**Audio File Format**

In order to demonstrate the use of Steganography techniques combined with encryption, the AU format used on Sun and NeXT machines was chosen as the host audio file. Sun AU format is well documented elsewhere and Java Sound API provides convenient ways to handle formatted audio data.

Formatted audio data refers to sound in any of a number of standard formats. The Java Sound API distinguishes between *data formats* and *file formats*. A data format tells you how to interpret a series of bytes of “raw” sampled audio data, such as samples that have been captured from the microphone input. You might need to know, for example how many bits constitute one sample (the representation of the shortest instant of sound), and similarly you might need to know the sound’s sample rate (hoe fast the samples are supposed to follow one another). When setting up for playback or capture, you specify the data format of the sound you are capturing or playing.

In the Java Sound API, a *data format* is represented by an AudioFormat object, which includes the following attributes:

* Encoding technique, usually pulse code modulation (PCM)
* Number of channels (1 for mono, 2 for stereo, etc.)
* Sample rate (number of samples per second, per channel)
* Number of bits per sample (per channel)
* Frame rate
* Frame size in bytes
* Byte order (big-endian or little-endian)

PCM is one kind of encoding of the sound waveform. Compact disks, for example, use linear PCM-encoded sound. Mu-law encoding and a-law encoding are common nonlinear encodings that provide a more compressed version of the audio data.

A frame contains the data for all channels at a particular time. For PCM-encoded data, the frame is simply the set of simultaneous samples in all channels, for a given instant in time, without any additional information. In this case, the frame rate is equal to the sample rate, and the frame size in bytes is the number of channels multiplied by the sample size in bits, divided by the no of bits in a byte.

A *file format* specifies the structure of a sound file, including not only the format of the raw audio data in the file, but also other information that can be stored in the file. Sound files come in various standard varieties, such as WAVE (also known as WAV, and often associated with PCs), AIFF (often associated with Macintosh OS), and AU (often associated with UNIX systems). The different types of sound files have different structures. For example, they might have a different arrangement of data in the file’s ‘header’. A header contains descriptive information that typically precedes the file’s actual audio samples, although some file formats allow successive “chunks” of descriptive and audio data. The header includes a specification of the data format that was used for storing the audio in the second file. Any of these types of sound file can contain various data formats (although usually there is only one data format within a given file), and the same data format can be used in files that have different file formats.

In the Java Sound API, a file format is represented by an AudioFileFormat object, which contains:

* The file type (WAVE, AIFF, AU, etc.)
* The files length in bytes
* The length, in frames, of the audio data contained in the file
* An AudioFormat object that specifies the data format of the audio data contained in the file.

In our implementation, we treat the AU file as 8-bit Mu-law encoded.

**Encryption method used:**

We chose a password Based Encryption scheme based on RSA Laboratories PKCS #5 v2.0 standard [3].

The recommendations are intended for general application within computer and communications systems, and as such include a fair amount of flexibility. They are particularly intended for the protection of sensitive information such as private keys, as in PKCS #8 [21]. It is expected that application standards based on these specifications may include additional constraints.

Other cryptographic techniques based on passwords, such as password-based key entity authentication and key establishment protocols [4][5][22] are outside the scope of this document. Guidelines for the selection of passwords are also outside the scope.

This document supersedes PKCS #5 version 1.5 [20], but includes compatible techniques.

# 

# Notation

*C* ciphertext, an octet string

*c* iteration count, a positive integer

*DK* derived key, an octet string

*dkLen* length in octets of derived key, an integer

*Enc* underlying encryption scheme

*EM* encoded message, an octet string

*Hash* underlying hash function

*hLen* output length of hash function, an integer

*l* length in blocks of derived key, an integer

*IV* initialization vector, an octet string

*K* encryption key, an octet string

*KDF* key derivation function

*M* message, an octet string

*P* password, an octet string

*PS* padding string, an octet string

*psLen* length in octets of padding string, an integer

*S* salt, an octet string

*T* message authentication code, an octet string

*T*1,…,*Tc*, *Tl* intermediate values, octet strings

01, 02, …, 08 octets with value 1, 2, …, 8

\xorbit-wise exclusive-or of two octet strings

|| **.** || octet length operator

|| concatenation operator

<*i*..*j*> substring extraction operator: extracts octets *i* through *j*, 0 ≤ *i* ≤ *j*

# 

# Salt and iteration count

Inasmuch as salt and iteration count are central to the techniques defined in this document, some further discussion is warranted.

## Salt

A salt in password-based cryptography has traditionally served the purpose of producing a large set of keys corresponding to a given password, among which one is selected at random according to the salt. An individual key in the set is selected by applying a key derivation function *KDF*, as

*DK* = *KDF* (*P*, *S*)

where *DK* is the derived key, *P* is the password, and *S* is the salt. This has two benefits:

1. It is difficult for an opponent to precompute all the keys corresponding to a dictionary of passwords, or even the most likely keys. If the salt is 64 bits long, for instance, there will be as many as 264 keys for each password. An opponent is thus limited to searching for passwords after a password-based operation has been performed and the salt is known.

2. It is unlikely that the same key will be selected twice. Again, if the salt is 64 bits long, the chance of “collision” between keys does not become significant until about 232 keys have been produced, according to the Birthday Paradox. This addresses concerns about interactions between multiple uses of the same key, which may apply for some encryption and authentication techniques.

In password-based encryption, the party encrypting a message can gain assurance that these benefits are realized simply by selecting a large and sufficiently random salt when deriving a encryption key from a password. A party generating a message authentication code can gain such assurance in a similar fashion.

The party decrypting a message or verifying a message authentication code, however, cannot be sure that a salt supplied by another part has actually been generated at random. It is possible, for instance, that an opponent may have copied the salt from another password-based operation, in an attempt to exploit interactions between multiple uses of the same key. For instance, the opponent may take the salt for an encryption operation with a 80-bit key and provide it to a party as though it were for a 40-bit key. If the party performs a decryption with the resulting key, the opponent may be able to determine the 40-bit key from the result of the decryption operation, and thereby solve for half of the 80-bit. Similar attacks are possible in the case of message authentication.

To defend against such attacks, either the interactions between multiple uses of the same key should be carefully analyzed, or the salt should contain data that explicitly distinguishes between different operations. For instance, the salt might have an additional, non-random octet that specifies whether the derived key is for encryption, for message authentication, or for some other operation.

Based on this, the following is recommended for salt selection:

1. If there is no concern about interactions between multiple uses of the same key with the password-based encryption and authentication techniques supported for a given password, then the salt may be generated at random. It should be at least eight octets (64 bits) long.

2. Otherwise, the salt should contain data that explicitly distinguishes between different operations, in addition to a random part that is at least eight octets long. For instance, the salt could have an additional non-random octet that specifies the purpose of the derived key. Alternatively, it could be the encoding of a structure that specifies detailed information about the derived key, such as the encryption or authentication technique and a sequence number among the different keys derived from the password. The particular format of the additional data is left to the application.

**Note.** If a random number generator or pseudorandom generator is not available, a deterministic alternative for generating the salt (or the random part of it) is to apply a password-based key derivation function to the password and the message *M* to be processed. For instance, the salt could be computed with a key derivation function as *S* = *KDF* (*P*, *M*). This approach is not recommended if the message *M* is known to belong to a small message space (e.g., “Yes” or “No”), however, since then there will only be a small number of possible salts.

## Iteration count

An iteration count has traditionally served the purpose of increasing the cost of producing keys from a password, thereby also increasing the difficulty of attack. For the methods in this document, a minimum of 1000 iterations is recommended. This will increase the cost of exhaustive search for passwords significantly, without a noticeable impact in the cost of deriving individual keys.

# Key derivation functions

A *key derivation function* produces a *derived key* from a *base key* and other parameters. In a *password-based key derivation function*, the base key is a password and the other parameters are a salt value and an iteration count, as outlined in Section 3.

The primary application of the password-based key derivation functions defined here is in the encryption schemes in Section 6 and the message authentication scheme in Section 7. Other applications are certainly possible, hence the independent definition of these functions.

Two functions are specified in this section: PBKDF1 and PBKDF2. PBKDF2 is recommended for new applications; PBKDF1 is included only for compatibility with existing applications, and is not recommended for new applications.

A typical application of the key derivation functions defined here might include the following steps:

1. Select a salt *S* and an iteration count *c*, as outlined in Section 4.

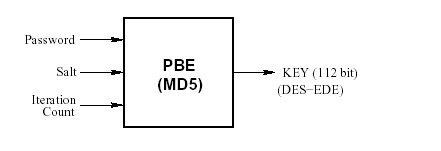
3. Select a length in octets for the derived key, *dkLen*.

4. Apply the key derivation function to the password, the salt, the iteration count and the key length to produce a derived key.

5. Output the derived key.

Since a password is not directly applicable as a key to any conventional cryptosystem, however, some processing of the password is required to perform cryptographic operations with it. Moreover, as passwords are often chosen from a relatively small space, special care is required in that processing to defend against search attacks. A general approach to password based cryptography, for the protection of password tables, is to combine a password with a salt to produce a key. The salt can be viewed as an index into a large set of keys derived from the password, and need not be kept secret. Although it may be possible for an opponent to construct a table of possible passwords (a so-called “dictionary attack”), constructing a table of possible keys will be difficult, since there will be many possible keys for each password. An opponent will thus be limited to searching through passwords separately for each salt.

Another approach to password-based cryptography is to construct key derivation techniques that are relatively expensive, thereby increase the cost of exhaustive search. One way to do this is to include an iteration count in the key derivation technique, indicating how many times to iterate some underlying function by which keys are derived. A modest number of iterations, say 1000, is not likely to be a burden for legitimate parties when computing a key, but will be a significant burden of opponents.



Key Derivation Function (KDF) in PBE

Salt and iteration count formed the basis for password-based encryption in PKCS #5 v2.0. The PBE schemes here are based on an underlying, conventional encryption schemes (for an example, in this implementation, triple DES-EDE with two keys in CBC mode), where the key for the conventional scheme is derived from the password.

A salt in password-based cryptography has traditionally served the purpose of producing a large set of keys corresponding to a given password, among which one is selected at random according to the salt. An individual key in the set is selected by applying a key derivation function KDF, as

*DK = KDF (P, S)*

where *DK* is the derived key, *P* is the password and *S* is the salt.

With this scheme, it is difficult for an opponent to precompute all the keys corresponding to a dictionary of passwords, or even the most likely keys. If the salt is 64 bits long, for instance, there will be as many as 264 keys for each password. It is unlikely that the same key will be selected twice. Again, if the salt is 64 bits long, the chance of “collision” between keys does not become significant until about 232 keys have been produced.

An iteration count has traditionally served the purpose of increasing the cost of producing keys from a password, thereby also increasing the difficulty of attack.

Of the two functions defined in [3], we chose PBKDF1, which employs a hash function, in this case, MD5.

PBKDF1 Algorithm (see [3])

*PBKDF1 (P, S, c, dkLen)*

*Options: Hash* underlying hash function

*Input: P* password, an octet string

*S* salt, an eight-octet string

*C* iteration count, a positive integer

*dkLen* intended length in octets of derived key, a positive integer, at most16 for MD2 or MD5 and 20 for SHA-1

*Output: DK* derived key, a dkLen-octet string

Steps:

1. If *dkLen* > 16 for MD2 and MD5, or *dkLen* >20 for SHA-1, output “derived key too long” and stop.
2. Apply the underlying hash function for c iterations to the concatenation of the password *P* and the salt *S*, then extract the first *dkLen* octets to produce a derived key *DK*: *T1 = Hash(P||S),* *T2 = Hash(T1), … Tc = Hash(Tc-1), DK = Tc <0..dkLen-1>.*
3. Output the derived key *DK*.

The Java™ Cryptography Extension (JCE) [4] provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.

JCE was previously an optional package (extension) to the Java™ 2 SDK, Standard Edition (J2SDK), versions 1.2.x and 1.3.x. JCE has now been integrated into the J2SDK, v 1.4. JCE provides an implementation of the MD5 with DES-CBC password-based encryption (PBE) algorithm defined in PKCS #5 together with “Secret-key factories” for bi-directional conversions between opaque DES, Triple DES and PBE key objects and transparent representations of their underlying key material.

Thus the implementation can be shown in a block diagram form as in Figure.

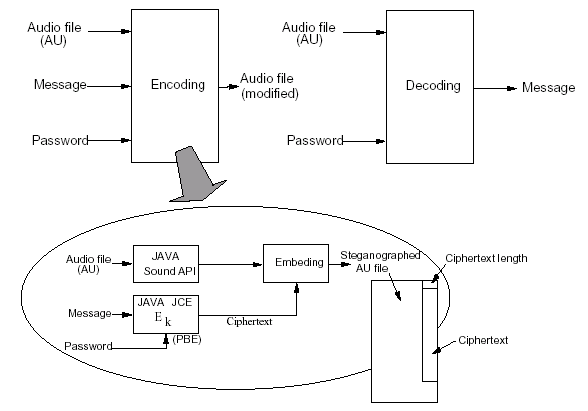


Fig.: Encoding and decoding modules