Cryptographic Agility

Bryan Sullivan
Senior Security Program Manager
Microsoft Corporation

Throughout history, people have used various forms of ciphers to conceal information from their adversaries. Julius Caesar used a three-place shift cipher (the letter A is converted three places down the alphabet to the letter D, B is converted to E, etc.) to communicate battle plans with his generals. During World War II, the German Navy used a significantly more advanced system – the Enigma machine – to encrypt messages sent to their U-boats. Today, we use even more sophisticated encryption mechanisms as part of the Public Key Infrastructure that helps us perform secure transactions on the Internet.

However, for as long as cryptographers have been making secret codes, there have also been cryptanalysts trying to break those codes and steal the information; and sometimes, the code breakers win. Cryptographic algorithms once considered secure are later broken and rendered useless. Sometimes subtle flaws are found in the algorithms, and sometimes it is simply a matter of attackers having access to more raw computing power to perform brute-force attacks. The Germans' Enigma machine was unbreakable early in the war, but by the end, Allied code breakers had exploited flaws in the system and were able to decrypt many Enigma messages. The shift cipher, once strong enough for Julius Caesar's battle plans, is now broken so easily that it's only suitable for children's secret decoder ring toys[1].

In more recent developments, a group of security researchers[2] have demonstrated weaknesses in the MD5 hash algorithm in the form of collisions; that is, they have shown that two different messages can have the same computed MD5 hash value. Furthermore, they have created a proof-of-concept attack against this weakness targeted at the Public Key Infrastructure. By purchasing a specially-crafted website certificate from a Certificate Authority (CA) that uses MD5 to sign its certificates, the researchers were able to create a rogue CA certificate that could effectively be used to impersonate potentially any site on the internet. Their conclusion was that MD5 is not appropriate for signing digital certificates and that a stronger alternative such as one of the SHA-2 algorithms should be used[3].

While these findings were certainly cause for concern, they were not actually a huge surprise. Theoretical MD5 weaknesses had been demonstrated for years, and the use of MD5 in Microsoft products has been banned per the Microsoft SDL Cryptographic Standards since 2005. Other once-popular algorithms such as SHA-1 and RC2 are similarly banned. See Figure 1 for a complete list of the SDL banned and approved cryptographic algorithms.

---

[1] Be sure to drink your Ovaltine!
[2] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger; 25th Chaos Communication Congress (25C3), December 2008
[3] http://www.win.tue.nl/hashclash/rogue-ca/

Algorithm Type	**Banned** *Algorithms to be replaced in existing code or used only for decryption*	**Minimally Acceptable** *Algorithms acceptable for existing code (except sensitive data)*	**Recommended** *Algorithms for new code*
Symmetric Block	DES, 3DES (2 key), DESX, RC2, SKIPJACK	3DES (3 key)	AES (>=128 bit)
Symmetric Stream	SEAL, CYLINK_MEK, RC4 (<128bit)	RC4 (>= 128bit)	None – Block cipher is preferred
Asymmetric	RSA (<2048 bit), Diffie-Hellman (<2048 bit)		RSA (>=2048bit), Diffie-Hellman (>=2048bit), ECC (>=256bit)
Hash (includes HMAC usage)	SHA-0 (SHA), SHA-1, MD2, MD4, MD5	3DES MAC	SHA-2 (includes: SHA-256, SHA-384, SHA-512)

Figure 1: Microsoft SDL Approved Cryptographic Algorithms

However, even if you follow these standards in your own code, only choosing the most secure algorithms and the longest key lengths, there's no guarantee that the code you write today will remain secure tomorrow – in fact, it likely will not remain secure, if history is any indication. Just like with MD5, a researcher could discover a weakness in AES or RSA and render your carefully crafted code vulnerable to attack.

You could address this unpleasant scenario reactively, by going through all your old applications' code bases, picking out instantiations of vulnerable algorithms, replacing them with new algorithms, rebuilding the applications, running them through regression tests, and finally issuing patches or service packs to all of your users. However, this is not only a lot of work for you, but still leaves your users at risk until you can get the fixes shipped. A better alternative is to plan for this scenario from the beginning. Several popular development frameworks including Java Development Kit (JDK) and .NET include cryptographic agility features. Instead of hard-coding specific cryptographic algorithms or implementations into your code, take advantage of these crypto agility features to help future-proof your application against potential future exploits.

Both JDK and .NET take advantage of their object-oriented natures to provide cryptographic agility. Implementations of specific algorithms are represented as classes that derive from abstract classes representing the general type of cryptographic algorithm type. For example, the .NET Framework class System.Security.Cryptography.SHA512CryptoServiceProvider that implements the SHA-512 hash algorithm derives from the abstract class System.Security.Cryptography.SHA512, which itself derives from the abstract class System.Security.Cryptography.HashAlgorithm.

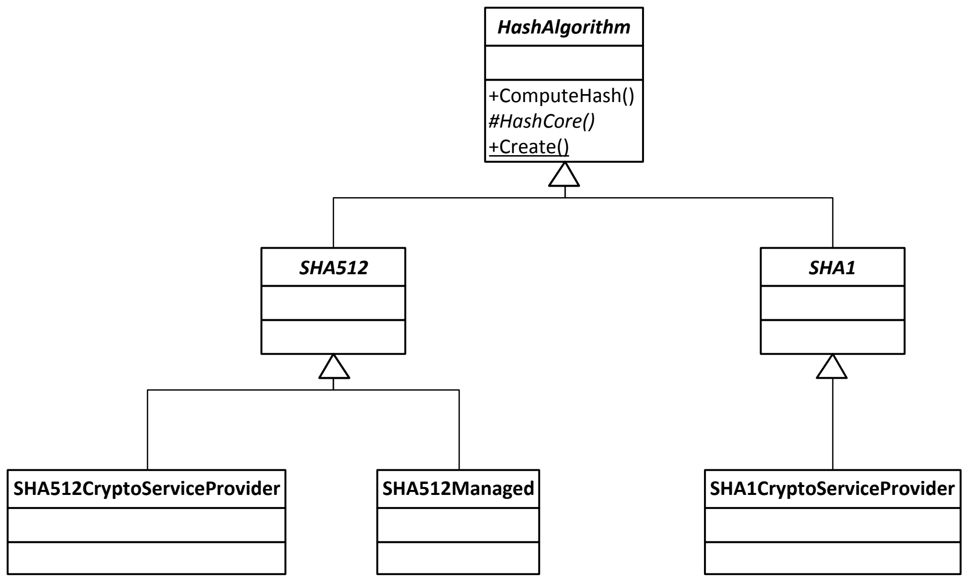

Figure 2: UML Diagram Demonstration of .NET System.Security.Cryptography Inheritance Structure

The base class HashAlgorithm defines a public method ComputeHash that calls into an abstract protected method HashCore, which performs the actual hash computation. Any instantiable class that derives from HashAlgorithm, such as SHA512CryptoServiceProvider, must implement its own HashCore method.

HashAlgorithm also implements a static (or class) method Create, which allows it to act as a factory class for the individual implementation classes that derive from it. The Create method takes a string parameter that represents the name of the algorithm to be instantiated:

```
HashAlgorithm myHash = HashAlgorithm.Create(desiredAlgorithmName);
byte[] result = myHash.ComputeHash(data);
```

This architecture permits the developer to avoid hardcoding specific implementation classes, which would lead to a cryptographically non-agile application. He or she can load the desired algorithm name from a configuration file or a database or some other user-configurable location (or more specifically, a location configurable by an administrator of the deployed application). The application administrator can now determine exactly which algorithm and implementation the application will use. They may choose to replace an algorithm that has recently been broken with one still considered secure (for example, to replace MD5 with SHA-256) or to proactively replace a secure algorithm with an alternative with a longer bit-length (for example, to replace SHA-256 with SHA-512).

The JDK cryptographic classes are architected in a similar fashion. The JDK factory class for cryptographic hashing is java.security.MessageDigest, and the factory method is getInstance.

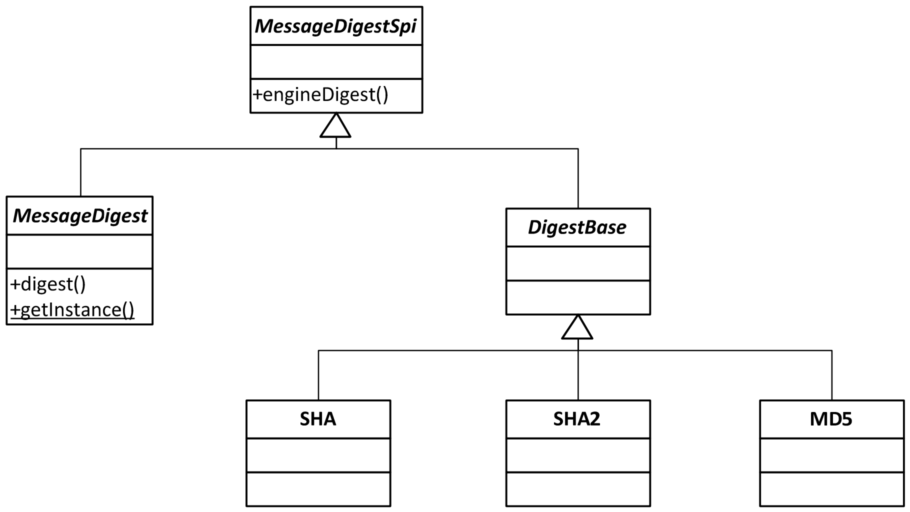

Figure 3: UML Diagram Demonstration of Java Cryptography Architecture (JCA) Inheritance Structure

The getInstance method takes a string parameter for the desired algorithm name and returns a MessageDigest object which encapsulates a MessageDigestSpi object (i.e., a message digest service provider interface) that performs the actual hashing:

```
MessageDigest myDigest = MessageDigest.getInstance(desiredAlgorithmName);
byte[] result = myDigest.digest(data);
```

The same architectural principles apply in both .NET and JDK to other cryptographic concepts besides hashing:

- Java Cryptography Architecture (JCA)[4]
    - javax.crypto.Cipher (supports both symmetric and public-key encryption)
    - javax.crypto.KeyAgreement (for example, Diffie-Hellman)
    - java.security.KeyFactory (serialization for public/private keys)
    - javax.crypto.KeyGenerator (for creating symmetric keys)
    - java.security.KeyPairGenerator (for creating public/private key pairs)
    - javax.crypto.Mac (message authentication codes)
    - java.security.MessageDigest (cryptographic hashes)
    - javax.crypto.SecretKeyFactory (serialization for symmetric keys)
    - java.security.Signature (private-key signed message digests)
- .NET System.Security.Cryptography
    - SymmetricAlgorithm (symmetric encryption)
    - AsymmetricAlgorithm (private-key encryption; includes RSA, Diffie-Hellman, ECC)
    - HashAlgorithm (cryptographic hashes)

[4] Source: Java Cryptography; Knudsen, Jonathan; O'Reilly Media

- KeyedHashAlgorithm (keyed cryptographic hashes)
- HMAC (hash-based message authentication codes)

It's important to note that while removing all hardcoded algorithm implementation class references from the application source code is necessary for cryptographic agility, it is not sufficient in and of itself. You will need to plan ahead regarding storage size, for both local variables (transient storage) and database and XML schemas (persistent storage). For example, MD5 hashes are always 128 bits in length. If you budget exactly 128 bits in your code or schema to store hash output, you will not be able to upgrade to SHA-256 (256 bit-length output) or SHA-512 (512 bit-length output).

This does beg the question of how much storage is enough. Is 512 bits enough, or should you use 1,024, 2,048, or more? It is impossible to provide a hard rule here, because every application has different requirements, but as a rule of thumb a good recommendation is that you budget twice as much space for hashes as you currently use. For symmetric- and asymmetric-encrypted data, you might reserve an extra 10% of space at most. It's unlikely that new algorithms with output sizes significantly larger than existing algorithms will be widely accepted.

However, applications that store hash values or encrypted data in a persistent state (for example, in a database or file) have bigger problems than reserving enough space. If the application persists data using one algorithm and then tries to operate on that data later using a different algorithm, you will not get the results you expect. For example, consider a user authentication module, written to store and compare hashes of passwords. When a user tries to log on, the module compares the hash of the password supplied by the user to the stored hash in the database. If the hashes match, the user is considered authentic. However, if a hash is stored in one format (say MD5) and an application is upgraded to use another algorithm (say SHA-256), users will never be able to log on because the SHA-256 hash value of the passwords will always be different from the MD5 hash value of those same passwords.

It is possible to get around this issue in some cases by storing the original algorithm as metadata along with the actual data. Then, when operating on stored data, use the agility methods (or reflection) to instantiate the algorithm originally used instead of the current algorithm. In some cases it will be necessary to store substantially more metadata than just the original algorithm name, such as initialization vectors, salt values, block size, spin count, etc. A comprehensive list of these metadata values can be found in the Microsoft Office Document Cryptography Structure Specification[5] (MS-OFFCRYPTO), which defines an XML EncryptionInfo structure that incorporates the necessary metadata.

In conclusion, given the time and expense of recoding your application in response to a broken cryptographic algorithm, not to mention the danger to your users until you can get a new version deployed to them, it would be wise to plan for this occurrence and write your application in a cryptographically agile fashion.

[5] http://msdn.microsoft.com/en-us/library/cc313071(office.12).aspx