Abstract

This document describes various issues with PKCS #12 and PFX, and proposes solutions to those problems. It is intended to be a strawman for further discussion of password- based encryption algorithms as well as secure private key transport.

Fixing Private Key Transport

Peter Gutmann
pgut001@cs.auckland.ac.nz

Blake C. Ramsdell
blaker@deming.com

October 7, 1998

1 Introduction

Over the years a number of storage formats for private keys have been created by vendors to supplement the original PKCS #5/PKCS #8 formats, which were limited to the use of DES and MD5 to secure RSA keys. This has lead to a proliferation of incompatible and often insecure private key storage formats, culminating in the extremely complex PKCS #12 format with a confusion of incompatible object identifiers, algorithms, data content, and processing requirements. As a result of this confusion, implementations have been created which are both insecure (due to use of 40-bit keys and no real key preprocessing) and incompatible (the requirements don't provide enough information to implement compatible versions, leading to implementors having to reverse-engineer other implementations to assure compatibility). Another drawback of the format is the problem it has with data bloat, which makes it unusable with limited-storage-space media like smart cards.

These problems have highlighted a need for a simple, secure, compatible private key storage format. The design goals for this format are:

- Keep private keys (very) secure.
- Use existing technology as much as possible, introducing a minimum of new, untested algorithms and methods. This means it should be able to be deployed with a minimum amount of effort.
- Be precise enough to ensure that it's relatively easy to produce an interoperable implementation, and relatively difficult to produce an insecure implementation.

  Author's note: *This is normal IETF standards practice, the requirement was a reaction to the original PFX/PKCS #12 spec which didn't* *provide any indication of what you should and shouldn't do, leading to people choosing insecure options and parameters. One suggestion made was to ban RC4, because as a pure keystream generator it has a long history of being used in an insecure manner.*

- Open to compact encoding for use in limited-space key carriers such as smart cards.
- Keep it simple. The format is designed to keep a single private key secure, so that a successful attack on a key container won't compromise a whole raft of other data. In contrast a successful PKCS #12 attack will compromise more or less every piece of sensitive information held by a person because it's all collected in the same file.

2 Key Container Format

A public/private key pair consists of two sets of parameters, the public parameters with optional additional identification information such as the users name and email address (stored unencrypted) and the private parameters (generally stored using some form of protection mechanism such as encryption).

The key container is defined using the format given in PKCS #7 version 1.5 [RFC 2315] and the S/MIME Cryptographic Message Syntax [CMS]. The following object identifier identifies the secured private key content type:

```
  id-securedPrivateKey OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) internet(1)
        private(4) enterprise(1) dds(3029) cryptlib(32) 42}
```

Author's note: *This is a temporary OID grabbed from the DDS tree until an official OID is assigned.*

The secured private key content type shall have an ASN.1 type SecuredPrivateKey:

```
SecuredPrivateKey ::= SEQUENCE {
  version         Version,
  publicKey       PublicKeyInfo,
  privateKey      ContentInfo
  }
```

The fields of type SecuredPrivateKey have the following meanings:

version is the syntax version number; it shall be 0.

publicKey is the public components of the key and optional additional identification information for the key owner. The details of PublicKeyInfo are discussed in section 2.1.

privateKey is the (usually) secured private key, consisting of a content type identifier and the content itself. Details of the secured private key are discussed in section 2.2.

2.1 Public Key Components

The public key components are stored in unencrypted form. This is done because there is no real reason to encrypt them, because encrypting them with any stream cipher provides an attacker with a sizeable quantity of known plaintext, and because it should be possible for a user to access their public key without being required to enter a password. The public key information is represented in the type PublicKeyInfo:

```
PublicKeyInfo ::= CHOICE {
  publicKeyInfo          SubjectPublicKeyInfo,         -- Raw public key info
  certRequest     [ 0 ] EXPLICIT PKCS10CertRequest,   -- PKCS #10 cert.req.
  certificate     [ 1 ] EXPLICIT Certificate,         -- X.509 certificate
  certChain       [ 2 ] EXPLICIT PKCS7CertChain       -- PKCS #7 cert.chain
  }
```

In its simplest form this contains just the public key as SubjectPublicKeyInfo. Once a certification request is generated, the raw public key information may be replaced with the request, and finally when the key is certified either the certificate or the full certificate chain can be saved. Since the certificate-related PDU's are discrete entities, they are explicitly tagged to allow them to be read and written without having to change their tagging.

2.2 Private Key Components

The private key components may be stored in unencrypted or encrypted form, the encrypted form is recommended unless the data is protected by other means (for example tamperproof hardware). If stored unencrypted, the Data content-type is used. If stored encrypted, the EncryptedData content-type is used with the contentEncryptionAlgorithm specified in section 3. The content or encrypted content is the private key fields (without the accompanying public key fields) which correspond to the public key given in the PublicKeyInfo. For example for RSA keys this is:

```
RSAPrivateKey ::= SEQUENCE {
  privateExponent       INTEGER,        -- Private exponent d
  prime1                INTEGER,        -- Prime factor p of n
  prime2                INTEGER,        -- Prime factor q of n
  exponent1             INTEGER,        -- d mod (p-1)
  exponent2             INTEGER,        -- d mod (q-1)
  coefficient           INTEGER         -- CRT coefficient (q^-1) mod p
  }
```

For DSA and Elgamal keys this is:

```
DSAPrivateKey ::= SEQUENCE {
  x                     INTEGER         -- Private random integer
  }
```

The reason why only the private key fields are stored is to avoid redundancy (they're already specified in the PublicKeyInfo) and to eliminate the possibility of trivially recovering sizeable amounts of keystream using the known public fields at the start of the key.

Author's note: *ECC algorithms generally don't need to store a private key, they can reconstruct it at any point from the users password. In order to stop dictionary attacks, some sort of stored nonce which is mixed into the password-processing would be useful, but there's no need to actually store the private key. Presumably for this case the encrypted private key would just be an indicator that it's a nonce rather than a key, followed by something like an OCTET STRING SIZE(8).*

3 User Password Processing

There are a number of mechanisms available for transforming a user password into an encryption key. The PKCS #5 specification [PKCS #5] is limited to iterations of MD2 and MD5 for use with DES keys. The SSL v3 function [SSL v3], defined as:

$$\begin{aligned} key &= \mathbf{MD5}(password + \mathbf{SHA1}(\text{“A”} + password + salt)) \\ &\quad + \mathbf{MD5}(password + \mathbf{SHA1}(\text{“BB”} + password + salt)) \end{aligned}$$

$$
\begin{aligned}
&+ \mathbf{MD5}(\mathit{password} + \mathbf{SHA1}(\text{``CCC''} + \mathit{password} + \mathit{salt})) \\
&+ \ldots
\end{aligned}
$$

suffers from the fact that the input used for the SHA1 step varies by only a few bits from the input used for the previous step, and that it requires the use of two seperate and fixed hash functions to turn a password into a key. In addition the function as defined doesn't allow the processing to be iterated to defeat dictionary attacks.

The TLS function [TLS] extends the SSL v3 function and is defined as:

$$
\begin{aligned}
\mathit{key} \; = \; & \mathbf{HMAC}(\mathit{password}, A(1) + \mathit{salt}) \\
& + \mathbf{HMAC}(\mathit{password}, A(2) + \mathit{salt}) \\
& + \ldots
\end{aligned}
$$

where $A(0) = \mathit{salt}, A(1) = \mathbf{HMAC}(\mathit{password}, A(0)), \ldots$

(in fact the key is the XOR of this function applied via HMAC-MD5 and HMAC-SHA1, again requiring the use of two seperate and fixed algorithms). Another disadvantage of the use of HMAC's is that they limit the upper size of the password to 64 ASCII characters, or 32 or even 16 characters for the wider character sets, due to the requirement for truncating long keys by hashing them [RFC 2104]. As with the SSL v3 function, there is no provision for iterating the function to defect dictionary attacks.

The X9.42 key derivation function [X9.42] is defined specifically in terms of SHA-1 and is:

$$
\begin{aligned}
\mathit{key} \; = \; & \mathbf{SHA1}(\mathit{password} + 1) \\
& + \mathbf{SHA1}(\mathit{password} + 2) \\
& + \ldots
\end{aligned}
$$

This is probably the worst of the lot, using a fixed hash function, changing only a single bit of the input for each key block, injecting the tiny amount of changing data after the fixed password rather than before it (effectively stripping off a number of SHA-1 rounds which are always processing the same password data), and not being iterable.

These considerations suggests the following requirements for a user password processing function:

- Must be independant of the underlying hash function, and mustn't rely on the availability of multiple hash functions.

- Must be iterable to defeat dictionary attacks.
- If repeated hashing is used, must vary the input to successive iterations as much as possible to defeat related-key/input attacks.

A further useful design goal is to make the output dependant on the encryption algorithm the key is being generated for in order to make transfer of key-recovery attacks from one algorithm to another impossible. If the same key is used for a number of algorithms then an attacker who could recover the key for one algorithm could take advantage of this to attack it when used for other algorithms (for example recovering a DES key would also recover nearly half of an IDEA key). Making the key processing step output dependant on the encryption algorithm, mode, and configuration used means that a key derived from the same password for use with any other algorithm, mode, or configuration cannot be easily recovered.

These requirements suggest the following general design:

```
key[] = { 0 };
state = hash( algorithm, mode, parameters, salt, password );

for count = 1 to iterations
  for length = 1 to keyLength
    state = hash( state );
    key[ length ] ^= hash( state, password );
```

The initial state is dependant on all input parameters (the encryption algorithm, mode, and parameters, and salt and password). Then at each step of the processing the state variable acts as a PRNG which both ensures that the input to the hash used to generate the key changes by a number of bits equal to the hash function output at each step, and ensures that the user key hashing is serialized so that any form of parallelization or precomputation isn't possible. Finally, by XOR-ing the output of successive processing steps into the key, each iteration round contributes to the final key.

3.1 Password Processing Parameters

The input to the hash function used to generate the state variable is:

```
StateHashData ::= SEQUENCE {
  encryptionAlgorithm   AlgorithmIdentifier,
  salt                  OCTET STRING SIZE(8) OPTIONAL,
  password              UTF8String
  }
```

The fields of type StateHashData have the following meanings:

encryptionAlgorithm is the encryption algorithm, mode, and optional parameters which the key is to be generated for. Implementations MUST support 3DES-CBC.

Author's note: *Any others for MUST? It's possible to get export approval for single-purpose encryption like this so there's no need for crippled crypto here.*

salt is a 64-bit salt. This value may be omitted if there is a need to generate a key which remains constant with a given password.

password is the user password, encoded as a UTF8 string.

The input to the hash function used to generate the key is:

```
KeyHashData ::= SEQUENCE {
  state                 OCTET STRING,
  password              UTF8String
  }
```

state is the output of the hash function-based PRNG.

password is the user password, encoded as a UTF8 string.

3.2 Password Processing Algorithm Identifier

When used with the EncryptedData content-type, the contentEncryption-Algorithm is identified with:

```
  id-passwordBasedEncryption OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6)
    internet(1) private(4) enterprise(1) dds(3029) cryptlib(32) 43}
```

Author's note: *Another temporary OID.*
The corresponding parameters are:

```
PBEParameters ::= SEQUENCE {
  hashAlgorithm         AlgorithmIdentifier,
  encryptionAlgorithm   AlgorithmIdentifier,
  salt                  OCTET STRING SIZE(8),
  iterationCount        INTEGER (200...MAX_ITERATION)
  }
```

The fields of type PBEParameters have the following meanings:

hashAlgorithm is the hash algorithm used to process the password. Implementations MUST support SHA-1, and SHOULD support MD5 and RIPEMD-160.

encryptionAlgorithm is the algorithm to generate the key for and to encrypt the data with, and has the same meaning as in StateHashData.

salt has the same meaning as in StateHashData.

iterationCount is the number of iterations of hashing to perform as defined in section 3. For reasonable security it is recommended that at least 500 iterations be used, which takes less than a second on a typical workstation.

4 Test Vectors

Author's note: *Test vectors to be added later.*

References

[CMS] Housley, R., "Cryptographic Message Syntax" draft-ietf-smime-cms-xx.txt, SPYRUS, June 1998.

[PKCS #5] RSA Laboratories, "PKCS #5: Password-Based Encryption Standard", RSA Laboratories, November 1993.

[RFC 2104] Krawczyk, H., Bellare, M., Canetti, R., "HMAC: Keyed-Hashing for Message Authentication" RFC 2104, February 1997.

[RFC 2315] Kaliski, B., "PKCS #7: Cryptographic Message Syntax Version 1.5" RFC 2315, RSA Laboratories East, March 1988.

[SSL v3] Freier, A., Karlton, P., Kocher, P., "The SSL Protocol Version 3.0" draft-freier-ssl-version3-xx.txt (withdrawn), November 1996.

[TLS] Dierks, T., Allen, C., "The TLS Protocol Version 1.0" draft-ietf-tls-protocol-xx.txt, Consensus Development, November 1997.

[X9.42] “Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algorithms” ANSI draft X9.42, 1998.