Network Working Group
<draft-horowitz-key-derivation-02.txt>
Internet-Draft

M. Horowitz
Stonecast, Inc.
August, 1998

Key Derivation for Authentication, Integrity, and Privacy

Status of this Memo

Abstract

Recent advances in cryptography have made it desirable to use longer cryptographic keys, and to make more careful use of these keys. In particular, it is considered unwise by some cryptographers to use the same key for multiple purposes. Since most cryptographic-based systems perform a range of functions, such as authentication, key exchange, integrity, and encryption, it is desirable to use different cryptographic keys for these purposes.

This RFC does not define a particular protocol, but defines a set of cryptographic transformations for use with arbitrary network protocols and block cryptographic algorithm.

Deriving Keys

In order to use multiple keys for different functions, there are two possibilities:

- Each protocol ``key'' contains multiple cryptographic keys. The implementation would know how to break up the protocol ``key'' for use by the underlying cryptographic routines.

- The protocol ``key'' is used to derive the cryptographic keys.
  The implementation would perform this derivation before calling

the underlying cryptographic routines.

In the first solution, the system has the opportunity to provide separate keys for different functions. This has the advantage that if one of these keys is broken, the others remain secret. However, this comes at the cost of larger ``keys'' at the protocol layer. In addition, since these ``keys'' may be encrypted, compromising the cryptographic key which is used to encrypt them compromises all the component keys. Also, the not all ``keys'' are used for all possible functions. Some ``keys'', especially those derived from passwords, are generated from limited amounts of entropy. Wasting some of this entropy on cryptographic keys which are never used is unwise.

The second solution uses keys derived from a base key to perform cryptographic operations. By carefully specifying how this key is used, all of the advantages of the first solution can be kept, while eliminating some disadvantages. In particular, the base key must be used only for generating the derived keys, and this derivation must be non-invertible and entropy-preserving. Given these restrictions, compromise of one derived keys does not compromise the other subkeys. Attack of the base key is limited, since it is only used for derivation, and is not exposed to any user data.

Since the derived key has as much entropy as the base keys (if the cryptosystem is good), password-derived keys have the full benefit of all the entropy in the password.

To generate a derived key from a base key:

Derived Key = DK(Base Key, Well-Known Constant)

where

DK(Key, Constant) = k-truncate(E(Key, Constant))

In this construction, E(Key, Plaintext) is a block cipher, Constant is a well-known constant defined by the protocol, and k-truncate truncates its argument by taking the first k bits; here, k is the key size of E.

If the output of E is is shorter than k bits, then some entropy in the key will be lost. If the Constant is smaller than the block size of E, then it must be padded so it may be encrypted. If the Constant is larger than the block size, then it must be folded down to the block size to avoid chaining, which affects the distribution of entropy.

In any of these situations, a variation of the above construction is used, where the folded Constant is encrypted, and the resulting

output is fed back into the encryption as necessary (the | indicates concatentation):

```
K1 = E(Key, n-fold(Constant))
K2 = E(Key, K1)
```

```
K3 = E(Key, K2)
K4 = ...

DK(Key, Constant) = k-truncate(K1 | K2 | K3 | K4 ...)
```

n-fold is an algorithm which takes m input bits and ``stretches'' them to form n output bits with no loss of entropy, as described in [Blumenthal96]. In this document, n-fold is always used to produce n bits of output, where n is the block size of E.

If the size of the Constant is not equal to the block size of E, then the Constant must be n-folded to the block size of E. This string is used as input to E. If the block size of E is less than the key size, then the output from E is taken as input to a second invocation of E. This process is repeated until the number of bits accumulated is greater than or equal to the key size of E. When enough bits have been computed, the first k are taken as the derived key.

Since the derived key is the result of one or more encryptions in the base key, deriving the base key from the derived key is equivalent to determining the key from a very small number of plaintext/ciphertext pairs. Thus, this construction is as strong as the cryptosystem itself.

Deriving Keys from Passwords

When protecting information with a password or other user data, it is necessary to convert an arbitrary bit string into an encryption key. In addition, it is sometimes desirable that the transformation from password to key be difficult to reverse. A simple variation on the construction in the prior section can be used:

```
Key = DK(k-fold(Password), Well-Known Constant)
```

k-fold is same algorithm as n-fold, used to fold the Password into the same number of bits as the key of E.

The k-fold algorithm is reversible, so recovery of the k-fold output is equivalent to recovery of Password. However, recovering the k-fold output is difficult for the same reason recovering the base key from a derived key is difficult.

Traditionally, the transformation from plaintext to ciphertext, or vice versa, is determined by the cryptographic algorithm and the key. A simple way to think of derived keys is that the transformation is determined by the cryptographic algorithm, the constant, and the key.

For interoperability, the constants used to derive keys for different purposes must be specified in the protocol specification. Also, the endian order of the keys must be specified.

The constants must not be specified on the wire, or else an attacker who determined one derived key could provide the associated constant and spoof data using that derived key, rather than the one the protocol designer intended.

Determining which parts of a protocol require their own constants is an issue for the designer of protocol using derived keys.

Security Considerations

This entire document deals with security considerations relating to the use of cryptography in network protocols.

Appendix

This Appendix quotes the n-fold algorithm from [Blumenthal96]. It is provided here as a convenience to the implementor. Sample vectors are also included. It should be noted that the sample vector in Appendix B.2 of the original paper appears to be incorrect. Two independent implementations from the specification (one in C by the author, and another in Scheme by Bill Sommerfeld) agree on a value different from that in [Blumenthal96].

> We first define a primitive called n-folding, which takes a variable-length input block and produces a fixed-length output sequence. The intent is to give each input bit approximately equal weight in determining the value of each output bit. Note that whenever we need to treat a string of bytes as a number, the assumed representation is Big-Endian -- Most Significant Byte first.
>
> To n-fold a number X, replicate the input value to a length that is the least common multiple of n and the length of X. Before each repetition, the input is rotated to the right by 13 bit positions. The successive n-bit chunks are added together using 1's-complement addiiton (that is, with end-around carry) to yield a n-bit result....

The result is the n-fold of X. Here are some sample vectors, in hexadecimal. For convenience, the inputs are ASCII encodings of strings.

```
64-fold("012345") =
64-fold(303132333435) = be072631276b1955

56-fold("password") =
56-fold(70617373776f7264) = 78a07b6caf85fa
```

```
   64-fold("Rough Consensus, and Running Code") =
   64-fold(526f75676820436f6e73656e7375732c20616e642052756e
           6e696e6720436f6465) = bb6ed30870b7f0e0
```

```
   168-fold("password") =
   168-fold(70617373776f7264) = 59e4a8ca7c0385c3c37b3f6d2000247cb6e6bd5b3e

   192-fold("MASSACHVSETTS INSTITVTE OF TECHNOLOGY"
   192-fold(4d41535341434856534554545320494e5354495456544520
            4f4620544543484e4f4c4f4759) =
            db3b0d8f0b061e603282b308a50841229ad798fab9540c1b
```

Acknowledgements

I would like to thank Uri Blumenthal, Hugo Krawczyk, and Bill Sommerfeld for their contributions to this document.

References

[Blumenthal96] Blumenthal, U., "A Better Key Schedule for DES-Like Ciphers", Proceedings of PRAGOCRYPT '96, 1996.

Author's Address

Marc Horowitz
Stonecast, Inc.
108 Stow Road
Harvard, MA 01451

Phone: +1 978 456 9103
Email: marc@stonecast.net