Network Working Group
Request for Comments: 2759
Category: Informational

G. Zorn
Microsoft Corporation

January 2000

Microsoft PPP CHAP Extensions, Version 2

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

Abstract

The Point-to-Point Protocol (PPP) [1] provides a standard method for transporting multi-protocol datagrams over point-to-point links. PPP defines an extensible Link Control Protocol and a family of Network Control Protocols (NCPs) for establishing and configuring different network-layer protocols.

This document describes version two of Microsoft's PPP CHAP dialect (MS-CHAP-V2). MS-CHAP-V2 is similar to, but incompatible with, MS-CHAP version one (MS-CHAP-V1, described in [9]). In particular, certain protocol fields have been deleted or reused but with different semantics. In addition, MS-CHAP-V2 features mutual authentication.

The algorithms used in the generation of various MS-CHAP-V2 protocol fields are described in section 8. Negotiation and hash generation examples are provided in section 9.

Specification of Requirements

In this document, the key words "MAY", "MUST, "MUST NOT", "optional", "recommended", "SHOULD", and "SHOULD NOT" are to be interpreted as described in [3].

Table of Contents

1. Introduction 3
2. LCP Configuration 3
3. Challenge Packet 3
4. Response Packet 4
5. Success Packet 4
6. Failure Packet 5
7. Change-Password Packet 6
8. Pseudocode 7
8.1. GenerateNTResponse() 7
8.2. ChallengeHash() 8
8.3. NtPasswordHash() 9
8.4. HashNtPasswordHash() 9
8.5. ChallengeResponse() 9
8.6. DesEncrypt() 10
8.7. GenerateAuthenticatorResponse() 10
8.8. CheckAuthenticatorResponse() 12
8.9. NewPasswordEncryptedWithOldNtPasswordHash() 12
8.10. EncryptPwBlockWithPasswordHash() 13
8.11. Rc4Encrypt() 13
8.12. OldNtPasswordHashEncryptedWithNewNtPasswordHash() 14
8.13. NtPasswordHashEncryptedWithBlock() 14
9. Examples 14
9.1. Negotiation Examples 14
9.1.1. Successful authentication 15
9.1.2. Authenticator authentication failure 15
9.1.3. Failed authentication with no retry allowed 15
9.1.4. Successful authentication after retry 15
9.1.5. Failed hack attack with 3 attempts allowed 15
9.1.6. Successful authentication with password change 16
9.1.7. Successful authentication with retry and password change. . 16
9.2. Hash Example 16
9.3. Example of DES Key Generation 17
10. Security Considerations 17
11. References 18
12. Acknowledgements 19
13. Author’s Address 19
14. Full Copyright Statement 20

1. Introduction

Where possible, MS-CHAP-V2 is consistent with both MS-CHAP-V1 and standard CHAP. Briefly, the differences between MS-CHAP-V2 and MS-CHAP-V1 are:

* MS-CHAP-V2 is enabled by negotiating CHAP Algorithm 0x81 in LCP option 3, Authentication Protocol.

* MS-CHAP-V2 provides mutual authentication between peers by piggybacking a peer challenge on the Response packet and an authenticator response on the Success packet.

* The calculation of the "Windows NT compatible challenge response" sub-field in the Response packet has been changed to include the peer challenge and the user name.

* In MS-CHAP-V1, the "LAN Manager compatible challenge response" sub-field was always sent in the Response packet. This field has been replaced in MS-CHAP-V2 by the Peer-Challenge field.

* The format of the Message field in the Failure packet has been changed.

* The Change Password (version 1) and Change Password (version 2) packets are no longer supported. They have been replaced with a single Change-Password packet.

2. LCP Configuration

The LCP configuration for MS-CHAP-V2 is identical to that for standard CHAP, except that the Algorithm field has value 0x81, rather than the MD5 value 0x05. PPP implementations which do not support MS-CHAP-V2, but correctly implement LCP Config-Rej, should have no problem dealing with this non-standard option.

3. Challenge Packet

The MS-CHAP-V2 Challenge packet is identical in format to the standard CHAP Challenge packet.

MS-CHAP-V2 authenticators send an 16-octet challenge Value field. Peers need not duplicate Microsoft’s algorithm for selecting the 16-octet value, but the standard guidelines on randomness [1,2,7] SHOULD be observed.

Microsoft authenticators do not currently provide information in the Name field. This may change in the future.

4. Response Packet

The MS-CHAP-V2 Response packet is identical in format to the standard CHAP Response packet. However, the Value field is sub-formatted differently as follows:

```
16 octets: Peer-Challenge
 8 octets: Reserved, must be zero
24 octets: NT-Response
 1 octet : Flags
```

The Peer-Challenge field is a 16-octet random number. As the name implies, it is generated by the peer and is used in the calculation of the NT-Response field, below. Peers need not duplicate Microsoft's algorithm for selecting the 16-octet value, but the standard guidelines on randomness [1,2,7] SHOULD be observed.

The NT-Response field is an encoded function of the password, the user name, the contents of the Peer-Challenge field and the received challenge as output by the routine GenerateNTResponse() (see section 8.1, below). The Windows NT password is a string of 0 to (theoretically) 256 case-sensitive Unicode [8] characters. Current versions of Windows NT limit passwords to 14 characters, mainly for compatibility reasons; this may change in the future. When computing the NT-Response field contents, only the user name is used, without any associated Windows NT domain name. This is true regardless of whether a Windows NT domain name is present in the Name field (see below).

The Flag field is reserved for future use and MUST be zero.

The Name field is a string of 0 to (theoretically) 256 case-sensitive ASCII characters which identifies the peer's user account name. The Windows NT domain name may prefix the user's account name (e.g. "BIGCO\johndoe" where "BIGCO" is a Windows NT domain containing the user account "johndoe"). If a domain is not provided, the backslash should also be omitted, (e.g. "johndoe").

5. Success Packet

The Success packet is identical in format to the standard CHAP Success packet. However, the Message field contains a 42-octet authenticator response string and a printable message. The format of the message field is illustrated below.

"S=<auth_string> M=<message>"

The <auth_string> quantity is a 20 octet number encoded in ASCII as 40 hexadecimal digits. The hexadecimal digits A-F (if present) MUST be uppercase. This number is derived from the challenge from the Challenge packet, the Peer-Challenge and NT-Response fields from the Response packet, and the peer password as output by the routine GenerateAuthenticatorResponse() (see section 8.7, below). The authenticating peer MUST verify the authenticator response when a Success packet is received. The method for verifying the authenticator is described in section 8.8, below. If the authenticator response is either missing or incorrect, the peer MUST end the session.

The <message> quantity is human-readable text in the appropriate charset and language [12].

6. Failure Packet

The Failure packet is identical in format to the standard CHAP Failure packet. There is, however, formatted text stored in the Message field which, contrary to the standard CHAP rules, does affect the operation of the protocol. The Message field format is:

"E=eeeeeeeeee R=r C=cccccccccccccccccccccccccccccccc V=vvvvvvvvvv M=<msg>"

where

The "eeeeeeeeee" is the ASCII representation of a decimal error code (need not be 10 digits) corresponding to one of those listed below, though implementations should deal with codes not on this list gracefully.

646 ERROR_RESTRICTED_LOGON_HOURS
647 ERROR_ACCT_DISABLED
648 ERROR_PASSWD_EXPIRED
649 ERROR_NO_DIALIN_PERMISSION
691 ERROR_AUTHENTICATION_FAILURE
709 ERROR_CHANGING_PASSWORD

The "r" is an ASCII flag set to '1' if a retry is allowed, and '0' if not. When the authenticator sets this flag to '1' it disables short timeouts, expecting the peer to prompt the user for new credentials and resubmit the response.

The "cccccccccccccccccccccccccccccccc" is the ASCII representation of a hexadecimal challenge value. This field MUST be exactly 32 octets long and MUST be present.

The "vvvvvvvvvv" is the ASCII representation of a decimal version code (need not be 10 digits) indicating the password changing protocol version supported on the server. For MS-CHAP-V2, this value SHOULD always be 3.

<msg> is human-readable text in the appropriate charset and language [12].

7. Change-Password Packet

The Change-Password packet does not appear in either standard CHAP or MS-CHAP-V1. It allows the peer to change the password on the account specified in the preceding Response packet. The Change-Password packet should be sent only if the authenticator reports ERROR_PASSWD_EXPIRED (E=648) in the Message field of the Failure packet.

This packet type is supported by recent versions of Windows NT 4.0, Windows 95 and Windows 98. It is not supported by Windows NT 3.5, Windows NT 3.51, or early versions of Windows NT 4.0, Windows 95 and Windows 98.

The format of this packet is as follows:

```
     1 octet  : Code
     1 octet  : Identifier
     2 octets : Length
   516 octets : Encrypted-Password
    16 octets : Encrypted-Hash
    16 octets : Peer-Challenge
     8 octets : Reserved
    24 octets : NT-Response
     2-octet  : Flags
```

Code
7

Identifier
The Identifier field is one octet and aids in matching requests and replies. The value is the Identifier of the received Failure packet to which this packet responds plus 1.

Length
586

Encrypted-Password
This field contains the PWBLOCK form of the new Windows NT password encrypted with the old Windows NT password hash, as output by the NewPasswordEncryptedWithOldNtPasswordHash() routine (see section 8.9, below).

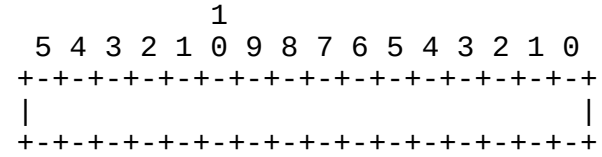
Encrypted-Hash
This field contains the old Windows NT password hash encrypted with the new Windows NT password hash, as output by the OldNtPasswordHashEncryptedWithNewNtPasswordHash() routine (see section 8.12, below).

Peer-Challenge
A 16-octet random quantity, as described in the Response packet description.

Reserved
8 octets, must be zero.

NT-Response
The NT-Response field (as described in the Response packet description), but calculated on the new password and the challenge received in the Failure packet.

Flags
This field is two octets in length. It is a bit field of option flags where 0 is the least significant bit of the 16-bit quantity. The format of this field is illustrated in the following diagram:



Bits 0-15
Reserved, always clear (0).

8. Pseudocode

The routines mentioned in the text above are described in pseudocode in the following sections.

8.1. GenerateNTResponse()

```
GenerateNTResponse(
IN  16-octet              AuthenticatorChallenge,
IN  16-octet              PeerChallenge,
```

```
   IN  0-to-256-char         UserName,

   IN  0-to-256-unicode-char Password,
   OUT 24-octet              Response )
   {
      8-octet  Challenge
      16-octet PasswordHash

      ChallengeHash( PeerChallenge, AuthenticatorChallenge, UserName,
                     giving Challenge)

      NtPasswordHash( Password, giving PasswordHash )
      ChallengeResponse( Challenge, PasswordHash, giving Response )
   }
```

8.2. ChallengeHash()

```
   ChallengeHash(
   IN 16-octet               PeerChallenge,
   IN 16-octet               AuthenticatorChallenge,
   IN  0-to-256-char         UserName,
   OUT 8-octet               Challenge
   {

      /*
       * SHAInit(), SHAUpdate() and SHAFinal() functions are an
       * implementation of Secure Hash Algorithm (SHA-1) [11]. These are
       * available in public domain or can be licensed from
       * RSA Data Security, Inc.
       */

      SHAInit(Context)
      SHAUpdate(Context, PeerChallenge, 16)
      SHAUpdate(Context, AuthenticatorChallenge, 16)

      /*
       * Only the user name (as presented by the peer and
       * excluding any prepended domain name)
       * is used as input to SHAUpdate().
       */

      SHAUpdate(Context, UserName, strlen(Username))
      SHAFinal(Context, Digest)
      memcpy(Challenge, Digest, 8)
   }
```

8.3. NtPasswordHash()

```
NtPasswordHash(
IN  0-to-256-unicode-char Password,
OUT 16-octet              PasswordHash )
{
   /*
    * Use the MD4 algorithm [5] to irreversibly hash Password
    * into PasswordHash.  Only the password is hashed without
    * including any terminating 0.
    */
}
```

8.4. HashNtPasswordHash()

```
HashNtPasswordHash(
IN  16-octet PasswordHash,
OUT 16-octet PasswordHashHash )
{
   /*
    * Use the MD4 algorithm [5] to irreversibly hash
    * PasswordHash into PasswordHashHash.
    */
}
```

8.5. ChallengeResponse()

```
ChallengeResponse(
IN  8-octet  Challenge,
IN  16-octet PasswordHash,
OUT 24-octet Response )
{
   Set ZPasswordHash to PasswordHash zero-padded to 21 octets

   DesEncrypt( Challenge,
               1st 7-octets of ZPasswordHash,
               giving 1st 8-octets of Response )

   DesEncrypt( Challenge,
               2nd 7-octets of ZPasswordHash,
               giving 2nd 8-octets of Response )

   DesEncrypt( Challenge,
               3rd 7-octets of ZPasswordHash,
               giving 3rd 8-octets of Response )
}
```

8.6. DesEncrypt()

```
DesEncrypt(
IN  8-octet Clear,
IN  7-octet Key,
OUT 8-octet Cypher )
{
   /*
    * Use the DES encryption algorithm [4] in ECB mode [10]
    * to encrypt Clear into Cypher such that Cypher can
    * only be decrypted back to Clear by providing Key.
    * Note that the DES algorithm takes as input a 64-bit
    * stream where the 8th, 16th, 24th, etc.  bits are
    * parity bits ignored by the encrypting algorithm.
    * Unless you write your own DES to accept 56-bit input
    * without parity, you will need to insert the parity bits
    * yourself.
    */
}
```

8.7. GenerateAuthenticatorResponse()

```
GenerateAuthenticatorResponse(
IN  0-to-256-unicode-char Password,
IN  24-octet              NT-Response,
IN  16-octet              PeerChallenge,
IN  16-octet              AuthenticatorChallenge,
IN  0-to-256-char         UserName,
OUT 42-octet              AuthenticatorResponse )
{
   16-octet             PasswordHash
   16-octet             PasswordHashHash
   8-octet              Challenge

   /*
    * "Magic" constants used in response generation
    */

   Magic1[39] =
      {0x4D, 0x61, 0x67, 0x69, 0x63, 0x20, 0x73, 0x65, 0x72, 0x76,
       0x65, 0x72, 0x20, 0x74, 0x6F, 0x20, 0x63, 0x6C, 0x69, 0x65,
       0x6E, 0x74, 0x20, 0x73, 0x69, 0x67, 0x6E, 0x69, 0x6E, 0x67,
       0x20, 0x63, 0x6F, 0x6E, 0x73, 0x74, 0x61, 0x6E, 0x74};
```

```
      Magic2[41] =
         {0x50, 0x61, 0x64, 0x20, 0x74, 0x6F, 0x20, 0x6D, 0x61, 0x6B,
          0x65, 0x20, 0x69, 0x74, 0x20, 0x64, 0x6F, 0x20, 0x6D, 0x6F,
          0x72, 0x65, 0x20, 0x74, 0x68, 0x61, 0x6E, 0x20, 0x6F, 0x6E,
          0x65, 0x20, 0x69, 0x74, 0x65, 0x72, 0x61, 0x74, 0x69, 0x6F,
          0x6E};

      /*
       * Hash the password with MD4
       */

      NtPasswordHash( Password, giving PasswordHash )

      /*
       * Now hash the hash
       */

      HashNtPasswordHash( PasswordHash, giving PasswordHashHash)

      SHAInit(Context)
      SHAUpdate(Context, PasswordHashHash, 16)
      SHAUpdate(Context, NTResponse, 24)
      SHAUpdate(Context, Magic1, 39)
      SHAFinal(Context, Digest)

      ChallengeHash( PeerChallenge, AuthenticatorChallenge, UserName,
                     giving Challenge)

      SHAInit(Context)
      SHAUpdate(Context, Digest, 20)
      SHAUpdate(Context, Challenge, 8)
      SHAUpdate(Context, Magic2, 41)
      SHAFinal(Context, Digest)

      /*
       * Encode the value of 'Digest' as "S=" followed by
       * 40 ASCII hexadecimal digits and return it in
       * AuthenticatorResponse.
       * For example,
       *   "S=0123456789ABCDEF0123456789ABCDEF01234567"
       */

   }
```

8.8. CheckAuthenticatorResponse()

```
CheckAuthenticatorResponse(
IN  0-to-256-unicode-char Password,
IN  24-octet              NtResponse,
IN  16-octet              PeerChallenge,
IN  16-octet              AuthenticatorChallenge,
IN  0-to-256-char         UserName,
IN  42-octet              ReceivedResponse,
OUT Boolean               ResponseOK )
{

   20-octet MyResponse

   set ResponseOK = FALSE
   GenerateAuthenticatorResponse( Password, NtResponse, PeerChallenge,
                                  AuthenticatorChallenge, UserName,
                                  giving MyResponse)

   if (MyResponse = ReceivedResponse) then set ResponseOK = TRUE
   return ResponseOK
}
```

8.9. NewPasswordEncryptedWithOldNtPasswordHash()

```
datatype-PWBLOCK
{
   256-unicode-char Password
   4-octets         PasswordLength
}

NewPasswordEncryptedWithOldNtPasswordHash(
IN  0-to-256-unicode-char NewPassword,
IN  0-to-256-unicode-char OldPassword,
OUT datatype-PWBLOCK      EncryptedPwBlock )
{
   NtPasswordHash( OldPassword, giving PasswordHash )

   EncryptPwBlockWithPasswordHash( NewPassword,
                                   PasswordHash,
                                   giving EncryptedPwBlock )
}
```

8.10. EncryptPwBlockWithPasswordHash()

```
EncryptPwBlockWithPasswordHash(
IN  0-to-256-unicode-char Password,
IN  16-octet              PasswordHash,
OUT datatype-PWBLOCK      PwBlock )
{

   Fill ClearPwBlock with random octet values

      PwSize = lstrlenW( Password ) * sizeof( unicode-char )
      PwOffset = sizeof( ClearPwBlock.Password ) - PwSize
      Move PwSize octets to (ClearPwBlock.Password + PwOffset ) from
Password
      ClearPwBlock.PasswordLength = PwSize
      Rc4Encrypt( ClearPwBlock,
                  sizeof( ClearPwBlock ),
                  PasswordHash,
                  sizeof( PasswordHash ),
                  giving PwBlock )
   }
```

8.11. Rc4Encrypt()

```
Rc4Encrypt(
IN  x-octet Clear,
IN  integer ClearLength,
IN  y-octet Key,
IN  integer KeyLength,
OUT x-octet Cypher )
{
   /*
    * Use the RC4 encryption algorithm [6] to encrypt Clear of
    * length ClearLength octets into a Cypher of the same length
    * such that the Cypher can only be decrypted back to Clear
    * by providing a Key of length KeyLength octets.
    */
}
```

8.12. OldNtPasswordHashEncryptedWithNewNtPasswordHash()

```
OldNtPasswordHashEncryptedWithNewNtPasswordHash(
IN  0-to-256-unicode-char NewPassword,
IN  0-to-256-unicode-char OldPassword,
OUT 16-octet              EncryptedPasswordHash )
{
   NtPasswordHash( OldPassword, giving OldPasswordHash )
   NtPasswordHash( NewPassword, giving NewPasswordHash )
   NtPasswordHashEncryptedWithBlock( OldPasswordHash,
                                     NewPasswordHash,
                                     giving EncryptedPasswordHash )
}
```

8.13. NtPasswordHashEncryptedWithBlock()

```
NtPasswordHashEncryptedWithBlock(
IN  16-octet PasswordHash,
IN  16-octet Block,
OUT 16-octet Cypher )
{
   DesEncrypt( 1st 8-octets PasswordHash,
               1st 7-octets Block,
               giving 1st 8-octets Cypher )

   DesEncrypt( 2nd 8-octets PasswordHash,
               2nd 7-octets Block,
               giving 2nd 8-octets Cypher )
}
```

9. Examples

The following sections include protocol negotiation and hash generation examples.

9.1. Negotiation Examples

Here are some examples of typical negotiations. The peer is on the left and the authenticator is on the right.

The packet sequence ID is incremented on each authentication retry response and on the change password response. All cases where the packet sequence ID is updated are noted below.

Response retry is never allowed after Change Password. Change Password may occur after response retry.

9.1.1. Successful authentication

```
                        <- Authenticator Challenge
       Peer Response/Challenge ->
                        <- Success/Authenticator Response

   (Authenticator Response verification succeeds, call continues)
```

9.1.2. Authenticator authentication failure

```
                        <- Authenticator Challenge
       Peer Response/Challenge ->
                        <- Success/Authenticator Response

   (Authenticator Response verification fails, peer disconnects)
```

9.1.3. Failed authentication with no retry allowed

```
                        <- Authenticator Challenge
       Peer Response/Challenge ->
                        <- Failure (E=691 R=0)

   (Authenticator disconnects)
```

9.1.4. Successful authentication after retry

```
                        <- Authenticator Challenge
       Peer Response/Challenge ->
                        <- Failure (E=691 R=1), disable short timeout
       Response (++ID) to challenge in failure message ->
                        <- Success/Authenticator Response

   (Authenticator Response verification succeeds, call continues)
```

9.1.5. Failed hack attack with 3 attempts allowed

```
                        <- Authenticator Challenge
       Peer Response/Challenge ->
                        <- Failure (E=691 R=1), disable short timeout
       Response (++ID) to challenge in Failure message ->
                        <- Failure (E=691 R=1), disable short timeout
       Response (++ID) to challenge in Failure message ->
                        <- Failure (E=691 R=0)
```

9.1.6. Successful authentication with password change

```
                          <- Authenticator Challenge
       Peer Response/Challenge ->
                          <- Failure (E=648 R=0 V=3), disable short
   timeout
       ChangePassword (++ID) to challenge in Failure message ->
                          <- Success/Authenticator Response

   (Authenticator Response verification succeeds, call continues)
```

9.1.7. Successful authentication with retry and password change

```
                          <- Authenticator Challenge
       Peer Response/Challenge ->
                          <- Failure (E=691 R=1), disable short timeout
       Response (++ID) to first challenge+23 ->
                          <- Failure (E=648 R=0 V=2), disable short
   timeout
       ChangePassword (++ID) to first challenge+23 ->
                          <- Success/Authenticator Response

   (Authenticator Response verification succeeds, call continues)
```

9.2. Hash Example

Intermediate values for user name "User" and password "clientPass". All numeric values are hexadecimal.

```
0-to-256-char UserName:
55 73 65 72

0-to-256-unicode-char Password:
63 00 6C 00 69 00 65 00 6E 00 74 00 50 00 61 00 73 00 73 00

16-octet AuthenticatorChallenge:
5B 5D 7C 7D 7B 3F 2F 3E 3C 2C 60 21 32 26 26 28

16-octet PeerChallenge:
21 40 23 24 25 5E 26 2A 28 29 5F 2B 3A 33 7C 7E

8-octet Challenge:
D0 2E 43 86 BC E9 12 26

16-octet PasswordHash:
44 EB BA 8D 53 12 B8 D6 11 47 44 11 F5 69 89 AE
```

24 octet NT-Response:
82 30 9E CD 8D 70 8B 5E A0 8F AA 39 81 CD 83 54 42 33 11 4A 3D 85 D6 DF

16-octet PasswordHashHash:
41 C0 0C 58 4B D2 D9 1C 40 17 A2 A1 2F A5 9F 3F

42-octet AuthenticatorResponse:
"S=407A5589115FD0D6209F510FE9C04566932CDA56"

9.3. Example of DES Key Generation

DES uses 56-bit keys, expanded to 64 bits by the insertion of parity bits. After the parity of the key has been fixed, every eighth bit is a parity bit and the number of bits that are set (1) in each octet is odd; i.e., odd parity. Note that many DES engines do not check parity, however, simply stripping the parity bits. The following example illustrates the values resulting from the use of the password "MyPw" to generate a pair of DES keys (e.g., for use in the NtPasswordHashEncryptedWithBlock() described in section 8.13).

0-to-256-unicode-char Password:
4D 79 50 77

16-octet PasswordHash:
FC 15 6A F7 ED CD 6C 0E DD E3 33 7D 42 7F 4E AC

First "raw" DES key (initial 7 octets of password hash):
FC 15 6A F7 ED CD 6C

First parity-corrected DES key (eight octets):
FD 0B 5B 5E 7F 6E 34 D9

Second "raw" DES key (second 7 octets of password hash)
0E DD E3 33 7D 42 7F

Second parity-corrected DES key (eight octets):
0E 6E 79 67 37 EA 08 FE

10. Security Considerations

As an implementation detail, the authenticator SHOULD limit the number of password retries allowed to make brute-force password guessing attacks more difficult.

11. References

[1] Simpson, W., "The Point-to-Point Protocol (PPP)", STD 51, RFC 1661, July 1994.

[2] Simpson, W., "PPP Challenge Handshake Authentication Protocol (CHAP)", RFC 1994, August 1996.

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

[4] "Data Encryption Standard (DES)", Federal Information Processing Standard Publication 46-2, National Institute of Standards and Technology, December 1993.

[5] Rivest, R., "MD4 Message Digest Algorithm", RFC 1320, April 1992.

[6] RC4 is a proprietary encryption algorithm available under license from RSA Data Security Inc. For licensing information, contact:

RSA Data Security, Inc.
100 Marine Parkway
Redwood City, CA 94065-1031

[7] Eastlake, D., Crocker, S. and J. Schiller, "Randomness Recommendations for Security", RFC 1750, December 1994.

[8] "The Unicode Standard, Version 2.0", The Unicode Consortium, Addison-Wesley, 1996. ISBN 0-201-48345-9.

[9] Zorn, G. and Cobb, S., "Microsoft PPP CHAP Extensions", RFC 2433, October 1998.

[10] "DES Modes of Operation", Federal Information Processing Standards Publication 81, National Institute of Standards and Technology, December 1980.

[11] "Secure Hash Standard", Federal Information Processing Standards Publication 180-1, National Institute of Standards and Technology, April 1995.

[12] Zorn, G., "PPP LCP Internationalization Configuration Option", RFC 2484, January 1999.

12. Acknowledgements

Thanks (in no particular order) to Bruce Johnson, Tony Bell, Paul Leach, Terence Spies, Dan Simon, Narendra Gidwani, Gurdeep Singh Pall, Jody Terrill, Brad Robel-Forrest, and Joe Davies for useful suggestions and feedback.

13. Author's Address

Questions about this memo can also be directed to:

Glen Zorn
Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052

Phone: +1 425 703 1559
Fax: +1 425 936 7329
EMail: gwz@acm.org

14. Full Copyright Statement

Acknowledgement

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