Noise Explorer: N

May 8, 2019

1 Message A

1.1 Message Pattern Analysis

Message `A` is the first message in the `N` Noise Handshake Pattern. It is sent from the initiator to the responder. In this detailed analysis, we attempt to give you some insight into the protocol logic underlying this message. The insight given here does not fully extend down to fully illustrate the exact state transformations conducted by the formal model, but it does describe them at least informally in order to help illustrate how Message `A` affects the protocol.

1.1.1 Sending Message A

In the applied pi calculus, the initiator's process prepares Message `A` using the following function:

```
letfun writeMessage_a(me:principal, them:principal, hs:handshakestate, payload:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty, empty,
            ↪ empty) in
        let e = generate_keypair(key_e(me, them, sid)) in
        let ne = key2bit(getpublickey(e)) in
        let ss = mixHash(ss, ne) in
        (* No PSK, so skipping mixKey *)
        let ss = mixKey(ss, dh(e, rs)) in
        let (ss:symmetricstate, ciphertext:bitstring) = encryptAndHash(ss,
            ↪ payload) in
        let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
        let message_buffer = concat3(ne, ns, ciphertext) in
        let (ssi:symmetricstate, cs1:cipherstate, cs2:cipherstate) = split(ss)
            ↪ in
        (hs, message_buffer, cs1, cs2).
```

How each token is processed by the initiator:

- `e`: Signals that the initiator is sending a fresh ephemeral key share as part of this message. This token adds the following state transformations to `writeMessage_a`:

    - `mixHash`, which hashes the new key into the session hash.

- **es**: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds the following state transformations to `writeMessage_a`:

  - **mixKey**, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(e, rs)`, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.

If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `A`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- **encryptAndHash**, which performs an authenticated encryption with added data (AEAD) on the payload, with the session hash as the added data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

1.1.2 Receiving Message A

In the applied pi calculus, the initiator's process prepares Message `A` using the following function:

```
letfun readMessage_a(me: principal, them: principal, hs: handshakestate, message:
    ↪ bitstring, sid: sessionid) =
        let (ss: symmetricstate, s: keypair, e: keypair, rs: key, re: key, psk: key,
            ↪ initiator: bool) = handshakestateunpack(hs) in
        let (ne: bitstring, ns: bitstring, ciphertext: bitstring) = deconcat3(
            ↪ message) in
        let valid1 = true in
        let re = bit2key(ne) in
        let ss = mixHash(ss, key2bit(re)) in
        (* No PSK, so skipping mixKey *)
        let ss = mixKey(ss, dh(s, re)) in
        let (ss: symmetricstate, plaintext: bitstring, valid2: bool) =
            ↪ decryptAndHash(ss, ciphertext) in
        if ((valid1 && valid2)) then (
                let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
                let (ssi: symmetricstate, cs1: cipherstate, cs2: cipherstate) =
                    ↪ split(ss) in
                (hs, plaintext, true, cs1, cs2)
        ).
```

How each token is processed by the responder:

- **e**: Signals that the responder is receiving a fresh ephemeral key share as part of this message. This token adds the following state transformations to `readMessage_a`:

  - **mixHash**, which hashes the new key into the session hash.

- **es**: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds the following state transformations to `readMessage_a`:

  - **mixKey**, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(e, rs)`, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.

If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `A`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- **decryptAndHash**, which performs an authenticated decryption with added data (AEAD) on the payload, with the session hash as the added data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

1.1.3 Queries and Results

Message `A` is tested against four authentication queries and five confidentiality queries.

Authentication Grade 1: Failed

RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) $\implies$ event(SendMsg(alice,
  $\hookrightarrow$ c_643,stagepack_a(sid_a),m)) || event(LeakS(phase0,alice)) || event(LeakS(
  $\hookrightarrow$ phase0,bob)) cannot be proved.

In this query, we test for *sender authentication* and *message integrity*. If Bob receives a valid message from Alice, then Alice must have sent that message to *someone*, or Alice had their static key compromised before the session began, or Bob had their static key compromised before the session began.

Authentication Grade 2: Failed

RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) $\implies$ event(SendMsg(alice,
  $\hookrightarrow$ c_643,stagepack_a(sid_a),m)) || event(LeakS(phase0,alice)) cannot be
  $\hookrightarrow$ proved.

In this query, we test for *sender authentication* and is *Key Compromise Impersonation* resistance. If Bob receives a valid message from Alice, then Alice must have sent that message to *someone*, or Alice had their static key compromised before the session began.

Authentication Grade 3: Failed

RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) $\implies$ event(SendMsg(alice,
  $\hookrightarrow$ bob,stagepack_a(sid_a),m)) || event(LeakS(phase0,alice)) || event(LeakS(
  $\hookrightarrow$ phase0,bob)) cannot be proved.

In this query, we test for *sender and receiver authentication* and *message integrity*. If Bob receives a valid message from Alice, then Alice must have sent that message to *Bob specifically*, or Alice had their static key compromised before the session began, or Bob had their static key compromised before the session began.

Authentication Grade 4: Failed

RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) ⟹ event(SendMsg(alice,
↪ bob,stagepack_a(sid_a),m)) || event(LeakS(phase0,alice)) cannot be proved.

In this query, we test for *sender and receiver authentication* and is *Key Compromise Impersonation* resistance. If Bob receives a valid message from Alice, then Alice must have sent that message to *Bob specifically*, or Alice had their static key compromised before the session began.

Confidentiality Grade 1: Passed

RESULT attacker_p1(msg_a(alice,bob,sid_a)) ⟹ event(LeakS(phase0,bob)) || event
↪ (LeakS(phase1,bob)) **is true**.

In this query, we test for *message secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Bob's static key either before or after the protocol session.

Confidentiality Grade 2: Passed

RESULT attacker_p1(msg_a(alice,bob,sid_a)) ⟹ event(LeakS(phase0,bob)) || event
↪ (LeakS(phase1,bob)) **is true**.

In this query, we test for *message secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key either before or after the protocol session.

Confidentiality Grade 3: Failed

RESULT attacker_p1(msg_a(alice,bob,sid_a)) ⟹ event(LeakS(phase0,bob)) || (
↪ event(LeakS(px,bob)) && event(LeakS(pz,alice))) cannot be proved.

In this query, we test for *forward secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Bob's static key before the protocol session, or after the protocol session along with Alice's static public key (at any time.)

Confidentiality Grade 4: Failed

RESULT attacker_p1(msg_a(alice,bob,sid_a)) ⟹ event(LeakS(phase0,bob)) || (
↪ event(LeakS(px,bob)) && event(LeakS(pz,alice))) cannot be proved.

In this query, we test for *weak forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key before the protocol session, or after the protocol session along with Alice's static public key (at any time.)

Confidentiality Grade 5: Failed

RESULT attacker_p1(msg_a(alice,bob,sid_a)) ⟹ event(LeakS(phase0,bob)) cannot
↪ be proved.

In this query, we test for *strong forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key before the protocol session.