Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2020

B. Haase
Endress + Hauser Liquid Analysis

March 9, 2020

Key encoding for manual typing operations. draft-haase-psk-encoding-00

Abstract

This document specifies a string encoding of external pre-shared keys (PSK) for applications where the key has to be entered manually by use of an alphanumeric or numeric keyboard.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on September 10, 2020.

Table of Contents

1. Introduction

In some applications pre-shared keys (PSK) are used as primary means of authentication, specifically in settings where a public key infrastructure is not available. When PSK are used as root of trust, a sufficient entropy of the keying material is crucial because otherwise attacks such as offline dictionary attacks could be mounted. Unfortunately, not all users might be aware of the corresponding pitfalls and might be tempted to use low entropy password strings as PSK.

The situation is particularly difficult, if no trusted binary machine communication interface is available for initially configuring the PSK in a device, as might be the case for several classes of wireless devices. In some environments, key material needs to be entered manually by use of keyboards or touch screens with limited functionality. On some devices, such as small touch screens, only a numerical keypad might be available.

Manually tying keys should be considered error prone. Users might also not be able to distinguish between authentication failures due to unmatching keys and unmatching keys that just result from typing errors. Without guidance and a user-friendly encoding, users might be tempted to use short low entropy passwords. Note that use of low-entropy passwords is suitable only for protocols such as password-authenticated key exchange (PAKE) but not for the typical protocols using pre-shared keys, such as TLS-PSK. This crucial difference, might not be transparent for some end-users.

This specification aims at addressing this issue by specifying a format for PSK key encoding specifically designed for convenient manual typing. Moreover it is assumed that the encoding specified

here needs to be generated by a software tool having access to a cryptographic random number generator. Such a tool-based approach could guarantee a sufficient entropy of the PSK and, thus, accidential mis-use of a PSK-based protocol with a low-entropy secret. Moreover this encoding provides guidings to the end-user by detecting obvious typing errors by use of error-detection codes.

Similar use cases were previously considered in

2. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Considered requirement set

The encoding was designed to consider the following requirement set.

- The string representation needs to group display characters and numbers in groups of up to 6 digits or characters for better segmentation in manual entry.

- Typing errors should be identified by the terminal by using an error detection code.

- The encoding should help identifying the position of typing errors by determining the error detection code to substrings.

- The encoding should be as short as possible in order to avoid cumbersome typing.

- The encoding should allow for use of purely numerical and alpha-numerical keyboards.

- On many keyboards, such as modern touch-screen keyboards, switching between uppercase and lowercase letters needs an additional typing operation. For this reason, the encodings shall only use uppercase letters.

4. Encoding structure

The encodings for numeric and alpha-numeric key entry SHALL be printed or displayed in the following form in groups of 2x6 digits (0...9) or groups of 2x5 characters respectively. These groups of

2x6 digits or 2x5 characters will be referred to as chunks throughout this specification.

An example for an encoding of a 128 bit PSK is given below:

```
Example representation for numeric keyboards:
 310096 751463
 862948 315830
 085914 540353
 606738 970243

Example representation for alphanumeric keyboards:
 X4RTQ 4KPKM
 PTWXS 3BP4Z
 C66D5 RRJ26
```

The character encoding use Crockford's variant for BASE32 which considers similar characters such as 0 and O as one and the same symbol.

Each chunk shall be individually checked by use of the CRC7 checksum, such as used for memory checking algorithms (MMC).

4.1. Alphanumeric encoding

The alphanumeric encoding splits the key into chunks of 2x5 characters. Since with Crockford's BASE32 each character encodes 5 bits, 10 characters can encode 50 bits in total. Within these 50 bits, 43 bits are used for encoding data, and 7 bits are used for encoding the result of the CRC7 algorithm over the preceding 43 bits and an application specific domain-separation string DSI.

This way, an 128 bit key can be encoded by using 30 characters. The CRC only accounts for an overhead of roughly 4 characters.

Applications using this encoding shall use a domain seperation string, such as "TLS-PSK128" or "TLS-PSK256". The domain separation string SHALL also specify the exact expected key length.

For the encoding the key shall be split into chunks of 43 bits. The last chunk's payload shall be padded with zero bits.

The CRC7 shall be calculated over the concatenation of the DSI string

4.2. Numeric encoding

The numeric encoding shall split the key into chunks of 2x6 digits, encoding 32 bits of data and 7 bits of checksum.

This way, a 128 bit key could be encoded by using 4 chunks of 2x6 digits.

5. Reference implementation for encoding

The following python 3 code could be used as reference implementation for the encoding and decoding functions.

```
<CODE BEGINS>
import libscrc

def LEStringToInteger(k):
        bytes = [b for b in k]
        return sum((bytes[i]<<(8 * i)) for i in range(len(bytes)))

def IntegerToLEString(k):
    kInt = k;
    result = [];
    n = 0
    while (kInt):
        result.append(kInt & 0xff);
        kInt = kInt >> 8
        n = n + 1
    return result

def crockfordBase32Encode(val,blockSize = 5, numBlocks = 2):
    table = b'0123456789ABCDEFGHJKNMPQRSTVWXYZ'
    result = b""
    ctr = 0
    while (1):
        ctr += 1
        index = val & 0x1f
        val = val >> 5
        result+= table[index:(index + 1)]

        if (ctr == blockSize):
            numBlocks -= 1
            if (val == 0) and (numBlocks == 0):
                return result
            result += b" "
            ctr = 0

    return result
```

```
  def Base10Encode(val,blockSize = 7, numBlocks = 2):
      table = b'0123456789'
      result = b""
      ctr = 0
      while (1):
          ctr += 1
          index = val % 10
          val = round((val - index) / 10)
          result+= table[index:(index + 1)]

          if (ctr == blockSize):
              numBlocks -= 1
              if (val == 0) and (numBlocks == 0):
                  return result
              result += b" "
              ctr = 0

      return result

  def encodeKeyAsString(key, domainSeparator = b"PSK128"):
      keyAsInt = LEStringToInteger(key)
      result = b""
      chunkNo = 0
      while (keyAsInt):
          # take chunks of 43 bits and calculate a CRC7
          # encode each chunk as 2 x 5 = 10 characters
          chunk = keyAsInt - ((keyAsInt >> 43) << 43)
          keyAsInt = keyAsInt >> 43
          crc = libscrc.mmc(domainSeparator + bytes([chunkNo])
                            + bytes(IntegerToLEString(chunk)))
          chunkWithCrc = chunk + (crc << 43)
          chunkNo += 1

          result += crockfordBase32Encode(chunkWithCrc)
          if (keyAsInt):
              result += b"   ";

      return result

  def encodeKeyAsDigits(key,domainSeparator = b"PSK128"):
      keyAsInt = LEStringToInteger(key)
      result = b""
      debugPrints = 0
      chunkNo = 0
      while (keyAsInt):
          # take chunks of 32 bits and calculate a CRC7
          # encode each chunk as 2 x 6 = 12 digits
          chunk = keyAsInt - ((keyAsInt >> 32) << 32)
```

```
        keyAsInt = keyAsInt >> 32

        crc = libscrc.mmc(domainSeparator + bytes([chunkNo])
                          + bytes(IntegerToLEString(chunk)))
        chunkWithCrc = chunk  + (crc << 32)

        result += Base10Encode(chunkWithCrc,6)
        chunkNo += 1

        if (keyAsInt):
            result += b"   ";

    return result

def crockfordBase32DecodeChar(x):
    toDecode = x.upper();
    table = b'0123456789ABCDEFGHJKNMPQRSTVWXYZ'

    if (toDecode == b'O'):
        toDecode = b'0'

    if ((toDecode == b'I') or (toDecode == b'L')):
        toDecode = b'1'

    return table.index(toDecode);

def decodeString(x):
    result = 0

    characters = x;

    if (len(characters) > 0):
        result += crockfordBase32DecodeChar(characters[0:1]);
        result += 32 * decodeString(characters[1:])

    return result

def decodeDigits(digits):
    result = 0

    if (len(digits) > 0):
        result += digits[0] - ord('0')
        result += 10 * decodeDigits(digits[1:])
    return result


def decodeKeyFromDigits(digits,domainSeparator = b"PSK128"):
    remainingDigits = digits
```

```
      result = 0

      factor = 1
      chunkNo = 0
      while (1):
          remainingDigits = remainingDigits.lstrip()

          if (len(remainingDigits) == 0):
              return IntegerToLEString(result)

          subChunk1 = remainingDigits[0:6]
          remainingDigits = (remainingDigits[6:]).lstrip()

          subChunk2 = remainingDigits[0:6]
          remainingDigits = (remainingDigits[6:]).lstrip()

          ChunkWithCrc = (decodeDigits(subChunk1)
                          + (10**6) * decodeDigits(subChunk2))

          # take chunks of 32 bits and calculate a CRC7
          chunk = ChunkWithCrc & 0xffffffff
          decodedCrc = ChunkWithCrc >> 32

          calculatedCrc = libscrc.mmc(domainSeparator + bytes([chunkNo])
                                      + bytes(IntegerToLEString(chunk)))
          if (calculatedCrc != decodedCrc):
              raise ValueError(b"detected typing error in chunk "
                               + subChunk1 + b" " + subChunk2 + b".")

          result += chunk * factor
          factor = factor << 32
          chunkNo += 1

      return result


  def decodeKeyFromString(digits,domainSeparator = b"PSK128"):
      remainingDigits = digits
      result = 0

      factor = 1
      chunkNo = 0
      while (1):
          remainingDigits = remainingDigits.lstrip()

          if (len(remainingDigits) == 0):
              return IntegerToLEString(result)
```

```
        subChunk1 = remainingDigits[0:5]
        remainingDigits = (remainingDigits[5:]).lstrip()

        subChunk2 = remainingDigits[0:5]
        remainingDigits = (remainingDigits[5:]).lstrip()


        ChunkWithCrc = (decodeString(subChunk1)
                       + (decodeString(subChunk2) << (5*5)))

        # take chunks of 43 bits and calculate a CRC7
        decodedCrc = ChunkWithCrc >> 43
        chunk = ChunkWithCrc - (decodedCrc << 43)

        calculatedCrc = libscrc.mmc(domainSeparator + bytes([chunkNo])
                                    + bytes(IntegerToLEString(chunk)))
        if (calculatedCrc != decodedCrc):
            raise ValueError(b"detected typing error in chunk "
                             + subChunk1 + b" " + subChunk2 + b".")

        result += chunk * factor
        factor = factor << 43
        chunkNo += 1
```

<CODE ENDS>

6. Security Considerations

The encoding defined here does not provide any security guarantees except for detection of accidential typing errors.

Accidential typing errors will be detected with a probability in the range of 1% only (CRC7).

Distinct applications SHALL use unique DSI strings, such that accidential re-use of the same key for different applications is typically observed already on the typing error detection level.

7. Status of this draft

Presently this draft is meant just to be used as a sketch of the general idea that came up in the process of the discussions for the preparation of the external PSK guidance documents. Comments are welcome, specifically regarding the question, whether a stronger checksum such as a CRC32 should be included over the entire key. Presently only a large fraction of typing errors will be detected, but with the present formulation using CRC7, this is far from a safe

detection level. This draft was based on the assessment, that for manual typing this overhead might not be acceptable.

8. IANA Considerations

No IANA action is required.

9. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

Appendix A. Test vectors for numeric and alphanumeric encodings

```
   ######################### /128 bit key ############################
   Encoding for digits with DSI = b'PSK128'
   Key: 0xa58c15a63325963cf79ab3b4c97d609d

   Encoded as digits:
        966215 677815
        822225 364180
        636215 761870
        490755 095742
   Encoded as string:
        X4RTQ 4KPKP
        PTWXS 3BPW6
        C66D5 RRJTJ
   ######################### /128 bit key ############################

   ########################  256 bit key ############################
   Encoding for digits with DSI = b'PSK256'
   Key:
   0xd35a29ef387e015227ea161f4d4af51cdd9d5cf099c8d414b1558faa8a9396cb

   Encoded as digits:
      158768 709272
      685258 719703
      697517 428940
      658360 268631
      009721 830874
      742921 983960
      229175 168951
      301905 580550
   Encoded as string:
      BP579 5AM75
      HMAC9 1A3HG
      7KG7Q TSEBS
      ENYAA KY1V6
      1MZ4J A0WQP
      GKQ75 TTNS0
   ######################### /256 bit key ############################
```

Author's Address

Bjoern Haase
Endress + Hauser Liquid Analysis

Email: bjoern.m.haase@web.de