biometryd Documentation

Release 0.0.1

Thomas Voß

July 11, 2016

Contents

CHAPTER 1

Introduction

Biometryd multiplexes and mediates access to devices for biometric identification and verification. A fingerprint reader is an example of such a device, but the overall system is designed with arbitrary devices and mechanisms in mind.

Security and privacy is one the most important design goal. For that, design, API and implementation do make sure that actual template boundary is never exposed to client applications. More to this, Biometryd and its API are designed such that actual template data is not needed for operation (unless really needed by a device). Instead, the API focuses on controlling and monitoring devices and operations instead of dealing with handling actual template data.

1.1 Coordinates

- Project: https://launchpad.net/biometryd
  - Code: https://launchpad.net/biometryd
  - Docs: http://biometryd.rtfd.io/
  - Bugs: https://bugs.launchpad.net/biometryd

CHAPTER 2

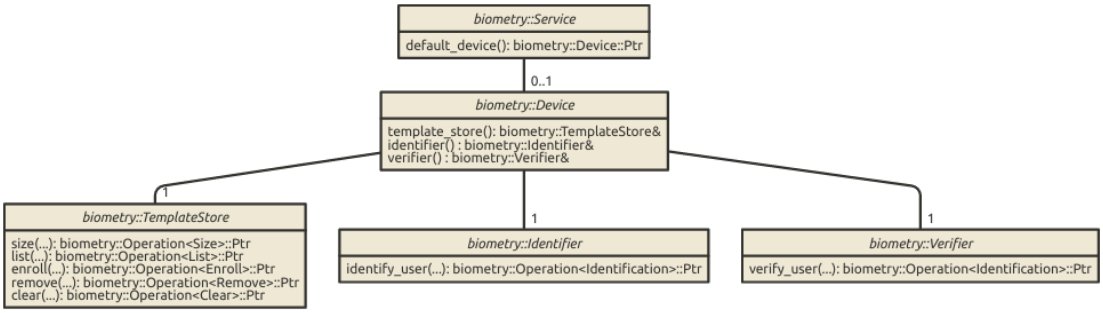
Architecture & Technology

This section presents a high-level overview of the system design. The system is devided into a set of core components and concepts that are exposed via DBus to client applications.

Please note that we designed the core components such that other types of (remote) interfaces are possible, e.g., a REST API. At the time of this writing, DBus is the primary interface though.

The primary implementation language is C++11, and we offer both a C++11 client library as well as QML bindings. Other languages, runtimes and toolkits can easily consume Biomtryd by either leveraging theq aforementioned client bindings or by directly consuming the DBus API.

The following diagram gives an overview of the main interfaces as further described in this secion:



2.1 Device

A Device abstracts an arbitrary biometric device. It bundles together access to a set of interfaces that enable client applications to:

- enroll and query information about known templates
- identify a user from a set of candidate users
- verify that a given user is actually interacting with a device

2.1.1 Template Store

A template store enables applications to manage and query information about enrolled templates. A template is device-specific and its actual data is *not* available to applications. Instead, it is referred to and uniquely identified by a numeric id in the context of one specific device implementation. Applications can:

- add (enroll) a template to the template store
- remove an individual template from the template store
- clear out all templates
- list all enrolled templates

2.1.2 Identifier

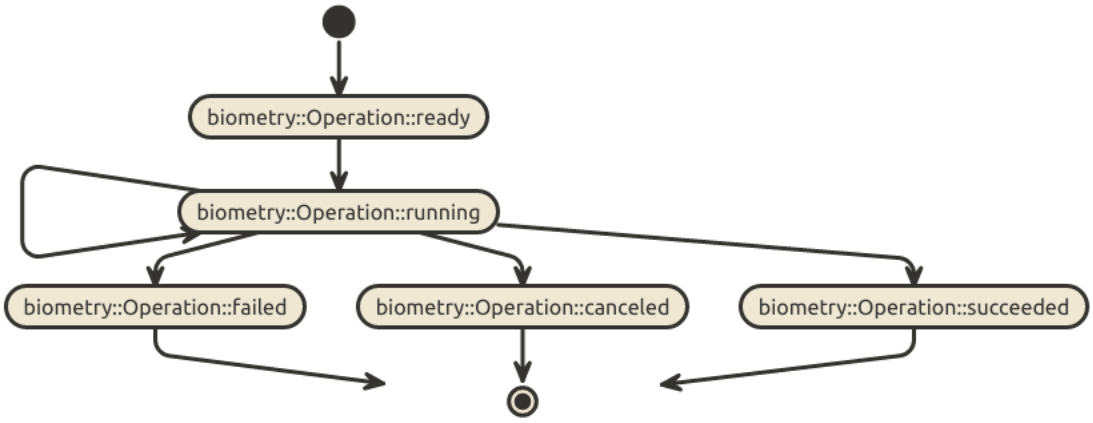
An identifier enables applications to identify one user from a given set of candidate users.

2.1.3 Verifier

A verifier enables applications to verify that a specific user is interacting with a device.

2.2 Operation

The overall system and access to its functionality is structured around the notion of an asynchronous operation. An operation is a state machine as shown in:



Client applications can start and cancel an operation, all other state transitions are triggered by the device implementation executing an operation. An operation and its state transitions can be observed by client applications and certain type of devices hand out detailed information about the ongoing operation.

Please note that both the service and device implementations might cancel an ongoing operation, too.

CHAPTER 3

Interfaces

class `biometry::Device`

Device models a biometric device.

Inherits from DoNotCopyOrMove

Subclassed by biometry::devices::FingerprintReader

Public Types

typedef std::string **`Id`**

Id is the unique name of a device.

Public Functions

virtual *TemplateStore* &**`template_store`**`()` = 0

enroller returns a device-specific template_store implementation.

virtual *Identifier* &**`identifier`**`()` = 0

identifier returns a device-specific *Identifier* implementation.

virtual *Verifier* &**`verifier`**`()` = 0

verifier returns a device-specific *Verifier* implementation.

class `Descriptor`

Descriptor bundles details about a device.

Inherits from DoNotCopyOrMove

Public Functions

virtual std::shared_ptr<*Device*> **`create`**`(`**const** util::Configuration&`)` = 0

create returns an instance of the device.

virtual std::string **`name`**`()` const = 0

name returns the human-readable name of the device.

virtual std::string **`author`**`()` const = 0

author returns the name of the author of the device implementation.

virtual std::string **`description`**`()` const = 0

description returns a one-line summary of the device implementation.

class `biometry::TemplateStore`

TemplateStore models maintenance of a device-specific template store in way that ensures that no template data ever crosses over the wire (or would need to be extracted from a TEE).

Inherits from DoNotCopyOrMove

Subclassed by biometry::devices::FingerprintReader::TemplateStore

Public Types

typedef std::uint64_t **`TemplateId`**

TemplateId is a numeric uniquely identifying a biometric template.

Public Functions

virtual *Operation*<*SizeQuery*>::Ptr **`size`**(**const** Application &*app*, **const** User &*user*) = 0

size() returns the number of templates known for user.

Parameters

- `app` - The application requesting the information.
- `user` - The user for which we want to query the number of known templates.

virtual *Operation*<*List*>::Ptr **`list`**(**const** Application &*app*, **const** User &*user*) = 0

list returns an operation that yields the list of all templates enrolled for app and user.

Parameters

- `app` - The application requesting the information.
- `user` - The user for which we want to query all enrolled templates.

virtual *Operation*<*Enrollment*>::Ptr **`enroll`**(**const** Application &*app*, **const** User &*user*) = 0

enroll returns an operation that represents the enrollment of a new template for a user.

Parameters

- `app` - The application requesting the enrollment operation.
- `user` - The user for which we want to enroll the new template.

virtual *Operation*<*Removal*>::Ptr **`remove`**(**const** Application &*app*, **const** User &*user*, *TemplateId* *id*) = 0

remove returns an operation that represents the removal of an individual template.

Parameters

- `app` - The application requesting the removal operation.
- `user` - The user for which we want to remove a specific template.
- `id` - The id of the template that should be removed.

virtual *Operation*<*Clearance*>::Ptr **`clear`**(**const** Application &*app*, **const** User &*user*) = 0

clear returns an operation that represents removal of all templates associated to user.

Parameters

- `app` - The application requesting the clear operation.
- `user` - The user for which we want to clear templates for.

struct **`Clearance`**

Clearance bundles the types passed to an observer of clearance operations.

Public Types

typedef biometry::Progress **Progress**
Progress information about the completion status of an operation.

typedef std::string **Reason**
Details about cancelation of an operation.

typedef std::string **Error**
Describes error conditions.

typedef Void **Result**
Describes the result of a *Clearance* operation.

struct Enrollment
Enrollment bundles the types passed to an observer of enrollment operations.

Public Types

typedef biometry::Progress **Progress**
Progress information about the completion status of an operation.

typedef std::string **Reason**
Details about cancelation of an operation.

typedef std::string **Error**
Describes error conditions.

typedef *TemplateId* **Result**
Describes the result of an *Enrollment* operation.

struct List
List bundles the types passed to an observer of a size operation.

Public Types

typedef biometry::Progress **Progress**
Progress information about the completion status of an operation.

typedef std::string **Reason**
Details about cancelation of an operation.

typedef std::string **Error**
Describes error conditions.

typedef std::vector<*TemplateId*> **Result**
Describes the result of a *List* operation.

struct Removal
Remove bundles the types passed to an observer of a removal operation.

Public Types

typedef biometry::Progress **Progress**
Progress information about the completion status of an operation.

typedef std::string **Reason**
Details about cancelation of an operation.

typedef std::string **Error**
Describes error conditions.

typedef *TemplateId* **Result**
Describes the result of an *Enrollment* operation.

struct **SizeQuery**
SizeQuery bundles the types passed to an observer of a size operation.

Public Types

typedef biometry::Progress **Progress**
Progress information about the completion status of an operation.

typedef std::string **Reason**
Details about cancelation of an operation.

typedef std::string **Error**
Describes error conditions.

typedef std::uint32_t **Result**
Describes the result of a *SizeQuery* operation.

class biometry::**Identifier**
Verifier abstracts verification of a user.

Inherits from DoNotCopyOrMove

Public Functions

virtual *Operation*<Identification>::Ptr **identify_user**(**const** Application &*app*, **const** Reason &*reason*) = 0
identify_user returns an operation that represents the identification of a user given a set of candidates with the given reason.

class biometry::**Verifier**
Verifier abstracts verification of a user.

Inherits from DoNotCopyOrMove

Public Functions

virtual *Operation*<Verification>::Ptr **verify_user**(**const** Application &*app*, **const** User &*user*, **const** Reason &*reason*) = 0
verify_user returns an operation that represents the verification of 'user' for 'reason'.

template <typename *T*>
class biometry::**Operation**
An *Operation* models an asynchronous operation that can be started and cancelled, as well as observed.

Inherits from DoNotCopyOrMove

Public Functions

virtual void **start_with_observer**(**const typename** *Observer*::Ptr &*observer*) = 0
start_with_observer starts the operation, handing updates to 'observer'.

virtual void **cancel**() = 0
cancel stops the operation, confirming cancellation to the installed observer.

class Observer
An *Observer* enables client code to monitor an ongoing operation.

Inherits from DoNotCopyOrMove

Subclassed by biometry::qml::TypedOperation< T >::Observer

Public Functions

virtual void **on_started**() = 0
on_state_changed is called whenever the state of an operation changed, handing the current (new) and previous state to the observer.

virtual void **on_progress**(**const** Progress &*progress*) = 0
on_progress is called whenever an operation advances.

Parameters
- `progress` - contains details describing the progress.

virtual void **on_canceled**(**const** Reason &*reason*) = 0
on_canceled is called when an operation is cancelled.

Parameters
- `reason` - contains details explaing the reson for cancelling.

virtual void **on_failed**(**const** Error &*error*) = 0
on_failed is called when an operation fails.

Parameters
- `error` - provides details describing the error condition.

virtual void **on_succeeded**(**const** Result &*result*) = 0
on_succeeded is called when an operation succeeds.

Parameters
- `result` - provides details handing the result of the operation to observers.

CHAPTER 4

Extending Biometryd

Biometryd can be extended by implementing the interface `biometry::Device`. We support both in-tree and out-of-tree plugins. In-tree plugin authors should add their device implementation in the folder `${BIOMETRYD_ROOT}/src/biometry/devices` and submit their code contribution as a merge proposal to `https://launchpad.net/biometryd`.

Out-of-tree plugin authors should rely on

- `BIOMETRYD_DEVICES_PLUGIN_DESCRIBE(name, author, desc, major, minor, patch)`
  - `name` The name of the plugin
  - `author` The author of the plugin
  - `desc` Human-readable description of the plugin
  - `major` Major revision of the plugin
  - `minor` Minor revision of the plugin
  - `patch` Patch level of the plugin
- `BIOMETRYD_DEVICES_PLUGIN_CREATE`
- `BIOMETRYD_DEVICES_PLUGIN_DESTROY`

to describe, instantiate and destroy their plugin, respectively. The following snippet demonstrates a complete plugin definition. The resulting shared object file should be installed to `biometryd config --flag default_plugin_directory`. Once the plugin is installed, it can be referenced by its name as passed to `BIOMETRYD_DEVICES_PLUGIN_DESCRIBE`.

```
#include <biometry/devices/plugin/interface.h>

#include "mock_device.h"

/// [Defining the create function]
BIOMETRYD_DEVICES_PLUGIN_CREATE
{
    return new testing::MockDevice();
}
/// [Defining the create function]

/// [Defining the destroy function]
BIOMETRYD_DEVICES_PLUGIN_DESTROY
{
    delete d;
```

```
    }
    /// [Defining the destroy function]

    /// [Describing the plugin]
    BIOMETRYD_DEVICES_PLUGIN_DESCRIBE(
        "TestPlugin",
        "Thomas Voß <thomas.voss@canonical.com>",
        "Just a plugin for testing purposes",
        0,
        0,
        0)
    /// [Describing the plugin]
```

CHAPTER 5

Manual Test Plan

This section lists manual test cases that should be executed prior to landing. The test cases exercise the main functionality and aim to guarantee a baseline level of functionality that should not regress across releases.

Please note that individual landings might require specific testing steps in addition to the ones listed here.

We assume that testers use a freshly bootstrapped device.

5.1 Turbo

5.1.1 Enrolling a New Template

- Boot the phone
- Unlock the greeter/complete the wizard
- Start “System Settings”
- Switch to the “Security & Privacy” page
- Select “Fingerprint ID”
- Select “Add Fingerprint”
  - Enroll a new template according to the onscreen instructions.
  - Make sure that feedback given during enrollment is meaningful and reasonable.
  - After completion, check if the list of enrolled fingerprints has grown by 1.
- Select the recently enrolled fingerprint and rename it:
  - Ensure that the name of the fingerprint is persistent across restarts of “System Settings”

5.1.2 Identifying With A Fingerprint

- In “System Settings”, choose Fingerprint ID as lock security.
- Lock the screen.
- Wake up the phone by pressing the power button.
- Try to identify with your previously enrolled fingerprint.

- Lock the screen again.
- Wake up with the home button.
- Try to identify with your previously enrolled fingerprint.
- Lock the screen again.
- Wake up the screen and try to identify with a finger that hasn't been enrolled previously. The attempts should fail and the device should fall back to your passcode.

5.1.3 Removing a Previously Enrolled Template

- Start "System Settings"
- Switch to "Security & Privacy" page
- Remove at least one enrolled fingerprint
  - Make sure that the fingerprint is removed from the list
- Lock the screen and try to identify with the fingerprint. The attemtps should fail.

B