A Model for Provably Secure Software Design

Alexander van den Berghe$^1$, Koen Yskout$^1$
Riccardo Scandariato$^2$, Wouter Joosen$^1$

$^1$imec-DistriNet, KU Leuven, Belgium
$^2$Software Engineering Division, Chalmers and Göteborg University, Sweden

FormaliSE 2017
27 May 2017
Setting the scene
Possible DFD for a banking system

- **Browser**
  - username + password; session id + transaction

- **Load balancer**
  - session id
  - username + password

- **Login**
  - session id
  - username
  - hashed password
  - session id + username

- **Transaction processor**
  - session id
  - transaction
  - username; nack

- **Customer store**
  - username

- **Session store**
  - hashed password
  - session id + username

- **Transaction store**
  - transaction
Possible DFD for a banking system

User flow:
1. User enters username and password into the browser.
2. Session ID is generated and sent to the load balancer.
3. The load balancer sends the session ID to the transaction processor.
4. The transaction processor processes the transaction and returns a response.
5. The session ID and transaction are stored in the session store.
6. The customer store stores the username and hashed password.

Data flows:
- Username and password from the browser to the load balancer.
- Username and session ID from the load balancer to the transaction processor.
- Session ID and transaction from the transaction processor to the session store.
- Username and hashed password from the customer store to the load balancer.
Possible DFD for a banking system

Browser

Load balancer

Login

Transaction processor

Customer store

Session store

Transaction store

username+password; session id+transaction

session id

username+
password

session id

hashed password

session id+username

username; nack

transaction

session id

transaction

session id+transaction

username
Our vision to improve on this
Our end goal

Security design and security properties

Properties satisfied or counterexamples
Model on familiar abstraction level

Security design and security properties

Properties satisfied or counterexamples

Modeller
Reuse well-known security solutions

Security design and security properties

Properties satisfied or counterexamples

Modeller

Catalogue
Automate property verification

Security design and security properties

Properties satisfied or counterexamples

Modeller  Catalogue  Verifier
All of this based on a formal foundation

Security design and security properties

Properties satisfied or counterexamples

Modeller  Catalogue  Verifier

Formal metamodel
A precise model for security design
Bird’s eye view of our model

Data operated on by processes that can be connected to each other to form networks

Formalised using the Coq Proof Assistant
Recall the banking system DFD

Behaviour:

- **Login**: Verifies whether some provided identity and credential match with a looked-up version.

- **Hasher**: Calculates a hash value of its input data.

- **Store**: Stores data as key-value pair.

Collections to construct complex data structures:

- cryptographic key
- identity
- credential
- session identifier
- signature

Transformed data:

- hashed
- encrypted

Abstract non-security data type:

- plain

Each encapsulating well-defined, possibly non-deterministic behaviour by a state machine; and sets of input and output queues.

**Authenticator**

**Behaviour**: Verifies whether some provided identity and credential match with a looked-up version.

**Hasher**

**Behaviour**: Calculates a hash value of its input data.

**Store**

- { (id 4, hashed (cred 5)) }

Customer store

Session store

Transaction store
Let us focus on the login process

Behaviour to model:
  Compare hash value of received password to the one stored
The data types in our model

Security specific data types
- cryptographic key, identity, credential, session identifier, signature

Transformed data
- hashed, encrypted

Abstract non-security data type
- plain

Collections to construct complex data structures
Pre-defined, off the shelf processes as building blocks

Each encapsulating well-defined, possibly non-deterministic behaviour by a state machine; and sets of input and output queues
Introducing the Authenticator process

**Behaviour:**

Verifies whether some provided identity and credential match with a looked-up version

**Diagram:**

```
  cred 5 --> Authenticator
    id 4
     |
     v
  hashed (cred 5) --> Customer store
    ACK or NACK
    id 4
```
Explicitly calculating the hash value

**Behaviour:**
Calculates a hash value of its input data
Replace the Customer Store by our Store process

**Behaviour:**
Stores data as key-value pair

```
Browser
Load
Balancer
Transaction
Processor
Session
Store
```

```
username+password;
session id+transaction
```

```
Login
Customer
store
```

```
username+
hashed password
```

**Behaviour to model:**
Compare hash value of received password to the one stored

```
id 4 +
cred 5
```

**Security specific data types**
- Cryptographic key
- Identity
- Credential
- Session identifier
- Signature

**Transformed data**
- Hashed
- Encrypted

**Abstract non-security data type**
- Plain

**Collections**
To construct complex data structures

```
(id 4, hashed (cred 5))
```

**Authenticator**
Behaviour:
Verifies whether some provided identity and credential match with a looked-up version

```
hashed (cred 5)
```

**Hasher**
Behaviour:
Calculates a hash value of its input data

```
(id 4, hashed (cred 5))
```

**Store**
```
{(id 4, hashed (cred 5))}
```
Security design as a “network” of processes

**Network** $\triangleq$ a set of processes connected by channels

Transition relation between 2 networks:
1. local state transition for each process; and
2. propagate (some) process outputs along connected channel

$\Rightarrow$ Can construct an infinite sequence of successive networks
Apply to the whole banking DFD
The banking system using our model
Username/password authentication with sessions

[Diagram of authentication process with various nodes and arrows indicating flow]

Key:
- Copier
- Discarder
- Joiner
- Composer
- Splitter
- Decomposer
- R Store
- D Store with Read, Write and Delete queues
- W Store with collection data flow
- X, Y
- All other processes

[Detailed explanation of the authentication process with nodes and arrows]
Simplified HTTPS

Browser
- [username, password]
- [session id, transaction]

User
- {username, password, transaction}
- session id

Decrypter
- {symk *}

Attacker

Load balancer
- [username, password]
- [session id, transaction]

User
- {username, password, transaction}
- session id

Encrypter
- {symk *}

Decrypter
- {symk *}

Source

Enforcer

Authenticator

Business logic

Transaction processor

Customer store

Key
- Copier
- Discarder
- Joiner
- Composer
- Splitter
- Decomposer

R Store
D Store with Read, Write and Delete queues
W

Transaction store

Collection data flow

[transaction id, transaction]

[username, transaction]

[session id]
Incorporated attacker model

[Diagram of network and data flow with labeled components and arrows indicating flow paths.]
Reasoning about security
Proving data origin authentication for transactions

Formalised using Linear-time Temporal Logic (LTL)

□(in_input tx bl ⇒ (♦ in_output tx user))

Hypothesis
transaction \(tx\) arrived as input for the business logic

Goal
transaction \(tx\) must have been sent by user earlier

Intuition of proof
start at business logic and “step backwards” process by process
Some resulting assumptions that became explicit while proving

Attacker cannot guess user’s password (i.e. brute force)
   Reasonable if good password policy is enforced.

Attacker cannot decrypt data without correct key
   Reasonable if strong encryption is used.

⇒ Should be verified against whole design (incl. documentation)
Conclusion
Recall our vision

- Security design and security properties
- Properties satisfied or counterexamples

- Modeller
- Catalogue
- Verifier

Formal metamodel

Currently extending model with process composition

Performed a user study with ±100 master students to assess understandability of model (elements)

Sneak peek: students indicated they found model (elements) easy to understand
Current state of affairs

Security design as network, security properties and manually written proof

Formal metamodel
Initial steps towards catalogue

Currently extending model with process composition

Catalogue

Formal metamodel
Assessing the model as foundation

Performed a user study with ±100 master students to assess understandability of model (elements)

Sneak peek: students indicated they found model (elements) easy to understand
Further down the road

- Modeller
- Catalogue
- Verifier

Formal metamodel

Currently extending model with process composition

Performed a user study with $\pm 100$ master students to assess understandability of model (elements)

Sneak peek: students indicated they found model (elements) easy to understand
A Model for Provably Secure Software Design

Alexander van den Berghé¹, Koen Yskout¹
Riccardo Scandariato², Wouter Joosen¹
¹imec-DistriNet, KU Leuven, Belgium
²Software Engineering Division, Chalmers and Göteborg University, Sweden

FormaliSE 2017
27 May 2017
Code samples
Inductive Data : Type :=
| plain: nat → nat → Data
| key: CryptoKey → Data
| id: Identity → Data
| cred: Credential → Data
| sid: SessionId → Data
| sig: Data → CryptoKey → Data
| enc: Data → CryptoKey → Data
| hashed: Data → Data
| collection: nat → list Data → Data
Inductive HState :=
  | h_idle: HState
  | h_hashing: Data → HState.

Record State := mk_hstate {
  hstate: HState; iostate: IOState }.

Inductive HTrans : State → State → Prop :=
  | h_read: ∀ (d : Data) (io io' : IOState),
    (Some d, io') = read_input io IN_DATA →
    HTrans (mk_hstate h_idle io)
    (mk_hstate (h_hashing d) io')
  | h_write: ∀ (d : Data) (io io' : IOState),
    io' = write_output io OUT_DATA (hashed d) →
    HTrans (mk_hstate (h_hashing d) io)
    (mk_hstate h_idle io').
Record Channel_End := mk_end {
    processID: ProcessID;
    queue_name: QueueName
}.

Record Channel := mk_chan {
    source: Channel_End;
    target: Channel_End
}.

Record Network := nw {
    processes: list Process;
    channels: list Channel
}.
Network transition relation

\textbf{Inductive} \texttt{N\_step} : Network \rightarrow Network \rightarrow Prop :=

\begin{align*}
| & \text{n\_step: } \forall \text{ ps ps' cs cs\_prop}, \\
& \quad \text{step\_all ps ps' } \rightarrow \\
& \quad \text{incl cs\_prop cs } \rightarrow \\
& \quad \text{N\_step (nw ps cs)} \\
& \quad \quad \text{(nw (propagate\_all cs\_prop ps') cs}).
\end{align*}
Definition confidential (d : Data) (n : NetworkWF) :=
\[ \forall s, \text{path } s \ n \rightarrow s@0 \models [] \text{ (no_attacker_knows } d). \]
Definition data_origin_authentication (f : Data → Prop)
  (rcv snd : ProcessID) (qr qs : QueueName) (n : NetworkWF) :=
  ∀ s d, path s n → f d →
  s@0 ⊦ [] (implies (contained_in_input d qr rcv)
  (◆(contained_in_output d qs snd))).
Some initial data from user study

The semantics of the model kind elements (processes, channels, networks) are straightforward to understand.
Overview of available processes
## Security processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hasher</td>
<td>Calculates a hash value of its input data.</td>
</tr>
<tr>
<td>Encrypter</td>
<td>Encrypts input data with a provided cryptographic key.</td>
</tr>
<tr>
<td>Decrypter</td>
<td>Decrypts input data with a provided cryptographic key.</td>
</tr>
<tr>
<td>Authenticator</td>
<td>Verifies whether an identity and credential match with a looked-up version.</td>
</tr>
<tr>
<td>Enforcer</td>
<td>Enforces input data to be cleared before passing on.</td>
</tr>
<tr>
<td>Authoriser</td>
<td>Encapsulates an authorisation policy by non-deterministically allowing or denying requests</td>
</tr>
<tr>
<td>Generator</td>
<td>Generates a digital signature given a data element and a cryptographic key.</td>
</tr>
<tr>
<td>Verifier</td>
<td>Verifies whether a data element and signature match.</td>
</tr>
</tbody>
</table>
# External processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Non-malicious user interacting with the system.</td>
</tr>
<tr>
<td>Attacker</td>
<td>Malicious user interacting with the system.</td>
</tr>
<tr>
<td>Source</td>
<td>Produces data satisfying a pre-defined property.</td>
</tr>
<tr>
<td>Sink</td>
<td>Consumes its input.</td>
</tr>
</tbody>
</table>
## Auxiliary processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business</td>
<td>Encapsulates the non-security related functionality of the system under design.</td>
</tr>
<tr>
<td>Store</td>
<td>Stores data as key-value pairs.</td>
</tr>
<tr>
<td>Comparator</td>
<td>Compares two data elements using a decidable function.</td>
</tr>
<tr>
<td>Collector</td>
<td>Collects the first data element of its $n$ first input queues into a collection.</td>
</tr>
<tr>
<td>Disperser</td>
<td>Disperses a collection into its contained elements.</td>
</tr>
<tr>
<td>Dropper</td>
<td>Non-deterministically chooses to forward or discard its input data.</td>
</tr>
<tr>
<td>Discarder</td>
<td>Discards its input data if directed to by another process.</td>
</tr>
<tr>
<td>Joiner</td>
<td>Outputs data from a non-deterministically selected input queue.</td>
</tr>
<tr>
<td>Copier</td>
<td>Copies its input data to each of its output queues.</td>
</tr>
<tr>
<td>Fork</td>
<td>Outputs input data to a non-deterministically selected output queue.</td>
</tr>
<tr>
<td>Latch</td>
<td>Remembers its last received input data and continues to output it.</td>
</tr>
</tbody>
</table>