Analysis of the SILC Protocol with Murphi

Overview

- What is SILC?
  - Stands for **S**ecure **I**nternet **L**ive **C**onferencing.
  - Designed as a secure replacement for IRC (**I**nternet **R**elay **C**hat).
  - Also has some features of instant messaging.
  - Stable implementations for clients and servers are available. (http://www.silcnet.org)

Project objectives

- Examine the security of SILC, and hopefully find attacks with Murphi.
- More specifically, we wanted to see if a malicious client can "eavesdrop" on a conversation in a channel to which he does not belong.

Results

- Used rational reconstruction to verify the necessity of key part of the chat protocol.
- Found a possible non-trivial attack.
- Bad news: Murphi didn't find it; we thought it up while fine-tuning our invariants. (It turned out that the invariant broke because of a bug in our code and not because of the exploit.)
- Good news: Murphi verifies the exploit.

Presentation outline

- The SILC channel protocol
- Our model of the protocol
- Rational reconstruction of the model
- The exploit
- Problems we encountered
- Future work

Terminology

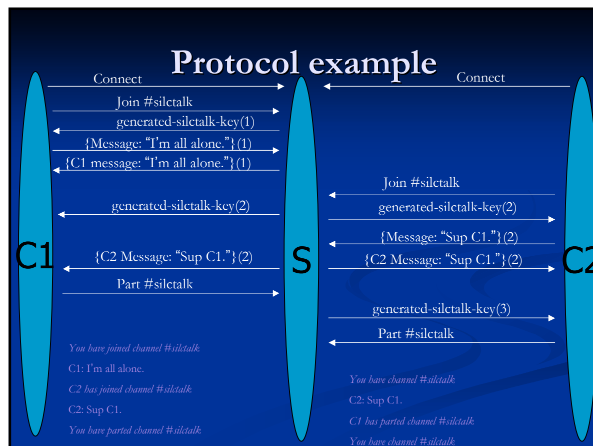
- A **server** handles channel maintenance and accepts connections from clients.
- A **client** connects to a server to join and part channels.
- A **channel** is a group of clients that are in the same conversation.
- No one outside a channel is supposed to be able to listen in on the conversation.
- It is assumed that each client has already established a session key with each server to which it talks.

Protocol description (Client)

- If entity A sends something to entity B in SILC, it is always encrypted with the session key between A and B.
- A client initially connects to a server.
- A connected client can request to join a channel on a server.
- The client knows that it has joined the channel when it receives a channel key from the server.
- Every time a client joins or parts a channel, a new channel key is generated and distributed among the remaining channel members.
- Each channel message, instead of being with the session key, is encrypted with the channel key. However, the packet header (which stores the source and destination) is still encrypted with the session key.
- A client, when it parts a channel, notifies the server so that it may update the channel roster and regenerate the channel key.

Protocol description (Server)

- A server, when it receives a join request for a channel from a client, adds that client to the channel roster if it is not already there.
- A server, when it receives a part request for a channel from a client, removes that client from the channel roster if it is there.
- If the channel roster changes, a new session key is created and distributed to all remaining clients in the channel roster.
- Whenever a message for a channel is received from a client of which it is a member, it is broadcast to all clients in the channel roster. (Only the header is reencrypted.)



Simplifications

- We assume no packet loss.
- We assume lag-free connections.
- In other words, as soon as a client joins or parts a channel, the new key is instantly distributed to all other clients (unless intercepted by an intruder).
- In practice, clients keep around old keys so that they may still decrypt messages that have been delayed, but we don't model that.
- Perfect cryptography and key exchange.

Intruder model

- Intruder can intercept packets and store them.
- Intruder can then forward packets it has stored.
- Intruder may have a partner client and/or a partner server.
- If a client/server is a partner of an intruder it is malicious.
- Intruder cannot directly decrypt packets, but it can pass it on to its partner(s), which may be able to decrypt it.

Murphi implementation (Command)

```
Command : record
   source:    AgentId;
   dest:      AgentId;
   intDest:   AgentId; -- intended destination
 -- (source, intDest) is the key

   cType:     CommandType;
 -- C_Join, C_Part, C_Msg, C_NewChannelKey

   channel:    ChannelId; -- all msg types
   channelKey: KeyId;     -- NewKey, Msg
   message:    MsgId;     -- Msg only
end;
```

Murphi Implementation (Client)

```
Client : record
   partnerServer:  ServerId;
   numMsgs:      MsgId;
   lastSeenMsg:   Command;
   wtjChannels:   multiset[NumChannels]                of
   ChannelId;
   channelRecords: multiset[NumChannels]               of
   ChannelRecord;
 -- record contains channel ID, joined boolean and
 -- channel key
   messagesSent:   multiset[NumMessages]               of
   Command;

  partnerIntruder: IntruderId;
end;
```

Murphi Implementation (Server)

```
Server : record
   channels: array[ChannelId] of
   ChannelRoster;
end;

ChannelRoster : record
   channelKey: ChannelKeyId;
   clients:   array[AgentId] of boolean;
  -- should be ClientId, but Murphi
  -- complains
end;
```

Murphi Implementation (intruder)

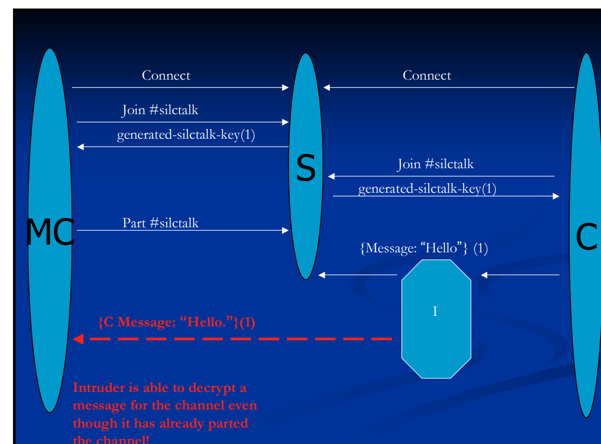
```
Intruder: record
  partnerClient: ClientId;
  partnerServer: ServerId;
  messages:    multiset[NumIntruderMessages] of Command;
End;
```

Invariants

- If the client thinks that it is joined to a channel, the server also thinks that the client is joined to that channel.
- If the server thinks that a particular client is not joined to a channel, then that client also thinks that it is not joined to that channel.
- If the client receives a message, the source of the message must have sent it. (No spoofing)
- If the client receives a message that it has the channel key for, the client must be currently in to that channel, or the message was sent while the client was in the channel. (No eavesdropping)

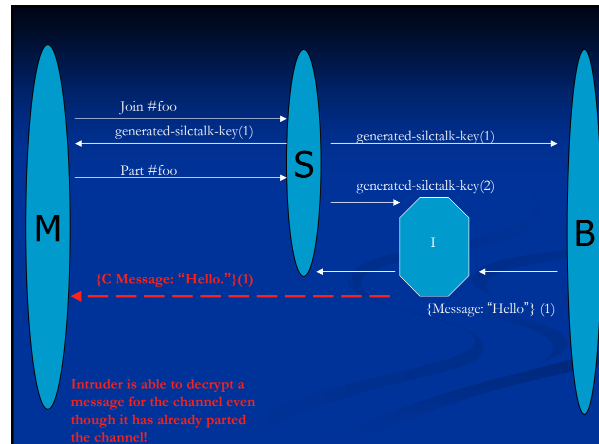
Rational Reconstruction

- We tried removing the part of the SILC protocol where a new channel key is generated every time a client joins or parts a channel from our Murphi model.
- Eavesdropping invariant breaks, as it should.
- Malicious client joins, gets the key for the channel, and parts.
- Malicious client can read any future message sent on that channel that is intercepted by its partner intruder.
- Murphi finds it within 19 states, 20 rules (DFS).



The exploit (as found by Murphi)

- Bob is in channel #foo.
- Murphy joins #foo, and key K1 is sent to Bob and Murphy.
- Murphy parts #foo and server tries to send key K2 to Bob.
- Intruder blocks key message. Bob sends a message with K1, intruder intercepts and passes it to Murphy, who can read it.
- Murphi finds it within 344 states, 543 rules (0.60s).



Practical?

- Bob may not have seen Murphi leave, so might still keep silent.
- Even if Bob saw Murphi leave, he could realize that he didn't receive a new key from the server yet, so may keep silent.

A more practical exploit

- Alice and Bob are in #foo.
- Murphy joins #foo. Server sends K1 to Murphi and tries to send K1 to Alice and Bob but intruder intercepts and stores.
- Murphy parts #foo. Server tries to send K2 to Alice and Bob but intruder intercepts and stores.
- Intruder forwards K2 and K1 to Alice and Bob **in that order**.
- Alice and Bob mistakenly think K1 is the most recent key from the server, and thus will use it to encrypt their messages.
- Intruder can intercept said messages and forward to Murphy to decrypt.
- Alice and Bob saw Murphy join and part, and they both received two keys, so they think everything is fine.

Why does the exploit exist?

- No timestamping or numbering of keys.
- No mention of timestamping or numbering in SILC spec.
- Why even use channel keys? Why not just encrypt using session keys?
- Generality; SILC supports a "private channel" mode, where even the server cannot decrypt the channel messages.
- Disallow messages encrypted with old key?
- Impractical; due to lag, clients may send messages encrypted with old key, and dropping those when someone joins or parts is unacceptable.
- In fact, disallowing messages encrypted with old key turns this exploit into a DOS attack.
- Found and verified exploit only last night, so haven't yet contacted SILC people.

Difficulties

- Most of our difficulties arose from the fact that a server could have multiple clients.
- Had to use arrays instead of multisets (which causes the number of states to explode)
- Difficult to model network—more specifically, to model the sequential guarantees of TCP.
- Forced to serialize everything.
- Murphi doesn't find exploits with BFS! (Error on our part?)
- Ran into possible Murphi bugs?
- Modeling in Murphi forced us to adapt to its idiosyncracies, and seemingly trivial changes to the model changed runtime/correctness drastically; programming in Murphi is "brittle."
- Known problem; see "Source-Level Transformations for Improved Formal Verification" (Winters, Hu)
- http://www.cs.ubc.ca/~bwinters/docs.and.publs/winters.msc.thesis.pdf
- A novice user will model a system in a different manner—semantically equivalent, but less efficient for the verification tool—than an expert user would.

Future work

- Explore other possible models—strand space (Lecture 12), PRISM (Lecture 7).
- Either seems to lead to a more intuitive model.
- However, whether either can model multiple clients/single server is unclear.

Conclusion

- Murphi confirms the necessity of channel key generation part of the examined protocol.
- However, Murphi finds a new (?) attack anyway.
- Murphi was not the ideal tool for this protocol; however, whether a better tool exists is unclear.

Fin