Network Working Group
Request for Comments: 3079
Category: Informational

G. Zorn
cisco Systems

March 2001

Deriving Keys for use with Microsoft Point-to-Point Encryption (MPPE)

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) provides a standard method for transporting multi-protocol datagrams over point-to-point links.

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

Microsoft Point to Point Encryption (MPPE) is a means of representing PPP packets in an encrypted form. MPPE uses the RSA RC4 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, 56-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 in the Compression Control Protocol.

This document describes the method used to derive initial MPPE session keys from a variety of credential types. It is expected that this memo will be updated whenever Microsoft defines a new key derivation method for MPPE, since its primary purpose is to provide an open, easily accessible reference for third-parties wishing to interoperate with Microsoft products.

MPPE itself (including the protocol used to negotiate its use, the details of the encryption method used and the algorithm used to change session keys during a session) is described in RFC 3078.

Table of Contents

1. Specification of Requirements 2
2. Deriving Session Keys from MS-CHAP Credentials 2
2.1. Generating 40-bit Session Keys 3
2.2. Generating 56-bit Session Keys 3
2.3. Generating 128-bit Session Keys 4
2.4. Key Derivation Functions 5
2.5. Sample Key Derivations 6
2.5.1. Sample 40-bit Key Derivation 6
2.5.2. Sample 56-bit Key Derivation 6
2.5.3. Sample 128-bit Key Derivation 7
3. Deriving Session Keys from MS-CHAP-2 Credentials 7
3.1. Generating 40-bit Session Keys 8
3.2. Generating 56-bit Session Keys 9
3.3. Generating 128-bit Session Keys 10
3.4. Key Derivation Functions 11
3.5. Sample Key Derivations 13
3.5.1. Sample 40-bit Key Derivation 13
3.5.2. Sample 56-bit Key Derivation 14
3.5.3. Sample 128-bit Key Derivation 15
4. Deriving MPPE Session Keys from TLS Session Keys 16
4.1. Generating 40-bit Session Keys 16
4.2. Generating 56-bit Session Keys 17
4.3. Generating 128-bit Session Keys 17
5. Security Considerations 18
5.1. MS-CHAP Credentials 18
5.2. EAP-TLS Credentials 19
6. References 19
7. Acknowledgements 20
8. Author's Address 20
9. Full Copyright Statement 21

1. 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 [6].

2. Deriving Session Keys from MS-CHAP Credentials

The Microsoft Challenge-Handshake Authentication Protocol (MS-CHAP-1) [2] is a Microsoft-proprietary PPP [1] authentication protocol, providing the functionality to which LAN-based users are accustomed while integrating the encryption and hashing algorithms used on Windows networks.

The following sections detail the methods used to derive initial session keys (40-, 56- and 128-bit) from MS-CHAP-1 credentials.

Implementation Note

The initial session key in both directions is derived from the credentials of the peer that initiated the call and the challenge used (if any) is the challenge from the first authentication. This is true for both unilateral and bilateral authentication, as well as 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.

2.1. Generating 40-bit Session Keys

MPPE uses a derivative of the peer's LAN Manager password as the 40-bit session key used for initializing the RC4 encryption tables.

The first step is to obfuscate the peer's password using the LmPasswordHash() function (described in [2]). The first 8 octets of the result are used as the basis for the session key generated in the following way:

```
/*
* PasswordHash is the basis for the session key
* SessionKey is a copy of PasswordHash and is the generative session key
* 8 is the length (in octets) of the key to be generated.
*
*/
Get_Key(PasswordHash, SessionKey, 8)

/*
* The effective length of the key is reduced to 40 bits by
* replacing the first three bytes as follows:
*/
SessionKey[0] = 0xd1 ;
SessionKey[1] = 0x26 ;
SessionKey[2] = 0x9e ;
```

2.2. Generating 56-bit Session Keys

MPPE uses a derivative of the peer's LAN Manager password as the 56-bit session key used for initializing the RC4 encryption tables.

The first step is to obfuscate the peer's password using the LmPasswordHash() function (described in [2]). The first 8 octets of the result are used as the basis for the session key generated in the following way:

```
/*
* PasswordHash is the basis for the session key
* SessionKey is a copy of PasswordHash and is the generative session key
* 8 is the length (in octets) of the key to be generated.
*
*/
Get_Key(PasswordHash, SessionKey, 8)

/*
* The effective length of the key is reduced to 56 bits by
* replacing the first byte as follows:
*/
SessionKey[0] = 0xd1 ;
```

2.3. Generating 128-bit Session Keys

MPPE uses a derivative of the peer's Windows NT password as the 128-bit session key used for initializing encryption tables.

The first step is to obfuscate the peer's password using NtPasswordHash() function as described in [2]. The first 16 octets of the result are then hashed again using the MD4 algorithm. The first 16 octets of the second hash are used as the basis for the session key generated in the following way:

```
/*
* Challenge (as described in [9]) is sent by the PPP authenticator
* during authentication and is 8 octets long.
* NtPasswordHashHash is the basis for the session key.
* On return, InitialSessionKey contains the initial session
* key to be used.
*/
Get_Start_Key(Challenge, NtPasswordHashHash, InitialSessionKey)

/*
* CurrentSessionKey is a copy of InitialSessionKey
* and is the generative session key.
* Length (in octets) of the key to generate is 16.
*
*/
Get_Key(InitialSessionKey, CurrentSessionKey, 16)
```

2.4. Key Derivation Functions

The following procedures are used to derive the session key.

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

SHApad1[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};

SHApad2[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};

/*
 * SHAInit(), SHAUpdate() and SHAFinal() functions are an
 * implementation of Secure Hash Algorithm (SHA-1) [7]. These are
 * available in public domain or can be licensed from
 * RSA Data Security, Inc.
 *
 * 1) InitialSessionKey is 8 octets long for 56- and 40-bit
 *    session keys, 16 octets long for 128 bit session keys.
 * 2) CurrentSessionKey is same as InitialSessionKey when this
 *    routine is called for the first time for the session.
 */

Get_Key(
IN     InitialSessionKey,
IN/OUT CurrentSessionKey
IN     LengthOfDesiredKey )
{
   SHAInit(Context)
   SHAUpdate(Context, InitialSessionKey, LengthOfDesiredKey)
   SHAUpdate(Context, SHAPad1, 40)
   SHAUpdate(Context, CurrentSessionKey, LengthOfDesiredKey)
   SHAUpdate(Context, SHAPad2, 40)
   SHAFinal(Context, Digest)
   memcpy(CurrentSessionKey, Digest, LengthOfDesiredKey)
}

Get_Start_Key(
IN  Challenge,
```

```
IN  NtPasswordHashHash,
OUT InitialSessionKey)
{
   SHAInit(Context)
   SHAUpdate(Context, NtPasswordHashHash, 16)
   SHAUpdate(Context, NtPasswordHashHash, 16)
   SHAUpdate(Context, Challenge, 8)
   SHAFinal(Context, Digest)
   memcpy(InitialSessionKey, Digest, 16)
}
```

2.5. Sample Key Derivations

The following sections illustrate 40-, 56- and 128-bit key derivations. All intermediate values are in hexadecimal.

2.5.1. Sample 40-bit Key Derivation

Initial Values
Password = "clientPass"

Step 1: LmPasswordHash(Password, PasswordHash)
PasswordHash = 76 a1 52 93 60 96 d7 83 0e 23 90 22 74 04 af d2

Step 2: Copy PasswordHash to SessionKey
SessionKey = 76 a1 52 93 60 96 d7 83 0e 23 90 22 74 04 af d2

Step 3: GetKey(PasswordHash, SessionKey, 8)
SessionKey = d8 08 01 53 8c ec 4a 08

Step 4: Reduce the effective key length to 40 bits
SessionKey = d1 26 9e 53 8c ec 4a 08

2.5.2. Sample 56-bit Key Derivation

Initial Values
Password = "clientPass"

Step 1: LmPasswordHash(Password, PasswordHash)
PasswordHash = 76 a1 52 93 60 96 d7 83 0e 23 90 22 74 04 af d2

Step 2: Copy PasswordHash to SessionKey
SessionKey = 76 a1 52 93 60 96 d7 83 0e 23 90 22 74 04 af d2

Step 3: GetKey(PasswordHash, SessionKey, 8)
SessionKey = d8 08 01 53 8c ec 4a 08

Step 4: Reduce the effective key length to 56 bits
SessionKey = d1 08 01 53 8c ec 4a 08

2.5.3. Sample 128-bit Key Derivation

Initial Values
Password = "clientPass"
Challenge = 10 2d b5 df 08 5d 30 41

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 3: GetStartKey(Challenge, PasswordHashHash, InitialSessionKey)
InitialSessionKey = a8 94 78 50 cf c0 ac ca d1 78 9f b6 2d dc dd b0

Step 4: Copy InitialSessionKey to CurrentSessionKey
CurrentSessionKey = a8 94 78 50 cf c0 ac c1 d1 78 9f b6 2d dc dd b0

Step 5: GetKey(InitialSessionKey, CurrentSessionKey, 16)
CurrentSessionKey = 59 d1 59 bc 09 f7 6f 1d a2 a8 6a 28 ff ec 0b 1e

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

Version 2 of the Microsoft Challenge-Handshake Authentication Protocol (MS-CHAP-2) [8] 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.

The following sections detail the methods used to derive initial session keys from MS-CHAP-2 credentials. 40-, 56- and 128-bit keys are all 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- and 56-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.

3.1. 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 [8].

```
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 [8] 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 GetNewKeyFromSHA() (described in [3]):

```
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)
```

3.2. Generating 56-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 [8].

```
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 [8] 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 56-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 GetNewKeyFromSHA() (described in [3]):

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

Next, the effective strength of both keys is reduced by setting the first octet to a known constant:

```
SendSessionKey[0] = ReceiveSessionKey[0] = 0xd1
```

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

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

3.3. 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 NtPasswordHash() function as described in [8].

```
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 [8] 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 GetNewKeyFromSHA() (described in [3]):

```
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)
```

3.4. 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 [7].
       */

      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 [7].
       */

      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);
   }
```

3.5. Sample Key Derivations

The following sections illustrate 40-, 56- and 128-bit key derivations. All intermediate values are in hexadecimal.

3.5.1. 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 3: Derive the master key (GetMasterKey())
   MasterKey = FD EC E3 71 7A 8C 83 8C B3 88 E5 27 AE 3C DD 31

Step 4: Derive the master send session key (GetAsymmetricStartKey())
   SendStartKey40 = 8B 7C DC 14 9B 99 3A 1B

Step 5: Derive the initial send session key (GetNewKeyFromSHA())
   SendSessionKey40 = D1 26 9E C4 9F A6 2E 3E

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

3.5.2. Sample 56-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 3: Derive the master key (GetMasterKey())
   MasterKey = FD EC E3 71 7A 8C 83 8C B3 88 E5 27 AE 3C DD 31

Step 4: Derive the master send session key (GetAsymmetricStartKey())
   SendStartKey56 = 8B 7C DC 14 9B 99 3A 1B
```

```
Step 5: Derive the initial send session key (GetNewKeyFromSHA())
   SendSessionKey56 = D1 5C 00 C4 9F A6 2E 3E

Sample Encrypted Message
   rc4(SendSessionKey40, "test message") = 3F 10 68 33 FA 44 8D
                                           A8 42 BC 57 58
```

3.5.3. 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 initial send session key (GetNewKeyFromSHA())
   SendSessionKey128 = 40 5C B2 24 7A 79 56 E6 E2 11 00 7A E2 7B 22 D4

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

4. Deriving MPPE Session Keys from TLS Session Keys

The Extensible Authentication Protocol (EAP) [10] is a PPP extension that provides support for additional authentication methods within PPP. Transport Level Security (TLS) [11] provides for mutual authentication, integrity-protected ciphersuite negotiation and key exchange between two endpoints. EAP-TLS [12] is an EAP authentication type which allows the use of TLS within the PPP authentication framework. The following sections describe the methods used to derive initial session keys from TLS session keys. 56-, 40- and 128-bit keys are derived using the same algorithm. The only difference is in the length of the keys and their effective strength: 56- and 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.

4.1. Generating 40-bit Session Keys

When MPPE is used in conjunction with EAP-TLS authentication, the TLS master secret is used as the master session key.

The algorithm used to derive asymmetrical master session keys from the TLS master secret is described in [12]. The master session keys are never used to encrypt or decrypt data; they are only used in the derivation of transient session keys.

Implementation Note

> If the asymmetrical master keys are less than 8 octets in length, they MUST be padded on the left with zeroes before being used to derive the initial transient session keys. Conversely, if the asymmetrical master keys are more than 8 octets in length, they must be truncated to 8 octets before being used to derive the initial transient session keys.

The initial transient session keys are obtained by calling the function GetNewKeyFromSHA() (described in [3]):

```
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 56-bit Session Keys

When MPPE is used in conjunction with EAP-TLS authentication, the TLS master secret is used as the master session key.

The algorithm used to derive asymmetrical master session keys from the TLS master secret is described in [12]. The master session keys are never used to encrypt or decrypt data; they are only used in the derivation of transient session keys.

Implementation Note

If the asymmetrical master keys are less than 8 octets in length, they MUST be padded on the left with zeroes before being used to derive the initial transient session keys. Conversely, if the asymmetrical master keys are more than 8 octets in length, they must be truncated to 8 octets before being used to derive the initial transient session keys.

The initial transient session keys are obtained by calling the function GetNewKeyFromSHA() (described in [3]):

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

Next, the effective strength of both keys is reduced by setting the initial octet to a known constant:

```
SendSessionKey[0] = ReceiveSessionKey[0] = 0xD1
```

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

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

4.3. Generating 128-bit Session Keys

When MPPE is used in conjunction with EAP-TLS authentication, the TLS master secret is used as the master session key.

The algorithm used to derive asymmetrical master session keys from the TLS master secret is described in [12]. Note that the send key on one side is the receive key on the other.

The master session keys are never used to encrypt or decrypt data; they are only used in the derivation of transient session keys.

Implementation Note

If the asymmetrical master keys are less than 16 octets in length, they MUST be padded on the left with zeroes before being used to derive the initial transient session keys. Conversely, if the asymmetrical master keys are more than 16 octets in length, they must be truncated to 16 octets before being used to derive the initial transient session keys.

The initial transient session keys are obtained by calling the function GetNewKeyFromSHA() (described in [3]):

```
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)
```

5. Security Considerations

5.1. MS-CHAP Credentials

Because of the way in which 40-bit keys are derived from MS-CHAP-1 credentials, the initial 40-bit session key will be identical in all sessions established under the same peer credentials. For this reason, and because RC4 with a 40-bit key length is believed to be a relatively weak cipher, peers SHOULD NOT use 40-bit keys derived from the LAN Manager password hash (as described above) if it can be avoided.

Since the MPPE session keys are derived from user passwords (in the MS- CHAP-1 and MS-CHAP-2 cases), care should be taken to ensure the selection of strong passwords and passwords should be changed frequently.

5.2. EAP-TLS Credentials

The strength of the session keys is dependent upon the security of the TLS protocol.

The EAP server may be on a separate machine from the PPP authenticator; if this is the case, adequate care must be taken in the transmission of the EAP-TLS master keys to the authenticator.

6. References

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

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

[3] Pall, G. and G. Zorn, "Microsoft Point-to-Point Encryption (MPPE) RFC 3078, March 2001.

[4] 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

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

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

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

[8] Zorn, G., "Microsoft PPP CHAP Extensions, Version 2", RFC 2759, January 2000.

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

[10] Blunk, L. and J. Vollbrecht, "PPP Extensible Authentication Protocol (EAP)", RFC 2284, March 1998.

[11] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[12] Aboba, B. and D. Simon, "PPP EAP TLS Authentication Protocol", RFC 2716, October 1999.

7. Acknowledgements

Anthony Bell, Richard B. Ward, Terence Spies and Thomas Dimitri, all of Microsoft Corporation, significantly contributed to the design and development of MPPE.

Additional thanks to Robert Friend, Joe Davies, Jody Terrill, Archie Cobbs, Mark Deuser, Vijay Baliga, Brad Robel-Forrest and Jeff Haag for useful feedback.

The technical portions of this memo were completed while the author was employed by Microsoft Corporation.

8. Author’s Address

Questions about this memo can also be directed to:

Glen Zorn
cisco Systems
500 108th Avenue N.E.
Suite 500
Bellevue, Washington 98004
USA

Phone: +1 425 438 8218
FAX: +1 425 438 1848
EMail: gwz@cisco.com

9. Full Copyright Statement

Acknowledgement

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