.

.NET

- Use RNGCryptoServiceProvider or RNGCng.

Windows Store Apps

- Store Apps can use CryptographicBuffer.GenerateRandom or CryptographicBuffer.GenerateRandomNumber.

Not Recommended

- Insecure functions related to random number generation include rand, System.Random (.NET), GetTickCount and GetTickCount64
- Use of the dual elliptic curve random number generator ("DUAL_EC_DRBG") algorithm is not recommended.

Windows Platform-supported Crypto Libraries

On the Windows platform, Microsoft recommends using the crypto APIs built into the operating system. On other platforms, developers may choose to evaluate non-platform crypto libraries for use. In general, platform crypto libraries will be updated more frequently since they ship as part of an operating system as opposed to being bundled with an application.

Any usage decision regarding platform vs non-platform crypto should be guided by the following requirements:

1. The library should be a current in-support version free of known security vulnerabilities
2. The latest security protocols, algorithms and key lengths should be supported
3. (Optional) The library should be capable of supporting older security protocols/algorithms for backwards compatibility only

Native Code

- Crypto Primitives: If your release is on Windows or Windows Phone, use CNG if possible. Otherwise, use the CryptoAPI (also called CAPI, which is supported as a legacy component on Windows from Windows Vista onward).
- SSL/TLS/DTLS: WinINet, WinHTTP, Schannel, IXMLHTTPRequest2, or IXMLHTTPRequest3.
    - WinHTTP apps should be built with WinHttpSetOption in order to support TLS 1.2
- Code signature verification: WinVerifyTrust is the supported API for verifying code signatures on Windows platforms.

- Certificate Validation (as used in restricted certificate validation for code signing or SSL/TLS/DTLS): CAPI2 API; for example, CertGetCertificateChain and CertVerifyCertificateChainPolicy

Managed Code

- Crypto Primitives: Use the API defined in System.Security.Cryptography namespace---the CNG classes are preferred.
- Use the latest version of the .Net Framework available. At a minimum this should be .Net Framework version 4.6. If an older version is required, ensure the "SchUseStrongCrypto" regkey is set to enable TLS 1.2 for the application in question.
- Certificate Validation: Use APIs defined under the System.Security.Cryptography.X509Certificates namespace.
- SSL/TLS/DTLS: Use APIs defined under the System.Net namespace (for example, HttpWebRequest).

Key Derivation Functions

Key derivation is the process of deriving cryptographic key material from a shared secret or a existing cryptographic key. Products should use recommended key derivation functions. Deriving keys from user-chosen passwords, or hashing passwords for storage in an authentication system is a special case not covered by this guidance; developers should consult an expert.

The following standards specify KDF functions recommended for use:

- NIST SP 800-108: *Recommendation For Key Derivation Using Pseudorandom Functions*. In particular, the KDF in counter mode, with HMAC as a pseudorandom function
- NIST SP 800-56A (Revision 2): *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*. In particular, the "Single-Step Key Derivation Function" in Section 5.8.1 is recommended.

To derive keys from existing keys, use the BCryptKeyDerivation API with one of the algorithms:

- BCRYPT_SP800108_CTR_HMAC_ALGORITHM
- BCRYPT_SP80056A_CONCAT_ALGORITHM

To derive keys from a shared secret (the output of a key agreement) use the BCryptDeriveKey API with one of the following algorithms:

- BCRYPT_KDF_SP80056A_CONCAT
- BCRYPT_KDF_HMAC

Certificate Validation

Products that use SSL, TLS, or DTLS should fully verify the X.509 certificates of the entities they connect to. This includes verification of the certificates':

- Domain name.
- Validity dates (both beginning and expiration dates).
- Revocation status.

- Usage (for example, “Server Authentication” for servers, “Client Authentication” for clients).
- Trust chain. Certificates should chain to a root certification authority (CA) that is trusted by the platform or explicitly configured by the administrator.

If any of these verification tests fail, the product should terminate the connection with the entity.

Clients that trust “self-signed” certificates (for example, a mail client connecting to an Exchange server in a default configuration) may ignore certificate verification checks. However, self-signed certificates do not inherently convey trust, support revocation, or support key renewal. You should only trust self-signed certificates if you have obtained them from another trusted source (for example, a trusted entity that provides the certificate over an authenticated and integrity-protected transport).

Cryptographic Hash Functions

Products should use the SHA-2 family of hash algorithms (SHA256, SHA384, and SHA512). Truncation of cryptographic hashes for security purposes to less than 128 bits is not recommended.

MAC/HMAC/keyed hash algorithms

A message authentication code (MAC) is a piece of information attached to a message that allows its recipient to verify both the authenticity of the sender and the integrity of the message using a secret key.

The use of either a hash-based MAC (HMAC) or block-cipher-based MAC is recommended as long as all underlying hash or symmetric encryption algorithms are also recommended for use; currently this includes the HMAC-SHA2 functions (HMAC-SHA256, HMAC-SHA384 and HMAC-SHA512).

Truncation of HMACs to less than 128 bits is not recommended.

Design and Operational Considerations

- You should provide a mechanism for replacing cryptographic keys as needed. Keys should be replaced once they have reached the end of their active lifetime or if the cryptographic key is compromised. Whenever you renew a certificate, you should renew it with a new key.
- Products using cryptographic algorithms to protect data should include enough metadata along with that content to support migrating to different algorithms in the future. This should include the algorithm used, key sizes, initialization vectors, and padding modes.
    - For more information on Cryptographic Agility, see Cryptographic Agility on MSDN.
- Where available, products should use established, platform-provided cryptographic protocols rather than re-implementing them. This includes signing formats (e.g. use a standard, existing format).
- Symmetric stream ciphers such as RC4 should not be used. Instead of symmetric stream ciphers, products should use a block cipher, specifically AES with a key length of at least 128 bits.
- Do not report cryptographic operation failures to end-users. When returning an error to a remote caller (e.g. web client, or client in a client-server scenario), use a generic error message only.

    - Avoid providing any unnecessary information, such as directly reporting out-of-range or invalid length errors. Log verbose errors on the server only, and only if verbose logging is enabled.
- Additional security review is highly recommended for any design incorporating the following:
    - A new protocol that is primarily focused on security (such as an authentication or authorization protocol)
    - A new protocol that uses cryptography in a novel or non-standard way
    - Example considerations include:
        - Will a product that implements the protocol call any crypto APIs or methods as part of the protocol implementation?
        - Does the protocol depend on any other protocol used for authentication or authorization?
        - Will the protocol define storage formats for cryptographic elements, such as keys?
- Self-signed certificates are not recommended for production environments. Use of a self-signed certificate, like use of a raw cryptographic key, does not inherently provide users or administrators any basis for making a trust decision.
    - In contrast, use of a certificate rooted in a trusted certificate authority makes clear the basis for relying on the associated private key and enables revocation and updates in the event of a security failure.

Encrypting Sensitive Data prior to Storage

DPAPI/DPAPI-NG

For data that needs to be persisted across system reboots:

- CryptProtectData
- CryptUnprotectData
- NCryptProtectSecret (Windows 8 CNG DPAPI)

For data that does not need to be persisted across system reboots:

- CryptProtectMemory
- CryptUnprotectMemory

For data that needs to be persisted and accessed by multiple domain accounts and computers:

- NCryptProtectSecret (in CNG DPAPI, available as of Windows 8)
- Microsoft Azure KeyVault

SQL Server TDE

You can use SQL Server Transparent Data Encryption (TDE) to protect sensitive data.

You should use a TDE database encryption key (DEK) that meets the SDL cryptographic algorithm and key strength requirements. Currently, only AES_128, AES_192 and AES_256 are recommended; TRIPLE_DES_3KEY is not recommended.

There are some important considerations for using SQL TDE that you should keep in mind:

- SQL Server does not support encryption for FILESTREAM data, even when TDE is enabled.
- TDE does not automatically provide encryption for data in transit to or from the database; you should also enable encrypted connections to the SQL Server database. Please see Enable Encrypted Connections to the Database Engine (SQL Server Configuration Manager) for guidance on enabling encrypted connections.
- If you move a TDE-protected database to a different SQL Server instance, you should also move the certificate that protects the TDE Data Encryption Key (DEK) and install it in the master database of the destination SQL Server instance. Please see the TechNet article Move a TDE Protected Database to Another SQL Server for more details.

Credential Management
Use the Windows Credential Manager API or Microsoft Azure KeyVault to protect password and credential data.

Windows Store Apps
Use the classes in the Windows.Security.Cryptography and Windows.Security.Cryptography.DataProtection namespaces to protect secrets and sensitive data.

- ProtectAsync
- ProtectStreamAsync
- UnprotectAsync
- UnprotectStreamAsync

Use the classes in the Windows.Security.Credentials namespace to protect password and credential data.

.NET
For data that needs to be persisted across system reboots:

- ProtectedData.Protect
- ProtectedData.Unprotect

For data that does not need to be persisted across system reboots:

- ProtectedMemory.Protect
- ProtectedMemory.Unprotect

For configuration files, use either RSAProtectedConfigurationProvider or DPAPIProtectedConfigurationProvider to protect your configuration, using either RSA encryption or DPAPI, respectively.

The RSAProtectedConfigurationProvider can be used across multiple machines in a cluster. See Encrypting Configuration Information Using Protected Configuration for more information.