Network Working Group
Request for Comments: 4757
Category: Informational

K. Jaganathan
L. Zhu
J. Brezak
Microsoft Corporation

December 2006

The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows

Status of This Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

IESG Note

This document documents the RC4 Kerberos encryption types first introduced in Microsoft Windows 2000. Since then, these encryption types have been implemented in a number of Kerberos implementations. The IETF Kerberos community supports publishing this specification as an informational document in order to describe this widely implemented technology. However, while these encryption types provide the operations necessary to implement the base Kerberos specification [RFC4120], they do not provide all the required operations in the Kerberos cryptography framework [RFC3961]. As a result, it is not generally possible to implement potential extensions to Kerberos using these encryption types. The Kerberos encryption type negotiation mechanism [RFC4537] provides one approach for using such extensions even when a Kerberos infrastructure uses long-term RC4 keys. Because this specification does not implement operations required by RFC 3961 and because of security concerns with the use of RC4 and MD4 discussed in Section 8, this specification is not appropriate for publication on the standards track.

Abstract

The Microsoft Windows 2000 implementation of Kerberos introduces a new encryption type based on the RC4 encryption algorithm and using an MD5 HMAC for checksum. This is offered as an alternative to using the existing DES-based encryption types.

The RC4-HMAC encryption types are used to ease upgrade of existing Windows NT environments, provide strong cryptography (128-bit key lengths), and provide exportable (meet United States government export restriction requirements) encryption. This document describes the implementation of those encryption types.

Table of Contents

1. Introduction

The Microsoft Windows 2000 implementation of Kerberos contains new encryption and checksum types for two reasons. First, for export reasons early in the development process, 56-bit DES encryption could not be exported, and, second, upon upgrade from Windows NT 4.0 to Windows 2000, accounts will not have the appropriate DES keying material to do the standard DES encryption. Furthermore, 3DES was not available for export when Windows 2000 was released, and there was a desire to use a single flavor of encryption in the product for both US and international products.

As a result, there are two new encryption types and one new checksum type introduced in Microsoft Windows 2000.

Note that these cryptosystems aren't intended to be complete, general-purpose Kerberos encryption or checksum systems as defined in [RFC3961]: there is no one-one mapping between the operations in this documents and the primitives described in [RFC3961].

1.1. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in [RFC2119].

2. Key Generation

On upgrade from existing Windows NT domains, the user accounts would not have a DES-based key available to enable the use of DES base encryption types specified in [RFC4120] and [RFC3961]. The key used for RC4-HMAC is the same as the existing Windows NT key (NT Password Hash) for compatibility reasons. Once the account password is changed, the DES-based keys are created and maintained. Once the DES keys are available, DES-based encryption types can be used with Kerberos.

The RC4-HMAC string to key function is defined as follows:

```
String2Key(password)

     K = MD4(UNICODE(password))
```

The RC4-HMAC keys are generated by using the Windows UNICODE version of the password. Each Windows UNICODE character is encoded in little-endian format of 2 octets each. Then an MD4 [RFC1320] hash operation is performed on just the UNICODE characters of the password (not including the terminating zero octets).

For an account with a password of "foo", this String2Key("foo") will return:

```
0xac, 0x8e, 0x65, 0x7f, 0x83, 0xdf, 0x82, 0xbe,
0xea, 0x5d, 0x43, 0xbd, 0xaf, 0x78, 0x00, 0xcc
```

3. Basic Operations

The MD5 HMAC function is defined in [RFC2104]. It is used in this encryption type for checksum operations. Refer to [RFC2104] for details on its operation. In this document, this function is referred to as HMAC(Key, Data) returning the checksum using the specified key on the data.

The basic MD5 hash operation is used in this encryption type and defined in [RFC1321]. In this document, this function is referred to as MD5(Data) returning the checksum of the data.

RC4 is a stream cipher licensed by RSA Data Security. In this document, the function is referred to as RC4(Key, Data) returning the encrypted data using the specified key on the data.

These encryption types use key derivation. With each message, the message type (T) is used as a component of the keying material. The following table summarizes the different key derivation values used in the various operations. Note that these differ from the key derivations used in other Kerberos encryption types. T = the message type, encoded as a little-endian four-byte integer.

1. AS-REQ PA-ENC-TIMESTAMP padata timestamp, encrypted with the client key (T=1)
2. AS-REP Ticket and TGS-REP Ticket (includes TGS session key or application session key), encrypted with the service key (T=2)
3. AS-REP encrypted part (includes TGS session key or application session key), encrypted with the client key (T=8)
4. TGS-REQ KDC-REQ-BODY AuthorizationData, encrypted with the TGS session key (T=4)
5. TGS-REQ KDC-REQ-BODY AuthorizationData, encrypted with the TGS authenticator subkey (T=5)
6. TGS-REQ PA-TGS-REQ padata AP-REQ Authenticator cksum, keyed with the TGS session key (T=6)
7. TGS-REQ PA-TGS-REQ padata AP-REQ Authenticator (includes TGS authenticator subkey), encrypted with the TGS session key T=7)
8. TGS-REP encrypted part (includes application session key), encrypted with the TGS session key (T=8)
9. TGS-REP encrypted part (includes application session key), encrypted with the TGS authenticator subkey (T=8)

10. AP-REQ Authenticator cksum, keyed with the application session key (T=10)
11. AP-REQ Authenticator (includes application authenticator subkey), encrypted with the application session key (T=11)
12. AP-REP encrypted part (includes application session subkey), encrypted with the application session key (T=12)
13. KRB-PRIV encrypted part, encrypted with a key chosen by the application. Also for data encrypted with GSS Wrap (T=13)
14. KRB-CRED encrypted part, encrypted with a key chosen by the application (T=14)
15. KRB-SAFE cksum, keyed with a key chosen by the application. Also for data signed in GSS MIC (T=15)

Relative to RFC-1964 key uses:

T = 0 in the generation of sequence number for the MIC token
T = 0 in the generation of sequence number for the WRAP token
T = 0 in the generation of encrypted data for the WRAPPED token

All strings in this document are ASCII unless otherwise specified. The lengths of ASCII-encoded character strings include the trailing terminator character (0). The concat(a,b,c,...) function will return the logical concatenation (left to right) of the values of the arguments. The nonce(n) function returns a pseudo-random number of "n" octets.

4. Checksum Types

There is one checksum type used in this encryption type. The Kerberos constant for this type is:

```
#define KERB_CHECKSUM_HMAC_MD5 (-138)
```

The function is defined as follows:

K = the Key
T = the message type, encoded as a little-endian four-byte integer

CHKSUM(K, T, data)

```
Ksign = HMAC(K, "signaturekey")  //includes zero octet at end
tmp = MD5(concat(T, data))
CHKSUM = HMAC(Ksign, tmp)
```

5. Encryption Types

There are two encryption types used in these encryption types. The Kerberos constants for these types are:

```
#define KERB_ETYPE_RC4_HMAC             23
#define KERB_ETYPE_RC4_HMAC_EXP         24
```

The basic encryption function is defined as follows:

T = the message type, encoded as a little-endian four-byte integer.

```
OCTET L40[14] = "fortybits";
```

The header field on the encrypted data in KDC messages is:

```
typedef struct _RC4_MDx_HEADER {
    OCTET Checksum[16];
    OCTET Confounder[8];
} RC4_MDx_HEADER, *PRC4_MDx_HEADER;


ENCRYPT (K, export, T, data)
{
    struct EDATA {
        struct HEADER {
                OCTET Checksum[16];
                OCTET Confounder[8];
        } Header;
        OCTET Data[0];
    } edata;

    if (export){
        *((DWORD *)(L40+10)) = T;
        K1 = HMAC(K, L40); // where the length of L40 in
                           // octets is 14
    }
    else
    {
        K1 = HMAC(K, &T); // where the length of T in octets
                          // is 4
    }
    memcpy (K2, K1, 16);
    if (export) memset (K1+7, 0xAB, 9);

    nonce (edata.Confounder, 8);
    memcpy (edata.Data, data);
```

```
            edata.Checksum = HMAC (K2, edata);
            K3 = HMAC (K1, edata.Checksum);

            RC4 (K3, edata.Confounder);
            RC4 (K3, data.Data);
        }

        DECRYPT (K, export, T, edata)
        {
            // edata looks like
            struct EDATA {
                struct HEADER {
                        OCTET Checksum[16];
                        OCTET Confounder[8];
                } Header;
                OCTET Data[0];
            } edata;

            if (export){
                *((DWORD *)(L40+10)) = T;
                HMAC (K, L40, 14, K1);
            }
            else
            {
                HMAC (K, &T, 4, K1);
            }
            memcpy (K2, K1, 16);
            if (export) memset (K1+7, 0xAB, 9);

            K3 = HMAC (K1, edata.Checksum);
            RC4 (K3, edata.Confounder);
            RC4 (K3, edata.Data);


            // verify generated and received checksums
          checksum = HMAC (K2, concat(edata.Confounder, edata.Data));
            if (checksum != edata.Checksum)
                printf("CHECKSUM ERROR  !!!!!!\n");
        }
```

The KDC message is encrypted using the ENCRYPT function not including the Checksum in the RC4_MDx_HEADER.

The character constant "fortybits" evolved from the time when a 40-bit key length was all that was exportable from the United States. It is now used to recognize that the key length is of "exportable" length. In this description, the key size is actually 56 bits.

The pseudo-random operation [RFC3961] for both enctypes above is defined as follows:

```
pseudo-random(K, S) = HMAC-SHA1(K, S)
```

where K is the protocol key and S is the input octet string. HMAC-SHA1 is defined in [RFC2104] and the output of HMAC-SHA1 is the 20-octet digest.

6. Key Strength Negotiation

A Kerberos client and server can negotiate over key length if they are using mutual authentication. If the client is unable to perform full-strength encryption, it may propose a key in the "subkey" field of the authenticator, using a weaker encryption type. The server must then either return the same key or suggest its own key in the subkey field of the AP reply message. The key used to encrypt data is derived from the key returned by the server. If the client is able to perform strong encryption but the server is not, it may propose a subkey in the AP reply without first being sent a subkey in the authenticator.

7. GSS-API Kerberos V5 Mechanism Type

7.1. Mechanism Specific Changes

The Generic Security Service Application Program Interface (GSS-API) per-message tokens also require new checksum and encryption types. The GSS-API per-message tokens are adapted to support these new encryption types. See [RFC1964] Section 1.2.2.

The only support quality of protection is:

```
#define GSS_KRB5_INTEG_C_QOP_DEFAULT    0x0
```

When using this RC4-based encryption type, the sequence number is always sent in big-endian rather than little-endian order.

The Windows 2000 implementation also defines new GSS-API flags in the initial token passed when initializing a security context. These flags are passed in the checksum field of the authenticator. See [RFC1964] Section 1.1.1.

GSS_C_DCE_STYLE - This flag was added for use with Microsoft’s implementation of Distributed Computing Environment Remote Procedure Call (DCE RPC), which initially expected three legs of authentication. Setting this flag causes an extra AP reply to be sent from the client back to the server after receiving the server’s

AP reply. In addition, the context negotiation tokens do not have GSS-API per-message tokens -- they are raw AP messages that do not include object identifiers.

```
#define GSS_C_DCE_STYLE                 0x1000
```

GSS_C_IDENTIFY_FLAG - This flag allows the client to indicate to the server that it should only allow the server application to identify the client by name and ID, but not to impersonate the client.

```
#define GSS_C_IDENTIFY_FLAG             0x2000
```

GSS_C_EXTENDED_ERROR_FLAG - Setting this flag indicates that the client wants to be informed of extended error information. In particular, Windows 2000 status codes may be returned in the data field of a Kerberos error message. This allows the client to understand a server failure more precisely. In addition, the server may return errors to the client that are normally handled at the application layer in the server, in order to let the client try to recover. After receiving an error message, the client may attempt to resubmit an AP request.

```
#define GSS_C_EXTENDED_ERROR_FLAG       0x4000
```

These flags are only used if a client is aware of these conventions when using the Security Support Provider Interface (SSPI) on the Windows platform; they are not generally used by default.

When NetBIOS addresses are used in the GSS-API, they are identified by the GSS_C_AF_NETBIOS value. This value is defined as:

```
#define GSS_C_AF_NETBIOS                0x14
```

NetBios addresses are 16-octet addresses typically composed of 1 to 15 characters, trailing blank (ASCII char 20) filled, with a 16th octet of 0x0.

7.2. GSS-API MIC Semantics

The GSS-API checksum type and algorithm are defined in Section 5. Only the first 8 octets of the checksum are used. The resulting checksum is stored in the SGN_CKSUM field. See [RFC1964] Section 1.2 for GSS_GetMIC() and GSS_Wrap(conf_flag=FALSE).

The GSS_GetMIC token has the following format:

Byte no	Name	Description
0..1	TOK_ID	Identification field. Tokens emitted by GSS_GetMIC() contain the hex value 01 01 in this field.
2..3	SGN_ALG	Integrity algorithm indicator. 11 00 - HMAC
4..7	Filler	Contains ff ff ff ff
8..15	SND_SEQ	Sequence number field.
16..23	SGN_CKSUM	Checksum of "to-be-signed data", calculated according to algorithm specified in SGN_ALG field.

The MIC mechanism used for GSS-MIC-based messages is as follows:

```
   GetMIC(Kss, direction, export, seq_num, data)
   {
           struct Token {
                  struct Header {
                         OCTET TOK_ID[2];
                         OCTET SGN_ALG[2];
                         OCTET Filler[4];
                    };
                  OCTET SND_SEQ[8];
                  OCTET SGN_CKSUM[8];
           } Token;


           Token.TOK_ID = 01 01;
           Token.SGN_SLG = 11 00;
           Token.Filler = ff ff ff ff;

           // Create the sequence number

           if (direction == sender_is_initiator)
           {
                  memset(Token.SEND_SEQ+4, 0xff, 4)
           }
           else if (direction == sender_is_acceptor)
           {
                  memset(Token.SEND_SEQ+4, 0, 4)
           }
           Token.SEND_SEQ[0] = (seq_num & 0xff000000) >> 24;
           Token.SEND_SEQ[1] = (seq_num & 0x00ff0000) >> 16;
           Token.SEND_SEQ[2] = (seq_num & 0x0000ff00) >> 8;
           Token.SEND_SEQ[3] = (seq_num & 0x000000ff);
```

```
                // Derive signing key from session key

                Ksign = HMAC(Kss, "signaturekey");
                                  // length includes terminating null

                // Generate checksum of message - SGN_CKSUM
                //   Key derivation salt = 15

                Sgn_Cksum = MD5((int32)15, Token.Header, data);

                // Save first 8 octets of HMAC Sgn_Cksum

                Sgn_Cksum = HMAC(Ksign, Sgn_Cksum);
                memcpy(Token.SGN_CKSUM, Sgn_Cksum, 8);

                // Encrypt the sequence number

                // Derive encryption key for the sequence number
                //   Key derivation salt = 0

                if (exportable)
                {
                        Kseq = HMAC(Kss, "fortybits", (int32)0);
                                     // len includes terminating null
                        memset(Kseq+7, 0xab, 7)
                }
                else
                {
                         Kseq = HMAC(Kss, (int32)0);
                }
                Kseq = HMAC(Kseq, Token.SGN_CKSUM);

                // Encrypt the sequence number

                RC4(Kseq, Token.SND_SEQ);
        }
```

7.3. GSS-API WRAP Semantics

There are two encryption keys for GSS-API message tokens, one that is 128 bits in strength and one that is 56 bits in strength as defined in Section 6.

All padding is rounded up to 1 byte. One byte is needed to say that there is 1 byte of padding. The DES-based mechanism type uses 8-byte padding. See [RFC1964] Section 1.2.2.3.

The RC4-HMAC GSS_Wrap() token has the following format:

Byte no	Name	Description
0..1	TOK_ID	Identification field. Tokens emitted by GSS_Wrap() contain the hex value 02 01 in this field.
2..3	SGN_ALG	Checksum algorithm indicator. 11 00 - HMAC
4..5	SEAL_ALG	ff ff - none 00 00 - DES-CBC 10 00 - RC4
6..7	Filler	Contains ff ff
8..15	SND_SEQ	Encrypted sequence number field.
16..23	SGN_CKSUM	Checksum of plaintext padded data, calculated according to algorithm specified in SGN_ALG field.
24..31	Confounder	Random confounder.
32..last	Data	Encrypted or plaintext padded data.

The encryption mechanism used for GSS-wrap-based messages is as follows:

```
WRAP(Kss, encrypt, direction, export, seq_num, data)
{
        struct Token {          // 32 octets
               struct Header {
                      OCTET TOK_ID[2];
                      OCTET SGN_ALG[2];
                      OCTET SEAL_ALG[2];
                      OCTET Filler[2];
               };
               OCTET SND_SEQ[8];
               OCTET SGN_CKSUM[8];
                 OCTET Confounder[8];
        } Token;


        Token.TOK_ID = 02 01;
        Token.SGN_SLG = 11 00;
        Token.SEAL_ALG = (no_encrypt)? ff ff : 10 00;
        Token.Filler = ff ff;

        // Create the sequence number

        if (direction == sender_is_initiator)
        {
```

```
        memset(&Token.SEND_SEQ[4], 0xff, 4)
}
else if (direction == sender_is_acceptor)
{
        memset(&Token.SEND_SEQ[4], 0, 4)
}
Token.SEND_SEQ[0] = (seq_num & 0xff000000) >> 24;
Token.SEND_SEQ[1] = (seq_num & 0x00ff0000) >> 16;
Token.SEND_SEQ[2] = (seq_num & 0x0000ff00) >> 8;
Token.SEND_SEQ[3] = (seq_num & 0x000000ff);

// Generate random confounder

nonce(&Token.Confounder, 8);

// Derive signing key from session key

Ksign = HMAC(Kss, "signaturekey");

// Generate checksum of message -
//  SGN_CKSUM + Token.Confounder
//   Key derivation salt = 15

Sgn_Cksum = MD5((int32)15, Token.Header,
                Token.Confounder);

// Derive encryption key for data
//   Key derivation salt = 0

for (i = 0; i < 16; i++) Klocal[i] = Kss[i] ^ 0xF0;
                                          // XOR
if (exportable)
{
        Kcrypt = HMAC(Klocal, "fortybits", (int32)0);
                    // len includes terminating null
        memset(Kcrypt+7, 0xab, 7);
}
else
{
        Kcrypt = HMAC(Klocal, (int32)0);
  }

// new encryption key salted with seq

Kcrypt = HMAC(Kcrypt, (int32)seq);
```

```
                // Encrypt confounder (if encrypting)

                if (encrypt)
                        RC4(Kcrypt, Token.Confounder);

                // Sum the data buffer

                Sgn_Cksum += MD5(data);         // Append to checksum

                // Encrypt the data (if encrypting)

                if (encrypt)
                        RC4(Kcrypt, data);

                // Save first 8 octets of HMAC Sgn_Cksum

                Sgn_Cksum = HMAC(Ksign, Sgn_Cksum);
                memcpy(Token.SGN_CKSUM, Sgn_Cksum, 8);

                // Derive encryption key for the sequence number
                //   Key derivation salt = 0

                if (exportable)
                {
                        Kseq = HMAC(Kss, "fortybits", (int32)0);
                                    // len includes terminating null
                        memset(Kseq+7, 0xab, 7)
                }
                else
                {
                        Kseq = HMAC(Kss, (int32)0);
                }
                Kseq = HMAC(Kseq, Token.SGN_CKSUM);

                // Encrypt the sequence number

                RC4(Kseq, Token.SND_SEQ);

                // Encrypted message = Token + Data
        }
```

The character constant "fortybits" evolved from the time when a 40-bit key length was all that was exportable from the United States. It is now used to recognize that the key length is of "exportable" length. In this description, the key size is actually 56 bits.

8. Security Considerations

Care must be taken in implementing these encryption types because they use a stream cipher. If a different IV is not used in each direction when using a session key, the encryption is weak. By using the sequence number as an IV, this is avoided.

There are two classes of attack on RC4 described in [MIRONOV]. Strong distinguishers distinguish an RC4 keystream from randomness at the start of the stream. Weak distinguishers can operate on any part of the keystream, and the best ones, described in [FMcG] and [MANTIN05], can exploit data from multiple, different keystreams. A consequence of these is that encrypting the same data (for instance, a password) sufficiently many times in separate RC4 keystreams can be sufficient to leak information to an adversary. The encryption types defined in this document defend against these by constructing a new keystream for every message. However, it is RECOMMENDED not to use the RC4 encryption types defined in this document for high-volume connections.

Weaknesses in MD4 [BOER91] were demonstrated by den Boer and Bosselaers in 1991. In August 2004, Xiaoyun Wang, et al., reported MD4 collisions generated using hand calculation [WANG04]. Implementations based on Wang's algorithm can find collisions in real time. However, the intended usage of MD4 described in this document does not rely on the collision-resistant property of MD4. Furthermore, MD4 is always used in the context of a keyed hash in this document. Although no evidence has suggested keyed MD4 hashes are vulnerable to collision-based attacks, no study has directly proved that the HMAC-MD4 is secure: the existing study simply assumed that the hash function used in HMAC is collision proof. It is thus RECOMMENDED not to use the RC4 encryption types defined in this document if alternative stronger encryption types, such as aes256-cts-hmac-sha1-96 [RFC3962], are available.

9. IANA Considerations

Section 5 of this document defines two Kerberos encryption types rc4-hmac (23) and rc4-hmac-exp (24). The Kerberos parameters registration page at <http://www.iana.org/assignments/kerberos-parameters> has been updated to reference this document for these two encryption types.

10. Acknowledgements

The authors wish to thank Sam Hartman, Ken Raeburn, and Qunli Li for their insightful comments.

11. References

11.1. Normative References

[RFC1320] Rivest, R., "The MD4 Message-Digest Algorithm", RFC 1320, April 1992.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.

[RFC3962] Raeburn, K., "Advanced Encryption Standard (AES) Encryption for Kerberos 5", RFC 3962, February 2005.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4537] Zhu, L., Leach, P., and K. Jaganathan, "Kerberos Cryptosystem Negotiation Extension", RFC 4537, June 2006.

11.2. Informative References

[BOER91] den Boer, B. and A. Bosselaers, "An Attack on the Last Two Rounds of MD4", Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology, pages: 194 - 203, 1991.

[FMcG] Fluhrer, S. and D. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator", Fast Software Encryption: 7th International Workshop, FSE 2000, April 2000, <http://www.mindspring.com/~dmcgrew/rc4-03.pdf>.

[MANTIN05] Mantin, I., "Predicting and Distinguishing Attacks on RC4 Keystream Generator", Advances in Cryptology -- EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, May 2005.

[MIRONOV] Mironov, I., "(Not So) Random Shuffles of RC4", Advances in Cryptology -- CRYPTO 2002: 22nd Annual International Cryptology Conference, August 2002, <http://eprint.iacr.org/2002/067.pdf>.

[WANG04] Wang, X., Lai, X., Feng, D., Chen, H., and X. Yu, "Cryptanalysis of Hash functions MD4 and RIPEMD", August 2004, <http://www.infosec.sdu.edu.cn/paper/md4-ripemd-attck.pdf>.

Authors' Addresses

Karthik Jaganathan
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
US

EMail: karthikj@microsoft.com

Larry Zhu
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
US

EMail: lzhu@microsoft.com

John Brezak
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
US

EMail: jbrezak@microsoft.com

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.