Device Pairing Using Short Authentication Strings
draft-kaiser-dnssd-pairing-00.txt

Abstract

   This document proposes a device pairing mechanism that establishes a
   relationship between two devices by agreeing on a secret and manually
   verifying the secret's authenticity using an SAS (short
   authentication string). Pairing has to be performed only once per
   pair of devices, as for a re-discovery at any later point in time,
   the exchanged secret can be used for mutual authentication.

   The proposed pairing method is suited for each application area where
   human operated devices need to establish a relation that allows
   configurationless and privacy preserving re-discovery at any later
   point in time. Since privacy preserving applications are the main
   suitors, we especially care about privacy.

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 http://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 March 31, 2017.

Table of Contents

[1](#). Introduction

   To engage in secure and privacy preserving communication, hosts need
   to differentiate between authorized peers, which must both know about
   the host's presence and be able to decrypt messages sent by the host,
   and other peers, which must not be able to decrypt the host's
   messages and ideally should not be aware of the host's presence. The
   necessary relationship between host and peer can be established by a
   centralized service, e.g. a certificate authority, by a web of trust,
   e.g. PGP, or -- without using global identities -- by device
   pairing.

   This document proposes a device pairing mechanism that provides human
   operated devices with pairwise authenticated secrets, allowing mutual
   automatic re-discovery at any later point in time along with mutual
   private authentication. We especially care about privacy and user-
   friendliness.

   The proposed pairing mechanism consists of three steps needed to
   establish a relationship between a host and a peer:

   1. Discovery of the peer device. The host needs a means to discover
       network parameters necessary to establish a connection to the
       peer. During this discovery process, neither the host nor the
       peer must disclose its presence.

   2. Agreeing on pairing data. The devices have to agree on pairing
       data, which can be used by both parties at any later point in
       time to generate identifiers for re-discovery and to prove the
       authenticity of the pairing. The pairing data can e.g. be a
       shared secret agreed upon via a Diffie-Hellman key exchange.

   3. Authenticate pairing data. Since in most cases the messages
       necessary to agree upon pairing data are send over an insecure
       channel, means that guarantee the authenticity of these messages
       are necessary; otherwise the pairing data is in turn not suited
       as a means for a later proof of authenticity. For the proposed
       pairing mechanism we use manual interaction involving an SAS
       (short authentication string) to proof the authenticity of the
       pairing data.

[1.1](#). Requirements

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

[2](#). Problem Statement and Requirements

The general pairing requirement is easy to state: establish a trust relation between two entities in a secure manner. But details matter, and in this section we explore the detailed requirements that guide our design.

[2.1](#). Secure Pairing Over Internet Connections

Many pairing protocols have been already developed, in particular for the pairing of devices over specific wireless networks. For example, the current BlueTooth specifications include a pairing protocol that has evolved over several revisions towards better security and usability [BTLEPairing]. The Wi-Fi Alliance defined the Wi-Fi Protected Setup process to ease the setup of security-enabled Wi-Fi networks in home and small office environments [WPS]. Other wireless standards have defined or are defining similar protocols, tailored to specific technologies.

This specification defines a pairing protocol that is independent of the underlying technology. We simply make the hypothesis that the two parties engaged in the pairing can discover each other and can exchange establish connections over IP.

TODO: discuss the actual goal. Is it a shared secret or a public key?

[2.2](#). Identity Assurance

The parties in the pairing must be able to identify each other. To put it simply, if Alice believes that she is establishing a pairing with Bob, she must somehow ensure that the pairing is happening with Bob, and not with some interloper like Eve or Nessie. Providing this assurance requires care at the user interface (UI), and also in the design of the protocol.

Consider for example an attack in which Eve tricks Alice into establishing a pairing, and then pretends to be Bob. Alice must be able to discover that something is wrong, and refuse to establish a pairing. At a minimum, the parties engaged in the pairing must be able to verify the identity under which the pairing is established.

[2.3](#). Adequate User Interface

Because the pairing protocol is executed without prior knowledge, it is typically vulnerable to "Man-in-the-middle" attacks. While Alice is trying to establish a pairing with Bob, Eve positions herself in the middle. Instead of getting a pairing between Alice and Bob, both

Alice and Bob get paired with Eve. This requires specific features in
the protocol to detect man-in-the-middle attacks, and if possible
resist them. The reference [NR11] analyzes the various proposals to
solve this problem, and in this document, we present a layman
description of these issues in Section 2.4. The various protocols
proposed in the literature impose diverse constraints at the UI
interface, which we will review here.

2.3.1. Short PIN Proved Inadequate

The initial Bluetooth pairing protocol relied on a four digit PIN,
displayed by one of the devices to be paired. The user would read
that PIN and provide it to the other device. The PIN would then be
used in a Password Authenticated Key Exchange. Wi-Fi Protected Setup
[WPS] offered a similar option. There were various attacks against
the actual protocol; some of the problems were caused by issues in
the protocol, but most were tied to the usage of short PINs.

In the reference implementation, the PIN is picked at random by the
paired device before the beginning of the exchange. But this
requires that the paired device is capable of generating and
displaying a four digit number. It turns out that many devices
cannot do that. For example, an audio headset does not have any
display capability. These limited devices ended up using static
PINs, with fixed values like "0000" or "0001".

Even when the paired device could display a random PIN, that PIN will
have to be copied by the user on the pairing device. It turns out
that users do not like copying long series of numbers, and the
usability thus dictated that the PINs be short -- four digits in
practice. But there is only so much assurance as can be derived from
a four digit key.

It is interesting to note that the latest revisions of the Bluetooth
Pairing protocol [BTLEPairing] do not include the short PIN option
anymore. The PIN entry methods have been superseded by the simple
"just works" method for devices without displays, and by a procedure
based on an SAS (short authentication string) when displays are
available.

A further problem with these PIN based approaches is that -- in
contrast to SASes -- the PIN is a secret instrumental in the security
algorithm. To guarantee security, this PIN had to be transmitted via
a secure out of band channel.

2.3.2. Push Buttons Just Work, But Are Insecure

   Some devices are unable to input or display any code. The industry
   more or less converged on a "push button" solution. When the button
   is pushed, devices enter a "pairing" mode, during which they will
   accept a pairing request from whatever other device connects to them.

   The Bluetooth Pairing protocol [BTLEPairing] denotes that as the
   "just works" method. It does indeed work, and if the pairing
   succeeds the devices will later be able to use the pairing keys to
   authenticate connections. However, the procedure does not provide
   any protection against MITM attacks during the pairing process. The
   only protection is that pushing the button will only allow pairing
   for a limited time, thus limiting the opportunities of attacks.

   As we set up to define a pairing protocol with a broad set of
   applications, we cannot limit ourselves to an insecure "push button"
   method. But we probably need to allow for a mode of operation that
   works for input-limited and display limited devices.

2.3.3. Short Range Communication

   There have been several attempts to define pairing protocols that use
   "secure channels." Most of them are based on short range
   communication systems, where the short range limits the feasibility
   for attackers to access the channels. Example of such limited
   systems include for example:

   o QR codes, displayed on the screen of one device, and read by the
      camera of the other device.

   o Near Field Communication (NFC) systems, which provides wireless
      communication with a very short range.

   o Sound systems, in which one systems emits a sequence of sounds or
      ultrasounds that is picked by the microphone of the other system.

   A common problem with these solutions is that they require special
   capabilities that may not be present in every device. Another
   problem is that they are often one-way channels. Yet another problem
   is that the side channel is not necessarily secret. QR codes could
   be read by third parties. Powerful radios antennas might be able to
   interfere with NFC. Sensitive microphones might pick the sounds. We
   will discuss the specific case of QR codes in Section 2.7.

2.3.4. Short Authentication Strings

   The evolving pairing protocols seem to converge towards a "display
   and compare" method. This is in line with academic studies, such as
   [KFR09] or [USK11]. This points to a very simple scenario:

   1. Alice initiates pairing

   2. Bob selects Alice's device from a list.

   3. Alice and Bob compare displayed strings that represent a
       fingerprint of the key.

   4. If the strings match, Alice and Bob accept the pairing.

   Most existing pairing protocols display the fingerprint of the key as
   a 6 or 7 digit numbers. Usability studies show that gives good
   results, with little risk that users mistakenly accept two different
   numbers as matching. However, the authors of [USK11] found that
   people had more success comparing computer generated sentences than
   comparing numbers. This is in line with the argument in [XKCD936] to
   use sequences of randomly chosen common words as passwords. On the
   other hand, standardizing strings is more complicated than
   standardizing numbers. We would need to specify a list of common
   words, and the process to go from a binary fingerprint to a set of
   words. We would need to be concerned with internationalization
   issues, such as using different lists of words in German and in
   English. This could require negotiation of word lists or languages
   inside the pairing protocols.

   In contrast, numbers are easy to specify, as in "take a 20 bit number
   and display it as an integer using decimal notation."

2.4. Resist cryptographic attacks

   It is tempting to believe that once two peers are connected, they
   could create a secret with a few simple steps, such as for example
   exchange two nonces, hash the concatenation of this nonces with the
   shared secret, display a short authentication string composed of a
   short version of that hash on each device, and verify that the two
   values match. The sequence of messages would be something like:

```
     Alice                         Bob
     g^xA -->

                          <-- g^xB

     nA -->
                            <-- nB
     Computes              Computes
     s = g^xAxB            s = g^xAxB
     h = hash(s|nA|nB)     h = hash(s|nA|nB)
     Displays short        Displays short
     version of h          version of h
```

If the two short hashes match, Alice and Bob are supposedly assured
that they have computed the same secret, but there is a problem. The
exchange may not deter a smart attacker in the middle. Let's redraw
the same message flow, this time involving Eve:

```
     Alice                 Eve                 Bob
     g^xA -->

                    g^xA'-->
                                    <-- g^xB
                 <--g^xB'
     nA -->
                    nA -->
                                     <-- nB
                  Picks nB'
                  smartly
                 <--nB'
     Computes                        Computes
     s' = g^xAxB'                      s" = g^xA'xB
     h' = hash(s|nA|nB')              h" = hash(s"|nA|nB)
     Displays short                   Displays short
     version of h'                    version of h"
```

Let's now assumes that to pick the nonce nB' smartly, Eve runs the
following algorithm:

```
     s' = g^xAxB'
     s" = g^xA'xB
     repeat
        pick a new version of nB'
        h' = hash(s|nA|nB')
        h" = hash(s"|nA|nB)
     until the short version of h'
     matches the short version of h"
```

Of course, running this algorithm will require in theory as many
iterations as the possible values of the short hash. But hash
algorithms are fast, and it is possible to try millions of values in

less than a second. If the short string is made up of fewer than 6
digits, Eve will find a matching nonce quickly, and Alice and Bob
will hardly notice the delay. Even if the matching string is as long
as 8 letters, Eve will probably find a value where the short versions
of h' and h" are close enough, e.g. start and end with the same two
or three letters. Alice and Bob may well be fooled.

The classic solution to such problems is to "commit" a possible
attacker to a nonce before sending it. This commitment can be
realized by a hash. In the modified exchange, Alice and Bob exchange
a secure hash of their nonces before sending the actual value:

```
    Alice                         Bob
    g^xA -->

                              <-- g^xB

    Computes                  Computes
    s = g^xAxB                s = g^xAxB
    h_a = hash(s|nA) -->

                                <-- nB

    nA -->
                              verifies h_a == hash(s|nA)
    Computes                  Computes
    h = hash(s|nA|nB)         h = hash(s|nA|nB)
    Displays short            Displays short
    version of h              version of h
```

Alice will only disclose nA after having confirmation from Bob that
hash(nA) has been received. At that point, Eve has a problem. She
can still forge the values of the nonces but she needs to pick the
nonce nA' before the actual value of nA has been disclosed. On first
sight, it seems that once she has committed by sending hash(nA'), it
will be impossible to send anything but the nA'. Eve would still
have a random chance of fooling Alice and Bob, but it will be a very
small chance: one in million if the short authentication string is
made of 6 digits, even fewer if that string is longer.

Nguyen et al. [NR11] survey these protocols and compare them with
respect to the amount of necessary user interaction and the
computation time needed on the devices. The authors state that such
a protocol is optimal with respect user interaction if it suffices
for users to verify a single b-bit SAS while having a one-shot attack
success probability of 2^{-b}. Further, n consecutive attacks on the
protocol must not have a better success probability then n one-shot
attacks.

There is still a theoretical problem, if Eve has somehow managed to
"crack" the hash function. We build some "defense in depth" by some

simple measures. In the design presented above, the hash "h_a"
depends of the shared secret "s", which acts as a "salt" and reduces
the effectiveness of potential attacks based on pre-computed
catalogs. For simplicity, the design used a simple concatenation
mechanism, but we could instead use a keyed-hash message
authentication code (HMAC, [RFC2104], [RFC6151]), using the shared
secret as a key, since the HMAC construct has proven very robust over
time. Then, we can constrain the size of the random numbers to be
exactly the same as the output of the hash function. Hash attacks
often require padding the input string with arbitrary data;
restraining the size limits the likelyhood of such padding.

2.5. Privacy Requirements

Pairing exposes a relation between several devices and their owners.
Adversaries may attempt to collect this information, for example in
an attempt to track devices, their owners, or their "social graph."
It is often argued that pairing could be performed in a safe place,
from which adversaries are assumed absent, but experience shows that
such assumptions are often misguided. It is much safer to
acknowledge the privacy issues and design the pairing process
accordingly.

In order to start the pairing process, devices must first discover
each other. We do not have the option of using the private discovery
protocol [I-D.huitema-dnssd-privacy] since the privacy of that
protocol depends on the pre-existing pairing. In the simplest
design, one of the devices will announce a "friendly name" using DNS-
SD. Adversaries could monitor the discovery protocol, and record
that name. An alternative would be for one device to announce a
random name, and communicate it to the other device via some private
channel. There is an obvious tradeoff here: friendly names are
easier to use but less private than random names. We anticipate that
different users will choose different tradeoffs, for example using
friendly names if they assume that the environment is "safe," and
using random names in public places.

During the pairing process, the two devices establish a connection
and validate a pairing secret. As discussed in Section 2.3, we have
to assume that adversaries can mount MITM attacks. The pairing
protocol can detect such attacks and resist them, but the attackers
will have access to all messages exchanged before validation is
performed. It is important to not exchange any privacy sensitive
information before that validation. This includes, for example, the
identities of the parties or their public keys.

2.6. Using TLS

The pairing algorithms typically combine the establishment of a
shared secret through an [EC]DH exchange, with the verification of
that secret through displaying and comparison of a "short
authentication string" (SAS). As explained in Section 2.4, the
secure comparison requires a "commit before disclose" mechanism.

We have three possible designs: create a pairing algorithm from
scratch, specifying our own crypto exchanges; use an [EC]DH version
of TLS to negotiate a shared secret, export the key to the
application as specified in [RFC5705], and implement the "commit
before disclose" and SAS verification as part of the pairing
application; or, use TLS, integrate the "commit before disclose" and
SAS verification as TLS extensions, and export the verified key to
the application as specified in [RFC5705].

Creating an algorithm from scratch is probably a bad idea. We would
need to reinvent a lot of the negotiation capabilities that are part
of TLS, not to mention algorithm agility, post quantum, and all that
sort of things. It is thus pretty clear that we should use TLS.

It turns out that there was already an attempt to define SAS
extensions for TLS ([I-D.miers-tls-sas]). It is a very close match
to our third design option, full integration of SAS in TLS, but the
draft has expired, and there does not seem to be any support for the
SAS options in the common TLS packages.

In our design, we will choose the middle ground option -- use TLS for
[EC]DH, and implement the SAS verification as part of the pairing
application. This minimizes dependencies on TLS packages to the
availability of a key export API following [RFC5705]. We will need
to specify the hash algorithm used for the SAS computation and
validation, which carries some of the issues associated with
"designing our own crypto." One solution would be to use the same
hash algorithm negotiated by the TLS connection, but common TLS
packages do not not always make this algorithm identifier available
through standard APIs. A fallback solution is to specify a state of
the art keyed MAC algorithm.

2.7. QR codes

In Section 2.3.3, we reviewed a number of short range communication
systems that can be used to facilitate pairing. Out of these, QR
codes stand aside because most devices that can display a short
string can also display the image of a QR code, and because many
pairing scenarios involve cell phones equipped with cameras capable
of reading a QR code.

QR codes could be particularly useful when starting discovery. QR
codes can encode an alphanumeric string, which could for example
encode the selected name of the pairing service. This would enable
automatic discovery, and would be easier to use than reading the
random name of the day and matching it against the results of DNS-SD.

In addition to the instance name, a QR code could also be leveraged
for authentication. It could encode an SAS or even a longer
authentication string. Transmitting the output of a cryptographic
hash function or HMAC via the OOB channel would make an offline
combinatorial search attack infeasible and thus allow to not sent the
commitment discussed in Section 2.4 saving a message. Further, if a
single device created both QR codes for discovery and verification,
respectively, and the other device scans these, the users could just
wait while both QRs are scanned subsequently as no user interaction
is necessary between these two scans (but it needs a QR scanner (app)
that support this). This could make the process feel like a single
user interaction.

But still, from a users point of view, scanning QR codes may not be
more efficient than visual verification of a short string. The user
has to take a picture of the QR code, which is arguably not simpler
than just "look at the number on the screen and tell me whether it is
the same as yours".

In the case of a man-in-the-middle attack, the evaluation of the QR
code will fail. The "client" that took the picture will know that,
but the "server" will not. The user will still need to click some
"Cancel" button on the server, which means that the process will not
be completely automated.

3. Design of the Pairing Mechanism

In this section we discuss the design of pairing protocols that use
manually verified short authentication strings (SAS), focusing on the
user experience. We will make provision for the optional usage of QR
codes.

We divide pairing in three parts: discovery, agreement, and
authentication, detailed in the following subsections.

3.1. Discovery

During the discovery phase, the client will find the server. There
are two possible experiences depending of whether QR codes are
supported or not. If QR codes are supported, the discovery proceeds
as follow:

1. The server displays a QR code containing the its chosen instance
    name.

2. The client scans the QR code, and discovery is performed
    automatically, without further client interaction.

If QR codes are not supported, the discovery will require some user
interaction:

1. The server displays its chosen instance name on its screen.

2. The client performs a discovery of all the "pairing" servers
    available on the local network. This may result in the discovery
    of several servers.

3. The client user selects from the list of available servers the
    instance name that matches the name displayed by the server.

3.2. Agreement

Once the server has been selected, the client connects to it without
further user intervention. Client and server exchange the required
messages, and both display an SAS, normally a number encoded over up
to 7 decimal digits.

The users are asked to validate the pairing by comparing the SASes.
If they match, each user enters an agreement, for example by pressing
a button labeled "OK". If they do not match, each user should cancel
the pairing, for example by pressing a button labeled "CANCEL".

3.2.1. Length and syntax of the SAS

The length of the SAS is a compromise between the capacity to resit
MITM attacks and a the burden placed on the users performing the
comparison. We chose the number 7 based on a "well known" psychology
result that "the longest sequence a normal person can recall on the
fly contains about seven items" [Number7]. We chose decimal numbers
because they are a least common denominator, and to not require
translations in differnt languages.

A simple way to obtain a 7 digit is to print a binary number 20 to 23
bits long. In the design, we will use 20 bits because 7 is in fact
an upper limit. 20 bits print on 7 digits about 5% of the time, on 6
digits about 86% of the time, and on 5 digits or fewer less than 10%
of the time.

3.3. Authentication

Once the agreement has been performed, each device displays the
"real" name of the other device, as exchanged through the pairing
protocol. Each user verifies that the name matches what they expect.
If the names are validated, each user approves the pairing, which
will then be remembered.

3.4. Intra User Pairing

Users can pair their own devices in secure (home) networks without
any interaction using a special DNS-SD pairing service. Verification
methods where a single user holds both devices, e.g. synchronously
pressing buttons on both devices a few times, are also suitable.
Further, a secure OOB could be established by connecting two devices
with an USB channel. Pairing via an USB connection is also used by
some Bluetooth devices, e.g. when pairing a controller with a gaming
console.

[[TODO: elaborate]]

3.5. Pairing Data Synchronization

To make it sufficient for users to pair only one of their devices to
one of their friends devices while still being able to engage in
later communication with all of this friend's devices using any of
the own devices, we offer the possibility to synchronize pairing data
among devices of the same user. Pairing data synchronization is
performed via a special DNS-SD service (_pdsync._tls).

[[TODO: elaborate]]

3.6. Public Authentication Keys

[[TODO: Should we discuss public authentication keys whose
fingerprints are verified during pairing?]]

4. Solution

[[TODO: elaborate on all subsections]]

In the proposed pairing protocol, one of the devices acts as a
"server", and the other acts as a "client". The server will publish
a "pairing service". The client will discover the service instance
during the discovery phase, as explained in Section 4.1. The pairing
service itself is specified in Section 4.2.

4.1. Discovery

The discovery uses DNS-SD [RFC6763] over mDNS [RFC6762]. The pairing
service is identified in DNS SD as "_pairing._tcp".

When the pairing service starts, the server starts publishing the
chosen instance name. The client will discover that name and the
corresponding connection parameters.

4.2. Agreement and Authentication

The pairing protocol is built using TLS. The following description
uses the presentation language defined in section 4 of [RFC5246].
The protocol uses five message types, defined in the following enum:

```
enum {
    ClientHash(1),
    ServerRandom(2),
    ClientRandom(3),
    ServerSuccess(4),
    ClientSuccess(5)
} PairingMessageType;
```

Devices implementing the service MUST support TLS 1.2 [RFC5246], and
SHOULD support TLS 1.3 as soon as it becomes available. When using
TLS, the client and server MUST negotiate a ciphersuite providing
forward secrecy (PFS), and strong encryption (256 bits symmetric
key). All implementations using TLS 1.2 SHOULD be able to negotiate
the cipher suite TLS_DH_anon_WITH_AES_256_CBC_SHA256.

Once the TLS connection has been established, each party extracts the
pairing secret S_p from the connection context per [RFC5705], using
the following parameters:

Disambiguating label string: "PAIRING SECRET"

Context value: empty.

Length value: 32 bytes (256 bits).

Once S_p has been obtained, the client picks a random number R_c,
exactly 32 bytes long. The client then selects a hash algorithm,
which SHOULD be the same algorithm as negotiated for building the PRF
in the TLS connection. If there is no suitable API to retrieve that
algorithm, the client MAY use SHA256 instead. The client then
computes the hash value H_c as:

    H_c = HMAC_hash(S_p, R_c)

Where "HMAC_hash" is the HMAC function constructed with the
selected algorithm.

The client transmits the selected hash function and the computed
value of H_c in the Client Hash message, over the TLS connection:

```
struct {
   PairingMessageType messageType;
   hashAlgorithm hash;
   uint8 hashLength;
   opaque H_c[hashLength];
} ClientHashMessage;
```

messageType Set to "ClientHash".

hash The code of the selected hash algorithm, per definition of
      HashAlgorithm in section 7.4.1.1.1 of [RFC5246].

hashLength The length of the hash H_c, which MUST be consistent with
      the selected algorithm "hash".

H_c   The value of the client hash.

Upon reception of this message, the server stores its value. The
server picks a random number R_s, exactly 32 bytes long, and
transmits it to the client in the server hash message, over the TLS
connection:

```
struct {
   PairingMessageType messageType;
   opaque R_s[32];
} ServerRandomMessage;
```

messageType Set to "ServerRandom".

R_s   The value of the random number chosen by the server.

Upon reception of this message, the client discloses its own random
number by transmitting client random message:

```
struct {
   PairingMessageType messageType;
   opaque R_c[32];
} ClientRandomMessage;
```

messageType Set to "ClientRandom".

R_c   The value of the random number chosen by the client.

Upon reception of this message, the server verifies that the number
R_c hashes to the previously received value H_c. If the number does
not match, the server MUST abandon the pairing attempt and abort the
TLS connection.

At this stage, both client and server can compute the short hash SAS
as:

   SAS = first 20 bits of HMAC_hash(S_p, R_c + R_s)

   Where "HMAC_hash" is the HMAC function constructed with the hash
   algorithm selected by the client in the ClientHashMessage.

Both client and server display the 20 bit value as a decimal integer,
and ask the user to compare the values. If the values do not match,
the user cancels the pairing. Otherwise, the protocol continues with
the exchange of names, both server and client announcing their own
preferred name in a Success message

```
struct {
   PairingMessageType messageType;
   uint8 nameLength;
   opaque name[nameLength];
} ClientSuccessMessage;
```

messageType Set to "ClientSuccess" if transmitted by the client,
      "ServerSuccess" if by the server.

nameLength The length of the string encoding the selected name.

name The selected name of the client or the server, encoded as a
      string of UTF8 characters.

After receiving these messages, client and servers can orderly close
the TLS connection, terminating the pairing exchange.

5. Security Considerations

[[TODO: elaborate]]

There are a variety of attacks against pairing systems. They may
result in compromised pairing keys.

To mitigate such attacks, nodes MUST be able to quickly revoke a
compromised pairing. This is however not sufficient, as the
compromise of the pairing key could remain undetected for a long
time. For further safety, nodes SHOULD assign a time limit to the

validity of pairings, discard the corresponding keys when the time
has passed, and establish new pairings.

The requirement of limiting the Time-To-Live of a pairing can raise
doubts about the usability of the protocol. The usability issues
would be mitigated if the initial pairing provided both a shared
secret and the means to renew that secret over time, e.g. using
authenticated public keys.

6. IANA Considerations

This draft does not require any IANA action.

7. Acknowledgments

TODO

8. References

8.1. 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,
            <http://www.rfc-editor.org/info/rfc2119>.

[RFC5246]   Dierks, T. and E. Rescorla, "The Transport Layer Security
            (TLS) Protocol Version 1.2", RFC 5246,
            DOI 10.17487/RFC5246, August 2008,
            <http://www.rfc-editor.org/info/rfc5246>.

[RFC5705]   Rescorla, E., "Keying Material Exporters for Transport
            Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705,
            March 2010, <http://www.rfc-editor.org/info/rfc5705>.

[RFC6762]   Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762,
            DOI 10.17487/RFC6762, February 2013,
            <http://www.rfc-editor.org/info/rfc6762>.

[RFC6763]   Cheshire, S. and M. Krochmal, "DNS-Based Service
            Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013,
            <http://www.rfc-editor.org/info/rfc6763>.

8.2. Informative References

   [BTLEPairing]
              Bluetooth SIG, "Bluetooth Low Energy Security Overview",
              2016,
              <https://developer.bluetooth.org/TechnologyOverview/Pages/
              LE-Security.aspx>.

   [I-D.huitema-dnssd-privacy]
              Huitema, C. and D. Kaiser, "Privacy Extensions for DNS-
              SD", draft-huitema-dnssd-privacy-01 (work in progress),
              June 2016.

   [I-D.miers-tls-sas]
              Miers, I., Green, M., and E. Rescorla, "Short
              Authentication Strings for TLS", draft-miers-tls-sas-00
              (work in progress), February 2014.

   [KFR09]   Kainda, R., Flechais, I., and A. Roscoe, "Authentication
              protocols based on low-bandwidth unspoofable channels: a
              comparative survey", 2009.

   [NR11]   Nguyen, L. and A. Roscoe, "Authentication protocols based
              on low-bandwidth unspoofable channels: a comparative
              survey", 2011.

   [Number7] Miller, G., "The magical number seven, plus or minus two:
              Some limits on our capacity for processing information",
              1956, <http://psychclassics.yorku.ca/Miller/>.

   [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-
              Hashing for Message Authentication", RFC 2104,
              DOI 10.17487/RFC2104, February 1997,
              <http://www.rfc-editor.org/info/rfc2104>.

   [RFC6151] Turner, S. and L. Chen, "Updated Security Considerations
              for the MD5 Message-Digest and the HMAC-MD5 Algorithms",
              RFC 6151, DOI 10.17487/RFC6151, March 2011,
              <http://www.rfc-editor.org/info/rfc6151>.

   [USK11]   Uzun, E., Saxena, N., and A. Kumar, ". Pairing devices for
              social interactions: a comparative usability evaluation",
              2009.

   [WPS]   Wi-Fi Alliance, "Wi-Fi Protected Setup", 2016,
              <http://www.wi-fi.org/discover-wi-fi/
              wi-fi-protected-setup>.

   [XKCD936] Munroe, R., "XKCD: Password Strength", 2011,
              <https://www.xkcd.com/936/>.

Authors' Addresses

    Christian Huitema
    Microsoft
    Redmond, WA 98052
    U.S.A.

    Email: huitema@huitema.net

    Daniel Kaiser
    University of Konstanz
    Konstanz 78457
    Germany

    Email: daniel.kaiser@uni-konstanz.de