Rumpelstiltskin Talk: Sending Private Data Without Sending it

> For little knows my royal dame
> that Rumpelstiltskin is my name!

This document describes a technique to securely send data via hidden names such that neither data nor the hidden name are actually sent. The tradeoff is that instead of one decoding step, 2^N attempts are needed for recovering N bits.

1) Technique

Cryptographic signatures can be used as filters that let a receiver recover data that was signed but not sent, based on the signature alone. To this end, the sender embeds the data in some context (hidden name) before signing (encoding): Using the same context, the receiver generates all content candidates and picks the one that verifies for the received signature (decoding).

The following pseudo code shows how to confidentially transfer an arbitrary byte value without sending the byte. In this example we assume asymmetric cryptography for generating signatures such that the receiver only needs to know the sender's public key as well as the hidden name (we cover shared-secret signatures further below):

```
Encode byte x:
  send signature(sender_secret_key, "SOMEHIDDENNAME-" + x)

Decode byte x from signature S:
  for i in [0x00 .. 0xff]:
    if verify(sender_public_key, "SOMEHIDDENNAME-" + i, S):
      return i
```

When recovering the intended data, the receiver also asserts the data's authenticity and integrity. A message larger than one byte can be encoded in a signature of unchanged length, but at the expense

of additional trial-and-error steps while encoding one bit only needs two decoding checks in the worst case.

The technique described above is very similar to Rivest's *Chaffing and Winnowing* method of providing confidentiality without encryption by simply using signatures, see [1].

2) Instantiations

Rumpelstiltskin Talk (RT) works both for symmetric and asymmetric signatures. Moreover, RT can be used to protect against replay attacks.

2.a) Symmetric Signatures

In the case that sender and receiver share some secret, using it as a signing key is sufficient for RT and the hidden name can remain empty. Again, only the signature is sent:

```
Encode byte x:
  send signature(shared_secret, x)

Decode content from signature S:
  for i in [0x00 .. 0xff]:
    if verify(shared_secret, i, S):
      return i
```

Literally, when using an HMAC algorithm, the signature becomes an "oblivious hash tag" that only reveals its meaning to those knowing the shared secret.

Using HMAC-MD5 (which still can be used, despite MD5's weakness) specifically, each signature is 20 bytes long. If a sender and receiver own asymmetric key pairs that they mutually trust, they can first use a Diffie-Hellmann key exchange to obtain the shared secret for their subsequent Rumpelstiltskin talking (in order to benefit from the potentially smaller signature).

If a shared secret is used in a bidirectional exchange, messages with the same hidden content will lead to the same signatures. In order to make them different it suffices to add different hidden names, as follows:

```
Alice encodes x:
    signature(shared_secret, 'Alice-to-Bob' + x)

Bob encodes y:
    signature(shared_secret, 'Bob-to-Alice' + y)
```

Finally, in order to hide repetition, a nonce that is part of the hidden name can be sent in clear together with the signature:

```
Encode byte x:
  n = random nonce
  send n + signature(shared_secret, n + x)

Decode content from packet <n, S>:
  for i in [0x00 .. 0xff]:
    if verify(shared_secret, n + i, S):
      return i
```

2.b) Asymmetric Signatures

Asymmetric signatures are considerable larger than the minimal 20 bytes from above. In case of RSA-based signatures, the length of the signatures is the same as the RSA key size (256B for a 2048-bit RSA key). For elliptic curve crypto, signatures are in the range of 64 bytes (or more, depending on their encoding).

Note that the encoding algorithm for shared keys (above) does not work for asymmetric cryptography:

```
Encode byte x:
  signature(sender_secret_key, x)   // do not do this!
```

because everybody knowing the sender's public key would be able to recover x. The use of a hidden name is thus required for asymmetric signatures, and the pseudo code shown in Section 1 must be used.

The hidden name plays the role of a (secret) session key where a sender can tag its content. When running several communication sessions in parallel, the decoding of received signatures becomes more involved (which is true for both the asymmetric and symmetric RT) if only a signature is exchanged (without any additional session- or sender-identifying information), adding two more loops to the decoding process:

```
Decode content AND sender AND session from signature S:

  for src in my_buddy_list:
    for sess in src.session_list:
      for i in [0x00 .. 0xff]:
        if verify(src.public_key, sess.hidden_name + i, S):
          return (sender=src, session=sess, data=i)
```

2.c) Preventing Replay Attacks (= Replicating Append-only Logs)

Rumpelstiltskin messages can be replayed by an attacker at arbitrary future times. In order to prevent such replays, the sender has to form a hidden name that encodes the context when a message was sent. Including a timestamp is not practical because the receiver would have to test too many candidates (namely all discrete time steps since the last successfully received message). However, a logical clock will do:

```
Sending a series of consecutive bytes x:
  pkt = signature(secret_key, "SOMEHIDDENNAME-" + enc(seqno) + x)
        where seqno is incremented after each transmission
```

The use of a logical clock between a sender and receiver is equivalent to replicating an append-only log where the receiver requires each entry to be an extension of the already received log. The following pseudo code describes a receiver that only accepts correctly numbered log entries. The content as well as the sequence number remain invisible to an observer and are not transmitted:

```
Receiving the next (hidden) content of an append-only log:
  for i in [0x00 .. 0xff]:
    if verify(public_secret, "SOMEHIDDENNAME-" + enc(next_seqno) + x), pkt):
      append x to local log replica
      next_seqno += 1
```

3) Reliable Private Transfer Using Rumpelstiltskin Messages

We now sketch how two parties, by exchanging signatures only, can reliably and confidentially transfer a data stream, despite the channel altering, loosing or reordering the packets. Optionally, nonces can be used to make repetition of packets improbable.

To prevent Alice or Bob from cheating about sequence numbers (issuing packets for future numbers, both for sending data but also for acknowledgments), the content can be chained by including the hash of the previous packet. This is also known as *secure append-only log replication* (but is not shown below).

```
case { // sender side (Alice)
  init:
    pos = ack = 0

  if "new data D to send" and pos == ack:
    N = nonce
    pkt = <N, signature(Alice_secret, "HIDDENNAME-DATA" + N + enc(pos) + D)>
    pos += 1
    send pkt
    start timer

  if receive packet <N, S>:
    if verify(Bob_public, "HIDDENNAME-ACK" + N + enc(pos-1), S):
      ack = pos
      cancel timer

  timeout:
    resend pkt
}

case { // receiver side (Bob)
  init:
    pos = -1

  if receive packet <N, S>:
    if verify(Alice_public, "HIDDENNAME-DATA" + N + enc(pos+1), S):
      pos += 1
      deliver recovered content
    M = nonce
    send <M, signature(Bob_secret, "HIDDENNAME-ACK" + M + enc(pos))
}
```

The filtering achieved through verifying the signature automatically discards garbled packet. Also, packets carrying the wrong sequence number are discarded as they do not match our expectations.

This reliable transfer protocol is not very efficient as it re-authenticates the sender with each message. In the following section we will look at various optimization scenarios and settings where Rumpelstiltskin Talk may be a viable option.

4) Application Considerations

Regarding the efficiency concern we observe that for example tcpcrypt [3] adds 38 bytes for every TCP packet that is sent. While most TCP packets carry more than one byte, in case of an encrypted telnet

session, single characters would be exchanged. In such a case, on top of the 38 bytes of tcpcrypt there is the 40 byte overhead of the TCP and IP headers which reveal in full the sender and source addresses. In this light, the 20 bytes of our HMAC-MD5 example per message look really good as they also hide (but nevertheless contain, thanks to the signature) the source identification!

Additional notes, to be sorted:

- not for "open PKI" where you don't know the sender in advance –> predictable communications, you have small social reach
- "unobservable source communications" (stealth sender) e.g. hybrid where only sender is hidden but payload is in the clear, this is Rivest's config

  i.e. one channel, mux traffic from different stealth sources radio channel, creates "virtual channels" without (HW) nodes having to setup encryption e.g., ethernet
- when bits must be authentic, but want to save transmission costs (i.e., authenticity is a must and transmission is costly) compare FT8 [2] which has 77 bit payloads, all not authenticated, here we hope for co-filtering of symbol recovery and source authentication: if a sequence of decoded symbols does not "hash out", reject it, or try to flip some bits if they are not certain. FT8 does RT for call signs, sends hash instead of call sign
- when a field can grow arbitrarily long (e.g. sequence number) but we want a fixed-size packet.
- symmetric is faster than asymmetric (because only two hash operation per decoding attempt), see note on negotiating a shared secret in the "symmetric" section for bootstrapping symmetric mode
- why not use (symmetric) encryption instead symmetric RT, once a shared secret has been established? Yes, filter now is the (AES-) decryption, can run in parallel with RT traffic. And again the field size concern.
- show (a) DTN, X3DH and double-ratchet, (b) monadic programming style of log replication
- DoS: use packets with two or more filters e.g.,
  - one on source ID, can be partial trunc(hash("hiddenname-source_pk"))
  - the real RT data, for authenticity

  pkt = unauthenticated-RT | (authenticated-RT or encrypted-data)
- prehash: in case of symmetric: jumptable, not redo every verification at packet reception time, combine jump tables from each "session" and jump based on the hash of the received signature hashpower today is enormous (GPU)

Literature

[1] R.L. Rivest: *Chaffing and Winnowing: Confidentiality without Encryption*, Mar 1998, https://people.csail.mit.edu/rivest/pubs/Riv98a.prepub.txt

[2] S. Franke, B. Somerville and J. Taylor: *The FT4 and FT8 Communication Protocols*, Jun 2020, https://www.physics.princeton.edu/pulsar/K1JT/FT4_FT8_QEX.pdf

[3] A. Bittau et al: *Cryptographic Protection of TCP Streams (tcpcrypt)*, 2010-2019, https://www.rfc-editor.org/rfc/rfc8548.txt

---

christian.tschudin@unibas.ch, March 2022

Appendix: Python Demo Code

```
#!/usr/bin/env python3

# rumpelstiltskin-demo.py, March 2022, <christian.tschudin@unibas.ch>
# For little knows my royal dame / that Rumpelstiltskin is my name!

import os
import sys

print("Demo of 'Rumpelstiltskin Talk' (sending data without sending it)")
print(f"usage: {sys.argv[0]} [-symmetric]")

try:
    import nacl.signing
except:
    sys.argv[1:] = ['-symmetric']  # force symmetric if no ed25519 support

if ''.join(sys.argv[1:]) == '-symmetric':
    import hmac
    pk = os.urandom(16)                      # shared secret
    mksign = lambda m: hmac.new(pk, m, digestmod='md5').digest()
    verify = lambda k,m,s: hmac.compare_digest(s, hmac.new(k, m, digestmod='md5').digest())
    print("  signature algorithm is HMAC-MD5")
else: # for ed25519
    sk = nacl.signing.SigningKey.generate()  # key pair
    pk = sk.verify_key._key
    mksign = lambda m: sk.sign(m)[:64]
    def verify(pk,m,s):
        try:
            nacl.signing.VerifyKey(pk).verify(m,s)
        except nacl.exceptions.BadSignatureError:
            return False
        return True
    print("  signature algorithm is ed25519")

# demo ----------------------------------------------------------------------

hin = os.urandom(42)         # hidden name is shared secret, recv must know it
                             # (this is the Rumpelstiltskin Name)
msg = os.urandom(1)          # content (1 Byte), random for demo purposes
pkt = mksign(hin + msg)      # pkt creation: signature IS the packet

print(f"  message to transfer: 0x{msg.hex()}")
print(f"  the signature IS the packet: 0x{pkt.hex()} ({len(pkt)} bytes)")

for i in range(256):         # decoding: exhaustive search
    b = bytes([i])           # candidate content
    if verify(pk, hin + b, pkt): # does data fit the received pkt/signature?
        print(f"  reconstructed message: 0x{b.hex()}")
        break
else:
    print("  no decoding found :-(")

# eof
```