Input languages: messages & sessions

- Handling inputs involves language of input messages

- Often it also involves language of sessions, ie. sequences of messages

- Do LangSec principles also apply at this session level?  
  - when it comes to specification & implementation?
Session language as *message sequence chart*

This *oversimplifies* the session language because it only specifies *one correct, happy flow*.
Session language as *protocol state machine*

This *still oversimplifies*: an implementation will have to be *input-enabled*, ie in every state every message may be received.

SSH transport layer
typical \textit{input enabled} state machine
Security flaws due to broken state machines

• **MIDPSSH**
  
  Open source Java implementation of SSH for Java feature phones
  
  *No protocol state machine implemented at all.*

  [Erik Poll at al., Verifying an implementation of SSH, WITS 2007]

• **e.dentifier2**
  
  USB-connected device for internet banking
  
  *Strange sequence of USB commands by-passes user OK*

  [Arjan Blom et al, Designed to Fail:....., NordSec 2012]

• **TLS**
  
  *Flawed state machines in many TLS implementations* - more to come

“Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message, it MUST NOT send any messages other than:

- Transport layer generic messages (1 to 19) (but SSH_MSG_SERVICE_REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);
- Algorithm negotiation messages (20 to 29) (but further SSH_MSG_KEXINIT messages MUST NOT be sent);
- Specific key exchange method messages (30 to 49).

The provisions of Section 11 apply to unrecognised messages”

...“An implementation MUST respond to all unrecognised messages with an SSH_MSG_UNIMPLEMENTED. Such messages MUST be otherwise ignored. Later protocol versions may define other meanings for these message types.”

**Understanding state machine from prose is hard!**
Typical implementation: **openssh**
**Typical implementation: openssh 😞**

```c
/** This array contains functions to handle protocol messages.
 * The type of the message is an index in this array. */
dispatch_fn *dispatch[255];
```

```c
...
server_init_dispatch_20(void) {
    dispatch_init(&dispatch_protocol_error);
    dispatch_set(SSH_MSG_CHANNEL_CLOSE, &channel_input_oclose);
    dispatch_set(SSH_MSG_CHANNEL_DATA, &channel_input_data);
    dispatch_set(SSH_MSG_CHANNEL_EOF, &channel_input_ieof);
    dispatch_set(SSH_MSG_CHANNEL_EXTENDED_DATA, &channel_input_extended);
    dispatch_set(SSH_MSG_CHANNEL_OPEN, &server_input_channel_open);
    dispatch_set(SSH_MSG_CHANNEL_OPEN_FAILURE, &channel_input_open_failure);
    dispatch_set(SSH_MSG_CHANNEL_REQUEST, &server_input_channel_req);
    dispatch_set(SSH_MSG_GLOBAL_REQUEST, &server_input_global_request);
    dispatch_set(SSH_MSG_KEXINIT, &kex_input_kexinit);
}
```

*Understanding protocol state machine from code is hard!*

Erik Poll, Joeri de Ruiter, Aleksy Schubert  Protocol state machines & session languages  9
LangSec also for session languages!

Protocol state machines deserve to be explicitly specified
Extracting protocol state machine from code

We can infer a finite state machine from implementation by black box testing using state machine learning
• using L* algorithm, as implemented in eg. LearnLib

This is effectively a form of ‘stateful’ fuzzing using a test harness that sends typical protocol messages

This is a great way to obtain protocol state machine
• without reading specs!
• without reading code!
State machine learning with L*

Basic idea: compare response of a deterministic system to different input sequences, eg.

1. b
2. a ; b

If response is different, then otherwise

The state machine inferred is only an approximation of the system, and only as good as your set of test messages.
Case study: EMV

- Most banking smartcards implement a variant of EMV

- EMV (Europay-Mastercard-Visa) defines set of protocols with *lots* of variants

- Specification in 4 books totalling > 700 pages
State machine learning of card
State machine learning of card

merging arrows with identical response
We found no bugs, but lots of variety between cards.

[Fides Aarts et al., Formal models of bank cards for free, SECTEST 2013]
State machine learning of internet banking device

State machines inferred for flawed & patched device

[Georg Chalupar et al., Automated reverse engineering using Lego, WOOT 2014]

Movie at http://tinyurl/legolearn
Scary state machine complexity

More complete state machine of the patched device, using a richer input alphabet

No flaws found in patched device, but were the developers really confident that this complex behaviour is secure? Or necessary?
Comforting to see this is so simple!
TLS state machine extracted from GnuTLS
TLS state machine extracted from OpenSSL
TLS state machine extracted from JSSE
Which TLS implementations are correct? or secure?

[Joeri de Ruiter et al., Protocol state fuzzing of TLS implementations, Usenix Security 2015]
Conclusions

LangSec principles not only apply to language of *input messages* but also for language of *protocol sessions* because in practice we see

- *unclear specifications* of session languages without explicit state machines
- *messy & flawed implementations* of session languages
- *security flaws* as a result of this

*Open question: How common is this category of security flaws?*
Comparing session languages to message formats

Bad news

1. even less likely to be rigorously specified
   • many specs provide EBNF but no protocol state machine
2. complete specification of state machine is tricky
   • input-enabled state machine becomes messy
3. generating code from spec is harder
   • handling state has to be interspersed with other functionality (cf. aspect)

Good news

1. we can extract state machines from code!
   to find flaws in program logic, but not malicious backdoors
2. bugs in state machine can cause security problems, but no weird machines?
NO MORE PROSE SPECIFICATIONS OF PROTOCOL STATE MACHINES