Network Working Group
Internet-Draft
Category: Informational
<draft-ietf-pppext-mschapv2-keys-02.txt>

G. Zorn
Microsoft Corporation

November 1998

Deriving MPPE Keys From MS-CHAP V2 Credentials

1. Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as ``work in progress''.

To learn the current status of any Internet-Draft, please check the ``1id-abstracts.txt'' listing contained in the Internet-Drafts Shadow Directories on ftp.ietf.org (US East Coast), nic.nordu.net (Europe), ftp.isi.edu (US West Coast), or munnari.oz.au (Pacific Rim).

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. The distribution of this memo is unlimited. It is filed as <draft-ietf-pppext-mschapv2-keys-02.txt> and expires May 15, 1999. Please send comments to the PPP Extensions Working Group mailing list (ietf-ppp@merit.edu) or to the author (gwz@acm.org).

2. Abstract

The Point-to-Point Protocol (PPP) [1] provides a standard method for transporting multi-protocol datagrams over point-to-point links.

The PPP Compression Control Protocol [2] provides a method to negotiate and utilize compression protocols over PPP encapsulated links.

Version 2 of the Microsoft Challenge-Handshake Authentication Protocol (MS-CHAP-2) [3] is a Microsoft-proprietary PPP authentication protocol, providing the functionality to which LAN-based users are accustomed while integrating the encryption and hashing algorithms used on Windows networks.

Microsoft Point to Point Encryption (MPPE) [4] is a means of

representing PPP packets in an encrypted form. MPPE uses the RSA RC4 [5] algorithm to provide data confidentiality. The length of the session key to be used for initializing encryption tables can be negotiated. MPPE currently supports 40-bit and 128-bit session keys. MPPE session keys are changed frequently; the exact frequency depends upon the options negotiated, but may be every packet. MPPE is negotiated within option 18 [6] in the Compression Control Protocol.

This document describes the method used to derive the initial MPPE session keys from MS-CHAP-2 credentials. The algorithm used to change session keys during a session is described in [4].

3. 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 [7].

4. Deriving Session Keys from MS-CHAP-2 Credentials

The following sections detail the methods used to derive initial session keys from MS-CHAP-2 credentials. Both 40- and 128-bit keys are derived using the same algorithm from the authenticating peer's Windows NT password. The only difference is in the length of the keys and their effective strength: 40-bit keys are 8 octets in length, while 128-bit keys are 16 octets long. Separate keys are derived for the send and receive directions of the session.

Implementation Note

> The initial session keys in both directions are derived from the credentials of the peer that initiated the call and the challenges used are those from the first authentication. This is true as well for each link in a multilink bundle. In the multi-chassis multilink case, implementations are responsible for ensuring that the correct keys are generated on all participating machines.

4.1. Generating 40-bit Session Keys

When used in conjunction with MS-CHAP-2 authentication, the initial MPPE session keys are derived from the peer's Windows NT password.

The first step is to obfuscate the peer's password using NtPasswordHash() function as described in [3].

```
NtPasswordHash(Password, PasswordHash)
```

The first 16 octets of the result are then hashed again using the MD4 algorithm.

```
PasswordHashHash = md4(PasswordHash)
```

The first 16 octets of this second hash are used together with the NT-Response field from the MS-CHAP-2 Response packet [3] as the basis for the master session key:

```
GetMasterKey(PasswordHashHash, NtResponse, MasterKey)
```

Once the master key has been generated, it is used to derive two 40-bit session keys, one for sending and one for receiving:

```
GetAsymmetricStartKey(MasterKey, MasterSendKey, 8, TRUE, TRUE)
GetAsymmetricStartKey(MasterKey, MasterReceiveKey, 8, FALSE, TRUE)
```

The master session keys are never used to encrypt or decrypt data; they are only used in the derivation of transient session keys. The initial transient session keys are obtained by calling the function Get-NewKeyFromSHA() (described in [4]):

```
GetNewKeyFromSHA(MasterSendKey, MasterSendKey, 8, SendSessionKey)
GetNewKeyFromSHA(MasterReceiveKey, MasterReceiveKey, 8, ReceiveSessionKey)
```

Next, the effective strength of both keys is reduced by setting the first three octets to known constants:

```
SendSessionKey[0] = ReceiveSessionKey[0] = 0xD1
SendSessionKey[1] = ReceiveSessionKey[1] = 0x26
SendSessionKey[2] = ReceiveSessionKey[2] = 0x9E
```

Finally, the RC4 tables are initialized using the new session keys:

```
rc4_key(SendRC4key, 8, SendSessionKey)
rc4_key(ReceiveRC4key, 8, ReceiveSessionKey)
```

4.2. Generating 128-bit Session Keys

When used in conjunction with MS-CHAP-2 authentication, the initial MPPE session keys are derived from the peer's Windows NT password.

The first step is to obfuscate the peer's password using NtPassword-Hash() function as described in [3].

```
NtPasswordHash(Password, PasswordHash)
```

The first 16 octets of the result are then hashed again using the MD4 algorithm.

```
PasswordHashHash = md4(PasswordHash)
```

The first 16 octets of this second hash are used together with the NT-Response field from the MS-CHAP-2 Response packet [3] as the basis for the master session key:

```
GetMasterKey(PasswordHashHash, NtResponse, MasterKey)
```

Once the master key has been generated, it is used to derive two 128-bit master session keys, one for sending and one for receiving:

```
GetAsymmetricStartKey(MasterKey, MasterSendKey, 16, TRUE, TRUE)
GetAsymmetricStartKey(MasterKey, MasterReceiveKey, 16, FALSE, TRUE)
```

The master session keys are never used to encrypt or decrypt data; they are only used in the derivation of transient session keys. The initial transient session keys are obtained by calling the function Get-NewKeyFromSHA() (described in [4]):

```
GetNewKeyFromSHA(MasterSendKey, MasterSendKey, 16, SendSessionKey)
GetNewKeyFromSHA(MasterReceiveKey, MasterReceiveKey, 16, ReceiveSessionKey)
```

Finally, the RC4 tables are initialized using the new session keys:

```
rc4_key(SendRC4key, 16, SendSessionKey)
rc4_key(ReceiveRC4key, 16, ReceiveSessionKey)
```

4.3. Key Derivation Functions

The following procedures are used to derive the session key.

```
/*
 * Pads used in key derivation
 */

SHSpad1[40] =
   {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

SHSpad2[40] =
```

```
      {0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2,
       0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2,
       0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2,
       0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2, 0xF2};

   /*
    * "Magic" constants used in key derivations
    */

   Magic1[27] =
      {0x54, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x74,
       0x68, 0x65, 0x20, 0x4D, 0x50, 0x50, 0x45, 0x20, 0x4D,
       0x61, 0x73, 0x74, 0x65, 0x72, 0x20, 0x4B, 0x65, 0x79};

   Magic2[84] =
      {0x4F, 0x6E, 0x20, 0x74, 0x68, 0x65, 0x20, 0x63, 0x6C, 0x69,
       0x65, 0x6E, 0x74, 0x20, 0x73, 0x69, 0x64, 0x65, 0x2C, 0x20,
       0x74, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x74, 0x68,
       0x65, 0x20, 0x73, 0x65, 0x6E, 0x64, 0x20, 0x6B, 0x65, 0x79,
       0x3B, 0x20, 0x6F, 0x6E, 0x20, 0x74, 0x68, 0x65, 0x20, 0x73,
       0x65, 0x72, 0x76, 0x65, 0x72, 0x20, 0x73, 0x69, 0x64, 0x65,
       0x2C, 0x20, 0x69, 0x74, 0x20, 0x69, 0x73, 0x20, 0x74, 0x68,
       0x65, 0x20, 0x72, 0x65, 0x63, 0x65, 0x69, 0x76, 0x65, 0x20,
       0x6B, 0x65, 0x79, 0x2E};

   Magic3[84] =
      {0x4F, 0x6E, 0x20, 0x74, 0x68, 0x65, 0x20, 0x63, 0x6C, 0x69,
       0x65, 0x6E, 0x74, 0x20, 0x73, 0x69, 0x64, 0x65, 0x2C, 0x20,
       0x74, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x74, 0x68,
       0x65, 0x20, 0x72, 0x65, 0x63, 0x65, 0x69, 0x76, 0x65, 0x20,
       0x6B, 0x65, 0x79, 0x3B, 0x20, 0x6F, 0x6E, 0x20, 0x74, 0x68,
       0x65, 0x20, 0x73, 0x65, 0x72, 0x76, 0x65, 0x72, 0x20, 0x73,
       0x69, 0x64, 0x65, 0x2C, 0x20, 0x69, 0x74, 0x20, 0x69, 0x73,
       0x20, 0x74, 0x68, 0x65, 0x20, 0x73, 0x65, 0x6E, 0x64, 0x20,
       0x6B, 0x65, 0x79, 0x2E};


      GetMasterKey(
      IN  16-octet  PasswordHashHash,
      IN  24-octet  NTResponse,
      OUT 16-octet  MasterKey )
      {
         20-octet Digest

         ZeroMemory(Digest, sizeof(Digest));

         /*
          * SHSInit(), SHSUpdate() and SHSFinal()
```

```
         * are an implementation of the Secure Hash Standard [8].
         */

        SHSInit(Context);
        SHSUpdate(Context, PasswordHashHash, 16);
        SHSUpdate(Context, NTResponse, 24);
        SHSUpdate(Context, Magic1, 27);
        SHSFinal(Context, Digest);

        MoveMemory(MasterKey, Digest, 16);
    }

    VOID
    GetAsymetricStartKey(
    IN   16-octet      MasterKey,
    OUT  8-to-16 octet SessionKey,
    IN   INTEGER       SessionKeyLength,
    IN   BOOLEAN       IsSend,
    IN   BOOLEAN       IsServer )
    {

        20-octet Digest;

        ZeroMemory(Digest, 20);

        if (IsSend) {
           if (IsServer) {
              s = Magic3
           } else {
              s = Magic2
           }
        } else {
           if (IsServer) {
              s = Magic2
           } else {
              s = Magic3
           }
        }

        /*
         * SHSInit(), SHSUpdate() and SHSFinal()
         * are an implementation of the Secure Hash Standard [8].
         */

        SHSInit(Context);
        SHSUpdate(Context, MasterKey, 16);
        SHSUpdate(Context, SHSpad1, 40);
        SHSUpdate(Context, s, 84);
```

```
        SHSUpdate(Context, SHSpad2, 40);
        SHSFinal(Context, Digest);

        MoveMemory(SessionKey, Digest, SessionKeyLength);
     }
```

5. Sample Key Derivations

The following sections illustrate both 40- and 128-bit key derivations. All intermediate values are in hexadecimal.

5.1. Sample 40-bit Key Derivation

```
Initial Values
   UserName = "User" =  55 73 65 72
   Password = "clientPass" = 63 00 6C 00 69 00 65 00 6E 00 74 00 50 00 61 00 73
00 73 00
   AuthenticatorChallenge = 5B 5D 7C 7D 7B 3F 2F 3E 3C 2C 60 21 32 26 26 28
   PeerChallenge = 21 40 23 24 25 5E 26 2A 28 29 5F 2B 3A 33 7C 7E
   Challenge = D0 2E 43 86 BC E9 12 26
   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

Step 1: NtPasswordHash(Password, PasswordHash)
   PasswordHash = 44 EB BA 8D 53 12 B8 D6 11 47 44 11 F5 69 89 AE

Step 2: PasswordHashHash = MD4(PasswordHash)
   PasswordHashHash = 41 C0 0C 58 4B D2 D9 1C 40 17 A2 A1 2F A5 9F 3F

Step 2: Derive the master key (GetMasterKey())
   MasterKey = FD EC E3 71 7A 8C 83 8C B3 88 E5 27 AE 3C DD 31

Step 3: Derive the master send session key (GetAsymmetricStartKey())

   SendStartKey40 = 8B 7C DC 14 9B 99 3A 1B

Step 4: Derive the intial send session key (GetNewKeyFromSHA())
   SendSessionKey40 = D1 26 9E C4 9F A6 2E 3E

Sample Enrypted Message
   rc4(SendSessionKey40, "test message") = 92 91 37 91 7E 58 03 D6 68 D7 58 98
```

5.2. Sample 128-bit Key Derivation

```
Initial Values
   UserName = "User" =  55 73 65 72
```

```
   Password = "clientPass" = 63 00 6C 00 69 00 65 00 6E 00 74 00 50 00 61 00 73
00 73 00
   AuthenticatorChallenge = 5B 5D 7C 7D 7B 3F 2F 3E 3C 2C 60 21 32 26 26 28
   PeerChallenge = 21 40 23 24 25 5E 26 2A 28 29 5F 2B 3A 33 7C 7E
   Challenge = D0 2E 43 86 BC E9 12 26
   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

Step 1: NtPasswordHash(Password, PasswordHash)
   PasswordHash = 44 EB BA 8D 53 12 B8 D6 11 47 44 11 F5 69 89 AE

Step 2: PasswordHashHash = MD4(PasswordHash)
   PasswordHashHash = 41 C0 0C 58 4B D2 D9 1C 40 17 A2 A1 2F A5 9F 3F

Step 2: Derive the master key (GetMasterKey())
   MasterKey = FD EC E3 71 7A 8C 83 8C B3 88 E5 27 AE 3C DD 31

Step 3: Derive the send master session key (GetAsymmetricStartKey())

   SendStartKey128 = 8B 7C DC 14 9B 99 3A 1B A1 18 CB 15 3F 56 DC CB

Step 4: Derive the intial send session key (GetNewKeyFromSHA())
   SendSessionKey128 = 40 5C B2 24 7A 79 56 E6 E2 11 00 7A E2 7B 22 D4

Sample Enrypted Message
   rc4(SendSessionKey128, "test message") = 81 84 83 17 DF 68 84 62 72 FB 5A BE
```

6. Security Considerations

Since the MPPE session keys are derived from user passwords, care should be taken to ensure the selection of strong passwords and passwords should be changed frequently.

7. References

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

[2] Rand, D., "The PPP Compression Control Protocol (CCP)", RFC 1962, June 1996

[3] Zorn, G., "Microsoft PPP CHAP Extensions, Version 2", draft-ietf-pppext-mschap-v2-01.txt (work in progress), October 1998

[4] Pall, G. S., & Zorn, G., "Microsoft Point-to-Point Encryption (MPPE) Protocol", draft-ietf-pppext-mppe-02.txt (work in progress), July 1998

[5] 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

[6] Pall, G. S., "Microsoft Point-to-Point Compression (MPPC) Protocol", RFC 2118, March 1997

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

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

8. Acknowledgements

Thanks (in no particular order) to Tony Bell, Paul Leach, Terence Spies, Robert Friend, Joe Davies, Jody Terrill, Archie Cobbs, Mark Deuser, Brad Robel-Forrest and Jeff Haag for useful suggestions and feedback.

9. Chair's Address

The PPP Extensions Working Group can be contacted via the current chair:

Karl Fox
Ascend Communications
3518 Riverside Drive
Suite 101
Columbus, OH 43221

Phone: +1 614 326 6841
Email: karl@ascend.com

10. 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

11. Expiration Date

This memo is filed as <draft-ietf-pppext-mschapv1-keys-02.txt> and expires on May 15, 1999.