Symbolic Software • Formal Analysis Report

SaltChannel v2: Formal verification for a simple, lightweight secure channel protocol in the symbolic model

Summary

SaltChannel[1] is a simple, lightweight secure channel protocol based on the TweetNaCL API developed by Bernstein et al. Its goal is to provide some level of transport layer security, but with a much smaller overhead cost, state machine complexity and implementation complexity.

In this report, we formally analyze SaltChannel (version 2) in the symbolic model to ensure that it meets specified security goals. We provide three symbolic models, written in the applied pi calculus, illustrating different SaltChannel deployment scenarios. We present the results of our formal model analysis with ProVerif[2], and a security assessment of SaltChannel as a transport layer security protocol.

Security Goals

SaltChannel claims the following security goals:

1. 128-bit security. The best attack should be a 2^{128} brute force attack. No attack should be feasible until there are (if there ever will be) large-enough quantum computers.
2. Internet-capable. The protocol should protect against all attacks that can occur on public communication channels. The attacker can read, modify, redirect all packages sent between any pair of peers. The attacker has access to every Salt Channel package ever sent and packages from all currently active Salt Channel sessions world-wide.
3. Forward secrecy. Recorded communication will not be possible to decrypt even if one or both peer's private keys are compromised.
4. Delay attack protection. The protocol should protect against delay attacks. If the attacker delays a package, the receiving peer should detect this if the delay is abnormal.
5. Secret client identity. An active or passive attacker cannot retrieve the long-term public key of the client. Tracking of the client is impossible.

Since our formal analysis is carried out in the symbolic model, where encryption operations are assumed to be non-algebraic, perfect black-box implementations, **Goal 1** is obtained through the modeling of cryptographic primitives as simply not vulnerable to any sort of attack: encryption functions are perfect PRPs (pseudorandom permutation functions) with authentication tags, while hash functions are perfect one-way injective maps.

Goal 2 is obtained in ProVerif by running the analysis under an active Dolev-Yao attacker model[3].

[1] https://github.com/assaabloy-ppi/salt-channel
[2] http://prosecco.gforge.inria.fr/personal/bblanche/proverif/
[3] http://www.cs.huji.ac.il/~dolev/pubs/dolev-yao-ieee-01056650.pdf

Goal 3 is modeled through authenticity and post-compromise security queries which are described in more detail further in this report.

Goal 4 is not modeled, since it is difficult to capture the notion of time appropriately within a symbolic model (and this comes with diminishing returns.)

Goal 5 is modeled via a confidentiality query with an event correlation, discussed in more detail further in this report.

Protocol Description

In SaltChannel, the **client** and the **server** each have a long-term key (used for signatures) and a session-specific ephemeral key used for encryption. Session flow goes as follows:

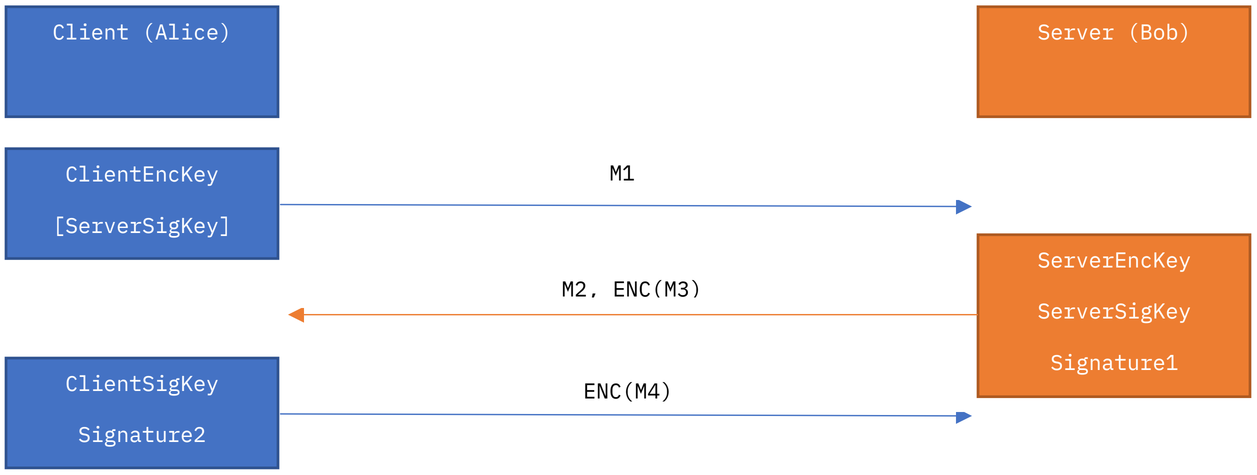

Figure 1: *SaltChannel v2 session establishment overview.*

```
Signature1 = sign("SC-SIG01" || hash(M1) || hash(M2))
Signature2 = sign("SC-SIG02" || hash(M1) || hash(M2))
```

Figure 2: *Signature values, explained.*

As seen in Figure 1, M1 only optionally specifies a *ServerSigKey* (the server's long term public key.) Partly due to this, we actually analyze three different models of the Protocol:

- **SaltChannel.pv**: this is the naïve protocol run in which providing *ServerSigKey* is optional.
- **SaltChannelServerAuth.pv**: here, providing *ServerSigKey* is not only mandatory, but the server is pre-authenticated to the client out of band.
- **SaltChannelFullAuth.pv**: the strongest model, in which both the server and the client are mutually pre-authenticated out of band before the session commences.

After the exchange of *M4,* the protocol is considered established. In our modeling, we follow up session establishment with one AppData message from the server (*MsgA*), and a reciprocating AppData message from the client (*MsgB*).

Queries and Results

```
A.  RESULT event(RecvMsgA(alice,bob)) ==> event(SendMsgA(bob,alice)) is false.
B.  RESULT event(RecvMsgB(bob,alice)) ==> event(SendMsgB(alice,bob)) is false.
C.  RESULT not attacker_p1(msg_a(bob,alice)) is false.
D.  RESULT not attacker_p2(msg_b(alice,bob)) is false.
E.  RESULT not attacker_p3(msg_a(bob,alice)) cannot be proved.
F.  RESULT not attacker_p3(msg_b(alice,bob)) is false.
G.  RESULT attacker(sigexp(sigKey(alice))) ==> event(ClientInitialized(alice,charlie)) is
    false.
```

Figure 3: *SaltChannel.pv raw formal verification results.*

```
A.  RESULT event(RecvMsgA(alice,bob)) ==> event(SendMsgA(bob,alice)) is false.
B.  RESULT event(RecvMsgB(bob,alice)) ==> event(SendMsgB(alice,bob)) is false.
C.  RESULT not attacker_p1(msg_a(bob,alice)) is false.
D.  RESULT not attacker_p2(msg_b(alice,bob)) is true.
E.  RESULT not attacker_p3(msg_a(bob,alice)) cannot be proved.
F.  RESULT not attacker_p3(msg_b(alice,bob)) is true.
G.  RESULT attacker(sigexp(sigKey(alice))) ==> event(ClientInitialized(alice,charlie)) is
    true.
```

Figure 4: *SaltChannelServerAuth.pv raw formal verification results.*

```
A.  RESULT event(RecvMsgA(alice,bob)) ==> event(SendMsgA(bob,alice)) is true.
B.  RESULT event(RecvMsgB(bob,alice)) ==> event(SendMsgB(alice,bob)) is true.
C.  RESULT not attacker_p1(msg_a(bob,alice)) is true.
D.  RESULT not attacker_p2(msg_b(alice,bob)) is true.
E.  RESULT not attacker_p3(msg_a(bob,alice)) is true.
F.  RESULT not attacker_p3(msg_b(alice,bob)) is true.
G.  RESULT attacker(sigexp(sigKey(alice))) ==> event(ClientInitialized(alice,charlie)) is
    true.
```

Figure 5: *SaltChannelFullAuth.pv raw formal verification results.*

In our analysis, we adopt a model with an active attacker monitors and tampers with the interactions of three principals: Alice (the client), Bob (the server) and Charlie.

Charlie is a compromised principal, controlled by the attacker, that Alice initializes a distinct, separate session with where Alice is a client and Charlie is the server. Bob also initializes a separate, distinct session with Charlie where Bob is a server and Charlie is a client.

Queries are the same across all three models (as seen in Figures 3, 4 and 5.) We explain the queries thus:

- **Queries A and B (authenticity)**: We query the model to verify that Alice may receive *MsgA* from Bob if and only if Bob has indeed sent *MsgA* to alice. Similarly, we attempt to verify that Bob may receive *MsgB* from Alice if and only if Alice has sent *MsgB* to Bob.
- **Queries C and D (confidentiality)**: We query the model to verify that the attacker cannot access the plaintexts for *MsgA* and *MsgB* if Alice and Bob's long-term secret keys are not leaked.
- **Queries E and F (forward secrecy)**: We query the model to verify that the attacker cannot access the plaintexts for *MsgA* and *MsgB* if Alice and Bob's long-term secret keys are leaked after the session is concluded.
- **Query G (secret client identity)**: We query the model to verify that the attacker can only obtain Alice's client long-term public key if and only if the attacker has initialized a session as Charlie, where Alice is a client and Charlie is a server.

Results are as follows: potentially unexpected and dangerous results that we consider "attacks" are in **red**, whereas results that indicate a weak protocol but within reasonable expectations are not specially outlined in any way.

- **Queries A and B (authenticity)**:
  - **SaltChannel.pv (Figure 3)**: In the scenario where the client and the server are not mutually authenticated to one another, an active attacker obtains considerable power and can man in the middle sessions with relative ease.
  - **SaltChannelServerAuth.pv (Figure 4)**: In the scenario where the server is authenticated to the client, we see that authenticity is still not fully obtained. When Alice receives an initial AppData *MsgA*, they cannot authenticate that it indeed was intended for them despite them being able to authenticate that it came from Bob: Bob could have sent that message to an attacker impersonating Alice by erroneously advertising their public keys as equaling Alice's. This is known as an unknown key share attack but is not considered a serious issue. And since Alice is unauthenticated to Bob, *MsgB*'s provenance cannot be authenticated for obvious reasons.
  - **SaltChannelFullAuth.pv (Figure 5)**: In the scenario where both parties are mutually authenticated to one another, authenticity is obtained.
- **Queries C and D (confidentiality)**:
  - **SaltChannel.pv (Figure 3)**: In the scenario where the client and the server are not mutually authenticated to one another, an active attacker obtains considerable power and can man in the middle sessions with relative ease.
  - **SaltChannelServerAuth.pv (Figure 4)**: In the scenario where the server is authenticated to the client, we see that confidentiality is only obtained for *MsgB*, which is sent from Alice to Bob. This is because a lack of client authentication allows the attacker to impersonate Alice and to trick Bob into sending *MsgA* to a compromised principal.
  - **SaltChannelFullAuth.pv (Figure 5)**: In the scenario where both parties are mutually authenticated to one another, confidentiality is obtained.

- **Queries E and F (forward secrecy)**: Since long-term keys are only used for signing, the post-session compromise of long-term keys does not lead to any additional decryption capability by the attacker. Therefore, these results completely match the results of **Queries C and D** with no differences.
- **Queries G (secret client identity)**:
    - **SaltChannel.pv (Figure 3)**: In the scenario where the client and the server are not mutually authenticated to one another, an active attacker **may obtain Alice's client identity by impersonating a server**. This can be remedied by ensuring that only legitimate servers are queried by Alice in real-world systems. Formally, however, the only valid assurance is to ensure server pre-authentication in all circumstances.
    - **SaltChannelServerAuth.pv (Figure 4)**: In the scenario where the server is authenticated to the client, secret client identity is obtained.
    - **SaltChannelFullAuth.pv (Figure 5)**: In the scenario where both parties are mutually authenticated to one another, secret client identity is obtained.

Conclusion

In this review, we have offered a comprehensive formal modeling and verification of the SaltChannel v2 protocol. We discovered two minor attacks in certain protocol execution circumstances and helped shed more light on what can reasonably be expected from this protocol. This report comes with the full ProVerif models as well as accessibility tools to re-run and re-verify the results locally.

By simply enforcing pre-authentication, SaltChannel immediately becomes a competitive protocol that meets strong security goals while maintaining a relatively miniscule engineering footprint. Implementing different, mirrored symmetric encryption keys between the sender and the participant would also greatly increase security.

We sincerely thank ASSA and ASSURED for their patience as this report was prepared and the formal verification results were elaborated, and for the excellent working relationship that was established more generally.