Fachhochschule Hannover
Faculty IV - Department of Computer Science
Trust@FHH Research Group
http://trust.inform.fh-hannover.de
trust@f4-i.fh-hannover.de

### tnc@fhh developer documentation

# imunit

**A framework for the development of IMC/IMV components according to the TNC architecture. For imunit version 0.6.0.**

Ingo Bente

September 15, 2009

## 1 Introduction

The imunit package is part of the tnc@fhh software. tnc@fhh[1] is an open source implementation of the TNC architecture specified by the Trusted Computing Group. The imunit package provides an easy to use framework for the development of new IMC/IMV pairs. imunit compiles and runs on many Unix-like systems. Windows is currently not supported. However, we assume that Windows-support would need only minor modifications of the build process.[2] This document describes the imunit package from a developer's perspective. Classes and interfaces that are necessary for the development of new IMC/V pairs will be discussed in detail.

## 2 Coverage

The following TNC interfaces are supported by imunit

- IF-IMC 1.2 (`http://www.trustedcomputinggroup.org/resources/tnc_ifimc_specification`)
- IF-IMV 1.2 (`http://www.trustedcomputinggroup.org/resources/tnc_ifimV_specification`)

[1] Old wording is TNC@FHH (uppercase)
[2] Appropriate patches are very welcome ...

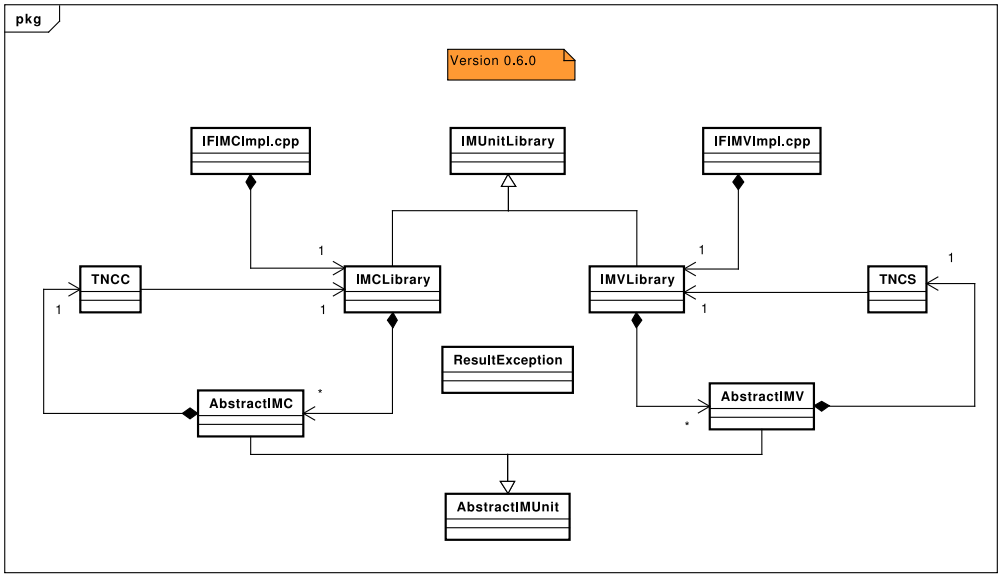


Figure 1: Architecture overview for imunit

To demonstrate the use of imunit for the development of new IMC/V pairs, this imunit-dev package contains a very basic ExampleIMC/V pair. This IMC/V pair implements very few functions and should serve as starting point for your own IMC/V pairs.

# 3 Architecture of imunit

Figure 1 gives an overview of the imunit package. The components and their functions will be described in the following.

## 3.1 Structure of Directories

The imunit package contains the following directories:

- `./build` empty directory for build artifacts
- `./cmake_modules` cmake find and install scripts
- `./include` TCG header files
- `./src/imunit` imunit source code
- `./src/imunit/exception` source code for imunit-specific exceptions
- `./src/imunit/imc` imc-specific source code (IF-IMC)

- `./src/imunit/imv` imv-specific source code (IF-IMV)

## 3.2 Overall Design, Classes, Functions

On Unix-like Systems, IMCs and IMVs are provided as shared objects (`*.so`). The IMC/V modules can be used via the IF-IMC/V interface. These interfaces are specified as C-functions. However, the imunit package is mostly written in C++, allowing us to use an object-oriented design internally.

Besides the C/C++ mapping, there is another interesting issue regarding the instantiation of objects: in imunit, there is one class that represents the shared library itself (named IMCLibrary/IMVLibrary). Normally, for a specific IMC or IMV shared library, there will be one object of this class for each TNCC/TNCS running on a platform. To address the issue that a TNCC/TNCS can handle multiple connections in parallel, there needs to be a connection-based representation of an IMC/IMV. This representation is provided by the AbstractIMC/AbstractIMV class. Normally, there will be one object of this class for each connection that is handled by a TNCC/TNCS on a platform. As a developer of an IMC/V pair, you have to provide your own implementation of IMCLibrary and AbstractIMC or IMVLibrary and AbstractIMV by inhereting from the classes of the imunit package. The good thing is that you do not have to deal with plain C this way.

In the following, we will give a short description of each class available in the imunit package. How they can be used to develop a simple ExampleIMC/V will be discussed in section 4.

### 3.2.1 General Classes

**IMUnitLibrary** This class encapsulates the similarities of an IMC and an IMV library. There is exactly one instance of this class for each TNCC or TNCS that uses the corresponding IMC or IMV library. The main purpose of this class is to provide general information about the library (name, message types used) and to handle the (de)initialization process. Direct known subclasses are IMCLibrary and IMVLibrary.

**AbstractIMUnit** This class encapsulates the similarities of an IMC and an IMV instance that is bound to a specific connection. The connection is handled via the TNCC or the TNCS. There is normally one instance of this class for each ongoing connection. The class implements methods that are available for IMCs and for IMVs (notifyConnectionChange(), batchEnding() and receiveMessage()). Direct known subclasses are AbstractIMC and AbstractIMV.

**ResultException** A simple exception class. Extends `std::exception`. This class can carry a `TNC_Result` return value. Exceptions of this class are used internally in the imunit package to handle errors in a more convenient way than it is possible with simple return values. If a TNCC/TNCS must be notified about an error, the ResultException can be easily mapped to a simple `TNC_Result` value that is return by an interface C-function. This class has no known subclasses.

### 3.2.2 IMC-specific Classes

**IMCLibrary** This class inherits from IMUnitLibrary and encapsulates all IMC specific functionalities of an IMC library. This class multiplexes incoming calls from an TNCC to a conrete instance of AbstractIMC. Furthermore, it holds all pointers to the TNCC functions as specified by IF-IMC. IMC developers must extend this class to implement their own IMC library. IMCLibrary defines a pure virtual factory method (createNewImcInstance()) that must be implemented by the IMC developer.

**AbstractIMC** This class inherits from AbstractIMUnit. It represents instances of an IMC that are bound to a certain connection. It manages the state of a concrete IMC related to a given connection ID. IMC developers must extend this class to implement their own IMC. This class defines a pure virtual method (beginHandshake()) that must be implemented by the IMC developer.

**TNCC** This is an interface class that encapsulates all TNCC functions of the IF-IMC interface. It allows AbstractIMC (and the sub-classes implemented by an IMC developer) to call the TNCC via an instance of this class (instead of directly using C-function pointers). TNCC has no known sub-classes.

**IFIMCImpl.cpp (deprecated)** This "'class"' is actually no class. It contains the mapping from C to C++ for the IMC functions of IF-IMC. *Note:* This file is empty since version 0.6.0. Its content has been moved to the `TNCFHH_IMCLIBRARY_INITIALIZE` macro in IMCLibrary.h. The file will be removed from imunit in the next release.

### 3.2.3 IMV-specific Classes

**IMVLibrary** This class inherits from IMUnitLibrary and encapsulates all IMV specific functionalities of an IMV library. This class multiplexes incoming calls from an TNCS to a conrete instance of AbstractIMV. Furthermore, it holds all pointers to the TNCS functions as specified by IF-IMV. IMV developers must extend this class to implement their own IMV library. IMVLibrary defines a pure virtual factory method (createNewImvInstance()) that must be implemented by the IMV developer.

**AbstractIMV** This class inherits from AbstractIMUnit. It represents instances of an IMV that are bound to a certain connection. It manages the state of a concrete IMV related to a given connection ID. IMV developers must extend this class to implement their own IMV.

**TNCS** This is an interface class that encapsulates all TNCS functions of the IF-IMV interface. It allows AbstractIMV (and the sub-classes implemented by an IMV developer) to call the TNCS via an instance of this class (instead of directly using C-function pointers). TNCS has no known sub-classes.

**IFIMVImpl.cpp (deprecated)** This "'class"' is actually no class. It contains the mapping from C to C++ for the IMV functions of IF-IMV. *Note:* This file is empty since version 0.6.0. Its content has been moved to the `TNCFHH_IMVLIBRARY_INITIALIZE` macro in IMVLibrary.h. The file will be removed from imunit in the next release.

# 4 An ExampleIMC/V pair

This section is a step-by-step guide that explains how to code your own IMC/V based upon imunit.

## 4.1 Coding the ExampleIMC

1. Create a class ExampleIMCLibrary that extends IMCLibrary.
   a) Define the message types for your IMC. Normally, each IMC has its own message type[3] (done in ExampleIMCLibrary.h). The message type is used for two purposes: 1) it is used to indicate the type of messages send to the TNCC and 2) it is used to tell the TNCC which message types the IMC is interested in receiving.

```
/* define Vendor ID (see IANA PEN). */
#define VENDOR_ID 0x0080ab
/* define Messagesubtype */
#define MESSAGE_SUBTYPE 0xfe
```

   b) Implement a ctor (and dtor if necessary). Add your message type defined above to the list of message types the IMC wants to receive.

```
ExampleIMCLibrary::ExampleIMCLibrary()
{
  LOG4CXX_INFO(logger, "Load␣ExampleIMC␣library␣");
  /* set all attributes inherited from tncfhh::iml::IMCLibrary */
  // the library name for logging
  this->imUnitLibraryName = "ExampleIMC";
  // add an messageType composed of Vendor ID (IANA PEN) and
       MessageSubtype
  this->addMessageType(VENDOR_ID, MESSAGE_SUBTYPE);
}
```

   c) Initialize the imunit framework (done in ExampleLibrary.cpp). This defines the C-functions interface according to IF-IMC and maps those functions to C++ methods of imunit. You must provide the class name of your ExampleIMCLibrary implementation as argument. This causes the framework to create an instance of ExampleIMCLibrary within the initialization macro.

```
// TNC@FHH IMCLibrary Initialization +
// implement IF-IMC c-functions
TNCFHH_IMCLIBRARY_INITIALIZE(ExampleIMCLibrary) ;
```

   d) Implement the pure virtual factory method. This method creates a new instance of the ExampleIMC class (described in step 2). The method is called when a new connection is created. The memory is freed when the same connection is deleted.

[3] This will likely change when IF-M is released.

```
tncfhh::iml::AbstractIMC *ExampleIMCLibrary::createNewImcInstance(
    TNC_ConnectionID conID)
{
  LOG4CXX_TRACE(logger, "createNewImcInstance(␣" << conID << ")");
  // just return a new instance of ExampleIMC
  return new ExampleIMC(conID, this);
}
```

2. Create a class ExampleIMC that extends AbstractIMC.

   a) Define the ctor (and dtor if necessary). The ctor needs the connection ID and a pointer to the corresponding ExampleIMCLibrary as arguments. Internally, this causes the instantiation of a TNCC object which can forward the calls to the "real" TNCC via the pointer to the ExampleIMCLibrary (which holds the function pointers to the "real" TNCC). The benefit is: you as IMC developer can call methods of the TNCC instantiation to talk to the "real" TNCC.

   ```
   ExampleIMC::ExampleIMC(TNC_ConnectionID conID, ExampleIMCLibrary *
       pExampleIMCLibrary)
     :AbstractIMC(conID, pExampleIMCLibrary)
   {
     // initialize
   }
   ```

   b) Implement the (pure virtual) mandatory beginHandshake() method. In this case, our IMC sends a first message to its ExampleIMV (by calling sendMessage() of the TNCC).

   ```
   TNC_Result ExampleIMC::beginHandshake()
   {
     LOG4CXX_TRACE(logger, "beginHandshake()");
     // this message should be send to ExampleIMV
     std::string sendMessage("Example␣message␣from␣ExampleIMC");
     LOG4CXX_TRACE(logger, "Send␣Message:␣" << sendMessage);
     // send message
     this->tncc.sendMessage((unsigned char*)sendMessage.c_str(),
         sendMessage.size()+1/*for'\0'*/, VENDOR_ID, MESSAGE_SUBTYPE);
     // return all ok
     return TNC_RESULT_SUCCESS;
   }
   ```

   c) Implement optional methods. These are already implemented by the imunit framework. But normally, to have them behave in a reasonable (from the IMC developers point of view) manner, these should be overwritten. We will override all optional methods.

      i. Implement receiveMessage(). This is called to deliver a message from the IMV which was received by the TNCC to the IMC. Here, our IMC just sends another message.

      ```
      TNC_Result ExampleIMC::receiveMessage(TNC_BufferReference message,
          TNC_UInt32 messageLength, TNC_MessageType messageType)
      {
        LOG4CXX_DEBUG(logger, "receiveMessage␣round␣" << this->getRound
            ());
        // print received message dirty out. WARNING: don't ape this,
        // message should end with non-null! Heed: Message can be evil!
        LOG4CXX_INFO(logger, "Received␣Message:␣" << message);
      ```

```
  // this message should be send to ExampleIMV
  std::string sendMessage("Another␣example␣message␣from␣ExampleIMC
      .");
  LOG4CXX_INFO(logger, "Send␣Message:␣" << message);
  // send message
  this->tncc.sendMessage((unsigned char*)sendMessage.c_str(),
      sendMessage.size()+1/*for'\0'*/, VENDOR_ID, MESSAGE_SUBTYPE)
      ;
  // return all ok
  return TNC_RESULT_SUCCESS;
}
```

ii. Implement batchEnding(). Here, it basically does nothing.

```
TNC_Result ExampleIMC::batchEnding()
{
  LOG4CXX_TRACE(logger, "batchEnding");
  // return all ok
  return TNC_RESULT_SUCCESS;
}
```

iii. Implement notifyConnectionChange(). The new connection state can be queried via the getConnectionState() method. Here, it basically does nothing. Normally, you would change the state of your IMC according to the connection state.

```
TNC_Result ExampleIMC::notifyConnectionChange()
{
  LOG4CXX_TRACE(logger, "notifyConnectionChange");
  /* if new handshake start */
  if(this->getConnectionState() == TNC_CONNECTION_STATE_HANDSHAKE)
  /* reset IMC */;
  // return all ok
  return TNC_RESULT_SUCCESS;
}
```

3. Finished. Thats all for the IMC part.

## 4.2 Coding the ExampleIMV

Coding the ExampleIMV conceptually works the same as coding the ExampleIMC. There are only minor differences regarding which methods must be overwritten/implemented.

1. Create a class ExampleIMVLibrary that extends IMVLibrary.
   a) Define the message types for your IMV. Normally, each IMV has its own message type[4] (done in ExampleIMVLibrary.h). The message type is used for two purposes: 1) it is used to indicate the type of messages send to the TNCS and 2) it is used to tell the TNCS which message types the IMV is interested in receiving.

```
/* define Vendor ID (see IANA PEN). */
#define VENDOR_ID 0x0080ab
/* define Messagesubtype */
#define MESSAGE_SUBTYPE 0xfe
```

[4] This will likely change when IF-M is released.

b) Implement a ctor (and dtor if necessary). Add your message type defined above to the list of message types the IMV wants to receive.

```
ExampleIMVLibrary::ExampleIMVLibrary()
{
  LOG4CXX_INFO(logger, "Load␣ExampleIMV␣library␣");
  /* set all attributes inherited from tncfhh::iml::IMVLibrary */
  // the library name for logging
  this->imUnitLibraryName = "ExampleIMV";
  // add an messageType composed of Vendor ID (IANA PEN) and
      MessageSubtype
  this->addMessageType(VENDOR_ID, MESSAGE_SUBTYPE);
}
```

c) Initialize the imunit framework (done in ExampleIMVLibrary.cpp). This defines the C-functions interface according to IF-IMV and maps those functions to C++ methods of imunit. You must provide the class name of your ExampleIMVLibrary implementation as argument. This causes the framework to create an instance of ExampleIMVLibrary within the initialization macro.

```
// TNC@FHH IMVLibrary Initialization +
// implement IF-IMV c-functions
TNCFHH_IMVLIBRARY_INITIALIZE(ExampleIMVLibrary) ;
```

d) Implement the pure virtual factory method. This method creates a new instance of the ExampleIMV class (described in step 2). The method is called when a new connection is created. The memory is freed when the same connection is deleted.

```
tncfhh::iml::AbstractIMV *ExampleIMVLibrary::createNewImvInstance(
    TNC_ConnectionID conID)
{
  LOG4CXX_TRACE(logger, "createNewImvInstance(␣" << conID << ")");
  // just return a new instance of ExampleIMV
  return new ExampleIMV(conID, this);
}
```

2. Create a class ExampleIMV that extends AbstractIMV.

a) Define the ctor (and dtor if necessary). The ctor needs the connection ID and a pointer to the corresponding ExampleIMVLibrary as arguments. Internally, this causes the instantiation of a TNCS object which can forward the calls to the "real" TNCS via the pointer to the ExampleIMVLibrary (which holds the function pointers to the "real" TNCS). The benefit is: you as IMV developer can call methods of the TNCS instantiation to talk to the "real" TNCS.

```
ExampleIMV::ExampleIMV(TNC_ConnectionID conID, ExampleIMVLibrary *
    pExampleIMVLibrary)
  :AbstractIMV(conID, pExampleIMVLibrary)
{
  // initialize
}
```

b) In contrast to the IMC part, there is no mandatory (pure virtual) method that must be implemented by ExampleIMV. However, we will override several optional methods.

i. Implement receiveMessage(). This is called to deliver a message from the IMC which was received by the TNCS to the IMV. Here, our IMV sends a new message if this is the first round of the TNC handshake. Otherwise, it provides an allow recommendation. The round counter is managed by the imunit framework as follows:
   - set to 0 at the end of IMC/VLibrary::notifyConnectionChange() when called with newState == `TNC_CONNECTION_STATE_HANDSHAKE`
   - for IMC/V : increased before IMC/VLibrary::batchEnding returns
   - for the IMC: increased before IMCLibrary::beginHandshake returns

```
TNC_Result ExampleIMV::receiveMessage(TNC_BufferReference message,
    TNC_UInt32 messageLength, TNC_MessageType messageType)
{
  LOG4CXX_DEBUG(logger, "receiveMessage round " << this->getRound())
      ;
  // print received message dirty out. WARNING: don't ape this,
  // message should end with non-null! Heed: Message can be evil!
  LOG4CXX_INFO(logger, "Received Message: " << message);
  /* only send one message to ExampleIMC */
  if (this->getRound() < 1) {
    // this message should be send to ExampleIMC
    std::string sendMessage("Example message from ExampleIMV");
    LOG4CXX_INFO(logger, "Send Message: " << sendMessage);
    // send message
    this->tncs.sendMessage((unsigned char*)sendMessage.c_str(),
        sendMessage.size()+1/*for '\0'*/, VENDOR_ID, MESSAGE_SUBTYPE)
        ;
  } else {
    /* validation finish, set recommendation & co */
    validationFinished = true;
    // for access allow:
    actionRecommendation = TNC_IMV_ACTION_RECOMMENDATION_ALLOW;
    // set evaluation (see TNC_IMV_EVALUATION_RESULT_...)
    evaluationResult = TNC_IMV_EVALUATION_RESULT_DONT_KNOW;
  }
  // return all ok
  return TNC_RESULT_SUCCESS;
}
```

ii. Implement batchEnding(). Here, it basically does nothing.

```
TNC_Result ExampleIMV::batchEnding()
{
  LOG4CXX_TRACE(logger, "batchEnding");
  // return all ok
  return TNC_RESULT_SUCCESS;
}
```

iii. Implement notifyConnectionChange(). The new connection state can be queried via the getConnectionState() method. Here, it basically does nothing. Normally, you would change the state of your IMV according to the connection state.

```
TNC_Result ExampleIMV::notifyConnectionChange()
{
  LOG4CXX_TRACE(logger, "notifyConnectionChange");
  /* if new handshake start */
  if(this->getConnectionState() == TNC_CONNECTION_STATE_HANDSHAKE)
  /* reset IMV */;
  // return all ok
  return TNC_RESULT_SUCCESS;
}
```

3. Finished. Thats all for the IMV part.

You can find the ExampleIMC/V pair discussed here in the imunit-dev package.

# 5 Conclusion

By using the imunit framework of tnc@fhh, it is (from our point of view) pretty easy to implement IMC/V pairs. One main benefit of our approach is that the developer can benefit from an object oriented design and does not have to deal with plain C.

We appreciate any feedback, comments, request for features and so on. Just send an email to trust@f4-i.fh-hannover.de.

# 6 Copyright and License