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

**Documentation Version 0.7.0**

# TNC@FHH

# An Open Source TNC Implementation

Ingo Bente, Bastian Hellmann, Jan Bernhardt, Arne Welzel

May 3, 2010
Trust@FHH Research Group

## Contents

This documentation describes the TNC@FHH software version 0.7.0. It covers all subprojects of TNC@FHH and highlights the overall project structure. Furthermore, it gives examples how to use parts of the TNC@FHH framework in your own projects (like coding your own IMC/V based upon TNC@FHH). This document is (although contained in this release) still a work in progress. We try to keep it up to date, but some sections might refer to older versions of TNC@FHH[1]. Please report any issues related to it to trust@f4-i.fh-hannover.de. This document does *not* describe all configuration options and tasks in order to set up a TNC@FHH environment. For installation and configuration issues, please refer to the README file and to our wiki: `http://trust.inform.fh-hannover.de/wiki`.

# 1 Introduction

The TNC@FHH project is an open source implementation of the Trusted Network Connect (TNC) framework which is specified by the Trusted Computing Group (TCG). TNC@FHH allows you to provision access to a network based upon factors like the user credentials and the requesting endpoint's integrity state.

The following TNC components and their respective interfaces are implemented by TNC@FHH:

- IMCs (IF-IMC 1.2)
- IMVs (IF-IMV 1.2)
- TNCS (IF-TNCCS 1.1)
- NAA (IF-T EAP 1.1)

On the Policy Decision Point, TNC@FHH works as an extension to FreeRADIUS. FreeRADIUS handles the user authentication and all of the standard EAP message processing. TNC@FHH is plugged into FreeRADIUS via a new EAP module that supports TNC.

# 2 Project Structure

TNC@FHH consists of several subprojects:

**freeradius-eaptnc-patch** Patch for FreeRADIUS to add support for TNC. This is basically a new EAP module that handles the TNC traffic and forwards the TNC specific data to the naaeap module. The patch is implemented in C.

[1] E.g., some of the UML diagrams refer to version 0.6.0 since there were no substantial changes.

**freeradius-eapttls-patch** Patch for FreeRADIUS to add support for multiple EAP-methods to be tunneld within one EAP-TTLS channel. This explicitly supports to chain multiple EAP methods, i.e. EAP-MD5 (for user authentication) and then EAP-TNC (for endpoint assessment) in one EAP-TTLS tunnel. The patch is implemented in C.

**naaeap** A shared library that is used by the EAP TNC module. It parses the EAP-TNC data, handles fragmentation and forwards the parsed data (i.e. a TNCCS message) to the tncs module. Outgoing TNCCS messages are in turn properly encapsulated within EAP TNC. naaeap is implemented in C++.

**tncs** tncs is a shared library that is used by the naaeap module. It represents the TNCS component within the TNC architecture. tncs handles the communication with the IMVs that are installed on the PDP and the TNCC on the AR. tncs is implemented in C++.

**imunit** imunit is a framework for developing IMC/V pairs. All TNC@FHH IMC/V pairs are based upon imunit. imunit is implemented in C++.

**IMC/V pairs** TNC@FHH comes with a set of working IMC/V pairs:

- example: A hello world example of an imc/v pair.
- dummy: Another hello world example with a bit more functionality.
- clamav: Checks the status of the AV software clamav.
- platid: Allows to authenticate an endpoint based upon X.509 certificates (supports TPMs)
- attestation: Allows to verify the integrity of an endpoint based on TPM attestation (using AIKs).
- hostscanner: Allows to check the status of arbitrary ports on an endpoint.

**tncsim** This is a simple test program that acts both as TNCC and TNCS, but without an NAR or NAA component. It was developed to ease the testing of IMC/V pairs. It uses libtnc as client and TNC@FHH tncs as server. tncsim can load both IMCs and IMVs and starts a single TNC handshake by calling `beginHandhake()` for each IMC. tncsim is implemented in C++.

Figure 1 gives an overview of the existing subprojects and der dependencies. Detailed information about each subproject is given in the following sections.

# 3 Building and Installing TNC@FHH

Since version 0.7.0, there is only one TNC@FHH tarball available that contains all software components. This makes the build and install process a lot easier. The following prerequisites should be fulfilled in order to be able to build all TNC@FHH components:

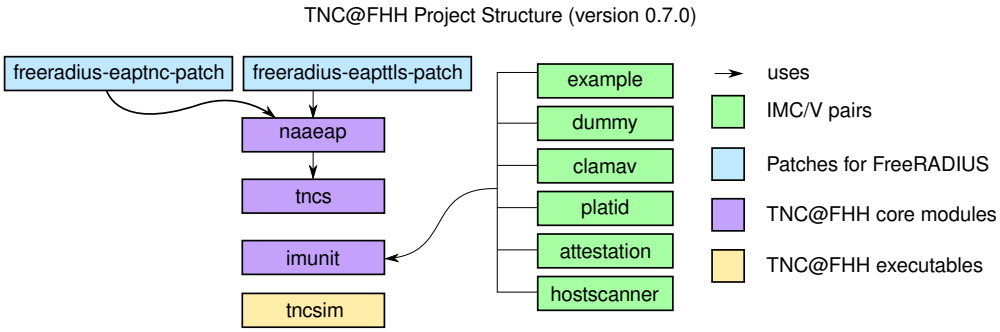


Figure 1: Project Structure of TNC@FHH

- cmake
  (use the version provided by your distribution)
- log4cxx (mandatory for all components)
  (use the version provided by your distribution)
- xerces-c (needed by tncs)
  (use the version provided by your distribution)
- trousers >= 0.3.4 (needed by attestation IMC)
  http://sourceforge.net/projects/trousers/
- libtnc >= 1.24 (needed by tncsim)
  http://sourceforge.net/projects/libtnc/

To build (and optionally install) TNC@FHH with its default configuration, just do the following:

```
// extract archive
tar -xzf tncfhh-0.7.0.tar.gz
// switch to directory
cd tncfhh-0.7.0
// create build directory
mkdir build
// switch to build directory
cd build
// build (and optionally install) TNC@FHH
cmake ../
make
make install
```

This will build (and install) all IMCs, all IMVs, imunit, naaeap, tncs and tncsim. This *will not* build or install any of the FreeRADIUS patches. The process to patch your FreeRADIUS server in order to use TNC@FHH is described in our wiki[2].

There are some cmake variables available to configure which TNC@FHH components are actually build:

[2] http://trust.inform.fh-hannover.de/wiki/index.php/Main_Page

- `TNCFHH_BUILD_IMCS` (ON/OFF)
  Controls whether IMCs are built or not. Default is ON.
- `TNCFHH_BUILD_IMVS` (ON/OFF)
  Controls whether IMVs are built or not. Default is ON.
- `TNCFHH_BUILD_SERVER` (ON/OFF)
  Controls whether naaeap and tncs are built or not. Default is ON.
- `TNCFHH_BUILD_TNCSIM` (ON/OFF)
  Controls whether tncsim is built or not. Default is ON (implies that naaeap and tncs are built as well).

If you get error messages during the build process, make sure that you fulfill the necessary prerequisites. E.g. make sure that you have libtnc if you want to build tncsim or trousers for the attestation IMC. If you want to exclude single IMCs or IMVs from the build process, just comment out the corresponding `add_subdirectory` statement in the CMakeLists.txt file (e.g. in `imcv/attestation/CMakeLists.txt:4`).

In order to actually use the TNC@FHH components, you need

- a patched FreeRADIUS server on the PDP
- a 802.1X supplicant that supports TNC (we suggest wpa_supplicant[3]). Note that you need wpa_supplicant version 0.7.1 or higher in order to use the platid and attestation IMC.
- a 802.1X compatible switch

Details on how to set up a working TNC environment are available in our wiki[4].

# 4 freeradius-eaptnc-patch

## 4.1 General Information

The patch adds a new EAP-TNC method to FreeRADIUS. An outdated version of this patch is already contained in the FreeRADIUS source tree. However, to get the latest version of TNC@FHH running, you will need to apply the patch that is contained in this TNC@FHH release. The EAP-TNC method can be used as tunneld EAP method, e.g. within EAP-TTLS.

## 4.2 Architecture

The architecture is quite simple. The EAP module just hooks into FreeRADIUS by implementing the necessary functions in `rlm_eap_tnc.c` and letting FreeRADIUS know about them:

[3] http://hostap.epitest.fi/wpa_supplicant/
[4] http://trust.inform.fh-hannover.de/wiki

```
/*
 * The module name should be the only globally exported symbol.
 * That is, everything else should be 'static'.
 */
 EAP_TYPE rlm_eap_tnc = {
  "eap_tnc",
  tnc_attach, /* attach */
  tnc_initiate, /* Start the initial request */
  NULL, /* authorization */
  tnc_authenticate, /* authentication */
  tnc_detach /* detach */
};
```

- `tnc_attach` This function is called when an instance of the EAP-TNC-module is created. This happens when FreeRADIUS parses the corresponding EAP configuration file (`eap.conf`). The function initializes the naaeap module.
- `tnc_initiate` This function is called when a new handshake with EAP-TNC is about to begin. The handshake is triggered by an incoming EAP-Response/Identity message from the client. The function checks for the presence of a secure tunnel, so that EAP-TNC is not run standalone but within a secure EAP-method. Then it calculates the connection ID for this handshake and creates the first EAP-TNC-Request message which is sent to the client.
- `tnc_authenticate` This function is called when a EAP-TNC-Response message was received. It basically forwards the EAP-TNC data to the naaeap module and composes an appropriate EAP-Response message. When the TNC handshake is finished, the result is forwarded to FreeRADIUS via the configuration-item `TNC-Status` for the corresponding connection. TNC-Status can have the values `Access` or `Isolate`. This value will than be processed by FreeRADIUS which maps it to a VLAN-assigment for the current client.
- `tnc_detach` This function handles the destruction of an instance of the EAP-TNC-module. This happens when FreeRADIUS is stopped. It deinitializes the naaeap module.

The code in `eap_tnc.c` just (de)marshalls the TNC payload in the EAP packet. The main logic described above is located in `rlm_eap_tnc.c`. For further details, please refer to the sourcecode.

# 5 freeradius-eapttls-patch

## 5.1 General Information

This section describes the concept and implementation of the EAP-TTLS-patch for FreeRADIUS.

#### Goal of the patch

The main goal of the patch is to allow multiple inner authentication methods inside an EAP-TTLS-tunnel. This could be EAP-MD5 as a user authentication method, followed by EAP-TNC as a hardware or platform authentication method.

#### Specification of the patch

The patch in the current version is implemented to do any authentication methods supported by the EAP-module of `FreeRADIUS` as the first inner method, and EAP-TNC afterwards. EAP-TNC is only started if the first method was successful, otherwise the authentication request of the supplicant will result in an Access-Reject. If the first inner method was successful, then the outgoing Access-Accept is intercepted and cached, and a new authentication with EAP-TNC is started inside the tunnel.

#### Not implemented

At the moment, there is no support for the use of non-EAP-methods as the first inner authentication method. Doing so will not properly start EAP-TNC as the second inner method.

#### Used sources

The patch was built upon `FreeRADIUS` version 2.1.7 and tested with `wpa_supplicant` version 0.6.9 on Ubuntu 9.04 and 9.10.

#### Documentation in the Trust@FHH-wiki

There are How To's for building and configuring the EAP-TTLS-patch in our wiki: http://trust.inform.fh-hannover.de/wiki/index.php/Main_Page.

## 5.2 RLM_EAP_TTLS

This section describes the implementation and the changes and additions to the original EAP-TTLS-module source code.

### 5.2.1 File `eap_ttls.h`

**struct ttls_tunnel_t** This header-file defines the data-structure of an TTLS-tunnel. The description of each item was copied from the comments in the file, if available.

#### Original attributes

**VALUE_PAIR* username** The username (extracted from the EAP-Identity).

**VALUE_PAIR* state** The state of the authentication.

**VALUE_PAIR* reply** Storage for the tunneled reply.

**int authenticated** Used for MS-CHAP2-Successes.

**int default_eap_type** Type-ID of the default tunneled EAP-type.

**int copy_request_to_tunnel** Use SOME of the request attributes from outside of the tunneled session in the tunneled request.

**int use_tunneled_reply** Use the reply attributes from the tunneled session in the non-tunneled reply to the client.

**const char* virtual_server** Virtual server for inner tunnel session.

#### Added attributes

**const char* tnc_virtual_server** The virtual server for EAP-TNC as the second inner method.

**VALUE_PAIR* auth_reply** A cache storage of the last reply of the **first** inner method.

**int auth_code** A cache storage for the reply-code of the **first** inner method.

**int doing_tnc** The status, if currently doing EAP-TNC.

### 5.2.2 File `rlm_eap_ttls.c`

**struct rlm_eap_ttls_t** This file defines the configuration-items of the EAP-TTLS-module, and how they are parsed and initialized. The configuration-file is /usr/local/etc/raddb/eap.conf as default.

#### Original configuration-items

**char* default_eap_type_name** Default tunneled EAP type (by its name).

**int default_eap_type** Type-ID of the default tunneled EAP type.

**int use_tunneled_reply** Use the reply attributes from the tunneled session in the non-tunneled reply to the client.

**int copy_request_to_tunnel** Use SOME of the request attributes from outside of the tunneled session in the tunneled request.

**char* virtual_server** Virtual server for inner tunnel session.

#### Added configuration-items

**char* tnc_virtual_server** Virtual server for the second inner tunnel method, which is EAP-TNC.

**static int eapttls_authenticate(void* arg, EAP_HANDLER* handler)** The file RLM_EAP_TTLS.C also implements the main-authentication-method for EAP-TTLS, which handles the establishment of the EAP-TLS-tunnel and then forwards the request to the method that handles the inner authentication method(s), which will be described in the next section.

### 5.2.3 File `ttls.c`

This file implements the processing of the inner authentication method(s), which means the methods inside the TLS-tunnel.

**int eapttls_process(EAP_HANDLER* handler, tls_session_t* tls_session)**

**Original behaviour** This method processes the tunneled method. Therefore it creates a fake-packet and tries to look up a username. First it looks in the incoming request, then in the tunnel-data, and at last in the EAP-message if it's an EAP-Identity. After that it copies some of the request attributes from outside the tunnel to inside the tunnel (if configured to do so). Then it sets the virtual server, which has to process the tunneled authentication (if configured, else it is the DEFAULT virtual server). The next step is the concrete authentication, which is done by calling the authenticate-method of FreeRADIUS with the fake-packet. Afterwards, the reply is used to determine the result-code of the method.

**Added behaviour** The method was changed in two places to allow two inner methods in sequence. It now checks if EAP-TNC as a second inner authentication method is running, and then sets the virtual server to the configured server for EAP-TNC. This causes the authentication of the request to use the virtual server *inner-tunnel-second*, which is configured to only allow EAP-TNC-packets.

The second change is before the reply is used to determine the result-code of the method. First, it is checked if an virtual server for EAP-TNC is configured. Otherwise, only the first inner method is done.

Then, it is checked if the result of the last request was an PW_AUTHENTICATION_ACK and if TNC as a second inner method is not running. If so, the value pairs and the code of that request is cached, and afterwards the method *start_tnc* is called.

Then it is checked, if TNC as a second method is running and the result of the last request was either PW_AUTHENTICATION_ACK or PW_AUTHENTICATION_REJECT. If so, the method *stop_tnc* is called.

**static REQUEST* start_tnc(EAP_HANDLER* handler, ttls_tunnel_t* t)** Starts EAP-TNC as a second inner method. Creates a new fake-request out of the original incoming request (via EAP_HANDLER). Then it creates a new EAP-START-packet with the code = PW_EAP_REQUEST and the type of EAP-TNC. This message is then processed by *rad_authenticate*, which then calls the the EAP-module. It recognizes

the EAP-START-packet and sets the type of the request to EAP-Identity. At the end, an Access-Challenge is send to the supplicant, which is an EAP-Identity-Request.

**static REQUEST* stop_tnc(EAP_HANDLER* handler, ttls_tunnel_t* t)** Stops EAP-TNC as a second inner method. It copies the value pairs from the cached Access-Accept of the first inner method to the Access-Accept/Reject package of EAP-TNC.

## 5.3 Raw flow of operations

This chapter will describe the flow of operations during an ongoing EAP-TTLS authentication with two inner methods in general.

1. The supplicant connects to the network and runs `wpa_supplicant`.
2. Regarding to its configuration, `FreeRADIUS` runs EAP-TTLS as it is the default EAP type. Therefore it calls the appropriate authenticate-method of the EAP-TTLS-module.
3. The module itself first starts a EAP-TLS-session to establish the tunnel.
4. When the TLS-handshake is done, the tunnel is ready for the inner methods to take place.
5. The first inner method is started, by sending an EAP-Request with the configured default EAP-type of the EAP-TTLS-module to the supplicant.
6. The first inner method is processed until it is finished.
7. If the first method was successful, the Access-Accept, which would be send to the supplicant, is intercepted and cached. Otherwise, the second method won't start, as a successful user authentication is mandatory.
8. Then, a new EAP-TNC-authentication inside the TLS-tunnel is started, by sending a fake-EAP-Start-packet to the virtual server that handles EAP-TNC. The corresponding EAP-message is build manually and is created as an EAP-Response with the type EAP-TNC and the data-length of zero. This triggers the EAP-module to change that to an EAP-Identity-packet, which then starts EAP-TNC.
9. The EAP-TNC handshake then takes place until it is finished.
10. When the EAP-TNC-authentication is finished, the value-pairs of the cached Access-Accept of the first inner method are copied to the current request.
11. The modified Access-Accept or respectively Access-Reject is then send to the PEP.
12. The supplicant is now authenticated both as by its user and by its platform.

## 5.4 Detailed flow of operations

In this chapter, the flow of operations inside `FreeRADIUS` is shown with the help of the debug-output and the corresponding sourcecode. In some cases, only the code for the debug-output itself and a general description is given, but with the given sourcecode-file and method name, a more precise analysis of the code is possible.

### Start of EAP-TTLS

1. **Debug-Output:**
   `Ready to process requests.`

   **Description:**
   First output after initialization. FreeRADIUS now runs in its main loop until exit.

   **Source-Code:**
   MAIN/EVENT.C, *event_status()*:

   ```
   DEBUG("Ready to process requests.");
   ```

2. **Debug-Output:**
   `rad_recv: Access-Request packet from host 192.168.1.6 port 1024, id=52, length=217`

   **Description:**
   First package from supplicant is received.

   **Source-Code:**
   MAIN/LISTEN.C, *stats_socket_recv()* and LIB/RADIUS.C, *rad_recv()*:

   ```
   DEBUG("rad_recv: %s packet from host %s port %d", ...)
   DEBUG(", id=%d, length=%d\n", ...)
   ```

3. **Debug-Output:**
   `+- entering group authorize ...`

   **Description:**
   At first, the request is processed by all configured authorizing-instances. These are defined in the AUTHORIZE-section in the DEFAULT-virtual server-configuration.

   **Source-Code:**
   MAIN/MODULES.C, *indexed_modcall(int, int, REQUEST*)* and MAIN/MODCALL.C, *modcall(int, modcallable*, REQUEST*)*:

```
RDEBUG2("%.*s- entering %s %s {...}",
```

4. **Debug-Output:**
   `++[preprocess] returns ok`

   **Description:**
   The preprocess-module sanitizes the packet. It returns `RLM_MODULE_OK`, which means that the next authorization module is called.

   **Source-Code:**
   RLM_PREPROCESS/RLM_PREPROCESS.C, *preprocess_authorize(void*, REQUEST*)*:

```
myresult = call_modsingle(child->method, sp, request,
default_component_results[component]);
        handle_result:
                RDEBUG2("%.*s[%s] returns %s",
```

5. **Debug-Output:**
   ```
   [eap] EAP packet type response id 1 length 12
   [eap] No EAP Start, assuming it's an on-going EAP conversation
   ++[eap] returns updated
   ```

   **Description:**
   The EAP-module and its authorize-method handles EAP-Start-messages and the extraction of the username out of the EAP-Identity-packet. It returns *RLM_MODULE_UPDATED*, as the packet has to be further processed.

   **Source-Code:**
   RLM_EAP/RLM_EAP.C, *eap_authorize(void*, REQUEST*)*:

```
RDEBUG2("EAP packet type %s id %d length %d", ...);
```

   RLM_EAP/RLM_EAP.C, *eap_start()*:

```
RDEBUG2("No EAP Start, assuming it's an on-going EAP conversation");
```

6. **Debug-Output:**
   `[files] users:  Matched entry tncuser at line 167`

`++[files] returns ok`

**Description:**
The files-module searches the users-file for the current username (`tncuser`).

**Source-Code:**
RLM_FILES/RLM_FILES.C, *file_authorize(void*, REQUEST*)* and RLM_FILES/RLM-_FILES.C, *file_common(...)*:

```
RDEBUG2("%s: Matched entry %s at line %d", ...)
```

### Establishing the tunnel for EAP-TTLS

1. **Debug-Output:**
   `+- entering group authenticate ...`

   **Description:**
   The authenticate-section of the policy is used on the packet.

   **Source-Code:**
   MAIN/MODULES.C, *indexed_modcall(int, int, REQUEST*)* and MAIN/MODCALL.C, *modcall(int, modcallable*, REQUEST*)*:

   ```
   RDEBUG2("%.*s- entering %s %s {...}",
   ```

2. **Debug-Output:**
   ```
   [eap] EAP Identity
   [eap] processing type tls
   [tls] Initiate
   [tls] Start returned 1
   ++[eap] returns handled
   ```

   **Description:**
   As EAP-TTLS is configured as the default type for EAP in FreeRADIUS, it first starts a TLS-authentication to establish the tunnel.

   **Source-Code:**
   EAP.C, *eaptype_select()*:

   ```
   switch(eaptype->type) {
   case PW_EAP_IDENTITY:
   RDEBUG2("EAP Identity");
   ```

   EAP.C, *eaptype_call()*:

   ```
   RDEBUG2("processing type %s", atype->typename);
   ```

   RLM_EAP_TLS.C, *eaptls_initiate*:

   ```
   RDEBUG2("Initiate");
   RDEBUG2("Start returned %d", status);
   ```

3. **Debug-Output:**

```
Sending Access-Challenge of id 52 to 192.168.1.6 port 1024
EAP-Message = 0x010200061520
Message-Authenticator = 0x00000000000000000000000000000000
State = 0xa0b5f0a6a0b7e5e1ed41351c180f735a
Finished request 0.
```

**Description:**
The first EAP-TTLS-packet which is send to the supplicant.

**Source-Code:**
LIB/RADIUS.C, *rad_send(...)*:

```
DEBUG("Sending %s of id %d to %s port %d\n", ...)
```

MAIN/EVENT.C, *request_post_handler(...)*:

```
RDEBUG2("Finished request %d.", request->number);
```

### Running EAP-MD5 as the first inner method

1. **Debug-Output:**
   ```
   +- entering group authenticate {...}
   [eap] Request found, released from the list
   [eap] EAP/ttls
   [eap] processing type ttls
   [ttls] Authenticate
   [ttls] processing EAP-TLS
   [ttls] eaptls_verify returned 7
   [ttls] Done initial handshake
   [ttls] eaptls_process returned 7
   [ttls] Session established.  Proceeding to decode tunneled attributes.
   ```

   **Description:**
   The TLS-tunnel is established. Now the inner method(s) are processed.

   **Source-Code:**
   RLM_EAP_TTLS/RLM_EAP_TTLS.C, *eapttls_authenticate()*:

   ```
   RDEBUG2("Session established.  Proceeding to decode tunneled attributes.");
   ```

2. **Debug-Output:**
   ```
   [ttls] Setting default EAP type for tunneled EAP session.
   ```

   **Description:**
   The configured EAP-type for the first inner authentication method is set (which is EAP-MD5 in this case).

   **Source-Code:**
   RLM_EAP_TTLS/TTLS.C, *eapttls_process()*:

   ```
   if (!fake->username) {
           if (!t->username) {
                   if (t->default_eap_type != 0) {
                           RDEBUG("Setting default EAP type for tunneled EAP
                                session.");
                           vp = paircreate(PW_EAP_TYPE, PW_TYPE_INTEGER);
                           rad_assert(vp != NULL);
                           vp->vp_integer = t->default_eap_type;
                           pairadd(&fake->config_items, vp);
   ```

3. **Debug-Output:**
   ```
   [ttls] Sending tunneled request
   EAP-Message = 0x0200000c01746e6375736572
   FreeRADIUS-Proxied-To = 127.0.0.1
   User-Name = "tncuser"
   ```

   **Description:**
   The packet is proxied to the virtual server configured for EAP-TTLS.

   **Source-Code:**
   RLM_EAP_TTLS/TTLS.C, *eapttls_process()*:

   ```
   if ((debug_flag > 0) && fr_log_fp) {
           RDEBUG("Sending tunneled request");
           debug_pair_list(fake->packet->vps);
   ```

4. **Debug-Output:**
   ```
   server inner-tunnel {
   ```

   **Description:**
   The virtual server for the inner method (configured in /USR/LOCAL/ETC/RADDB-/SITES-ENABLED/INNER-TUNNEL) starts its processing.

   **Source-Code:**
   MAIN/EVEN.C, *radius_handle_request(...)*:

   ```
   if (request->server) RDEBUG("server %s \{",
   ```

5. **Debug-Output:**
   ```
   +- entering group authorize {...}
   [eap] EAP packet type response id 0 length 12
   [eap] No EAP Start, assuming it's an on-going EAP conversation
   ++[eap] returns updated
   [files] users:  Matched entry tncuser at line 167
   ++[files] returns ok
   Found Auth-Type = EAP
   ```

   **Description:**
   Same processing as in section 5.4.

6. **Debug-Output:**
   `+- entering group authenticate {...}`
   `[eap] EAP Identity`
   `[eap] processing type md5`
   `rlm_eap_md5:  Issuing Challenge`
   `++[eap] returns handled`

   **Description:**
   EAP-MD5 is started; returns *RLM_MODULE_HANDLED*, as the packet has NOT to be further processed.

   **Source-Code:**
   EAP.C, *eaptype_select()*:

```
switch(eaptype->type) \{
case PW_EAP_IDENTITY:
RDEBUG2("EAP Identity");
```

   EAP.C, *eaptype_call()*:

```
RDEBUG2("processing type %s", atype->typename);
switch (handler->stage) {
        case INITIATE:
                if (!atype->type->initiate(atype->type_data, handler))
                        rcode = 0;
                break;
```

   RLM_EAP_MD5.C, *md5_initiate*:

```
DEBUG2("rlm_eap_md5: Issuing Challenge");
```

7. **Debug-Output:**
   `} # server inner-tunnel`

   **Description:**
   The virtual server ends its processing, as EAP-MD5 returned the state, that the request is fully processed..

   **Source-Code:**
   MAIN/EVENT.C, *radius_handle_request(...)*:

```
if (request->server) RDEBUG("} # server %s", ...)
```

### Finishing EAP-MD5 and starting EAP-TNC

1. **Debug-Output:**
   `[ttls] Reply-Code of the first inner method was:`
   `2 (PW_AUTHENTICATION_ACK)`
   `EAP-TNC as second inner authentication method starts now`

   **Description:**
   The Access-Accept of the first method is intercepted and the second method is started. The value-pairs of the Access-Accept are copied and stored in a field in the tunnel-structure (`ttls_tunnel_t`). Inside the *start_tnc()*-method,

   **Source-Code:**
   RLM_EAP_TTLS/TTLS.C, *eapttls_process(EAP_HANDLER*, tls_session_t*)*:

```
if (t->tnc_virtual_server) {
        if (fake->reply->code == PW_AUTHENTICATION_ACK
                && t->doing_tnc == FALSE) {
                        RDEBUG2("Reply-Code of the first inner method was: %d
                            (PW_AUTHENTICATION_ACK)", fake->reply->code);
```

   RLM_EAP_TTLS/TTLS.C, *start_tnc(EAP_HANDLER*, ttls_tunnel_t*)*:

```
RDEBUG2("EAP-TNC as second inner authentication method starts now");
```

2. **Debug-Output:**
   `Got tunneled request`
   `FreeRADIUS-Proxied-To = 127.0.0.1`

   **Description:**
   The new request is marked as fake-request.

   **Source-Code:**
   RLM_EAP_TTLS/TTLS.C, *start_tnc(EAP_HANDLER*, ttls_tunnel_t*)*:

```
VALUE_PAIR *vp;
vp = pairmake("Freeradius-Proxied-To", "127.0.0.1", T_OP_EQ);
if (vp) {
        pairadd(&fake->packet->vps, vp);
}

if ((debug_flag > 0) && fr_log_fp) {
        RDEBUG("Got tunneled request");

        debug_pair_list(fake->packet->vps);
}
```

3. **Debug-Output:**
   No debug-output, only the source-code will be described here.

   **Description:**
   This code creates a new request out of the original Access-Accept of the first inner method. It sets the processing virtual server to the second EAP-module-instance, `eap_tnc`, so that only EAP-TNC-packets are allowed in the second inner authentication. Therefore it creates a new EAP-Start-packet with type EAP-TNC and length = 0. This will trigger the EAP-module to recognize a EAP-Start-packet and then send a EAP-Identity-Request to the supplicant.

   **Soure-Code:**
   RLM_EAP_TTLS/TTLS.C, *eapttls_process(EAP_HANDLER*, tls_session_t*)*:

```
REQUEST* fake = request_alloc_fake(request);

fake->server = t->tnc_virtual_server;

VALUE_PAIR *eap_msg;
eap_msg = paircreate(PW_EAP_MESSAGE, PW_TYPE_OCTETS);

eap_msg->vp_octets[0] = PW_EAP_RESPONSE;
eap_msg->vp_octets[1] = 0x00;
eap_msg->vp_octets[4] = PW_EAP_TNC;
eap_msg->length = 0;

pairadd(&(fake->packet->vps), eap_msg);
```

4. **Debug-Output:**
   `+- entering group authorize {...}`
   `++[preprocess] returns ok`

   **Description:**
   The authorize-policy is applied to the new fake-request by the virtual server for EAP-TNC.

   **Source-Code:**
   TTLS.C, *start_tnc()*:

```
rad_authenticate(fake);
```

5. **Debug-Output:**
   `[eap_tnc] Got EAP_START message`

   **Description:**
   The EAP-Start-packet is recognized and changed to an EAP-Identity-packet. It is then send to the supplicat as an Access-Challenge.

   **Source-Code:**
   RLM_EAP/EAP.C, *eap_start()*:

```
if ((eap_msg->length == 0) || (eap_msg->length == 2)) {
        RDEBUG2("Got EAP_START message");
```

6. **Debug-Output:**
   `++[eap_tnc] returns handled`

   **Description:**
   Returns *RLM_MODULE_HANDLED*, as the packet has NOT to be further processed.

   **Source-Code:**
   MAIN/MODCALL.C, *int modcall(int component, modcallable *c, REQUEST *request)*:

```
handle_result:
        RDEBUG2("%.*s[%s] returns %s",
                stack.pointer + 1, modcall_spaces,
                child->name ? child->name : "",
                fr_int2str(rcode_table, myresult, "??"));
```

7. **Debug-Output:**
   `[ttls] Got tunneled Access-Challenge`

   **Description:**
   The EAP-Identity-Request is send to the client.

   **Source-Code:**
   TTLS.C, *process_reply()*:

```
switch (reply->code) {
        case PW_ACCESS_CHALLENGE:
                RDEBUG("Got tunneled Access-Challenge");
                rcode = RLM_MODULE_HANDLED;
```

### Ongoing EAP-TNC authentication

1. **Debug-Output:**

```
server inner-tunnel-second {
+- entering group authorize {...}
++[preprocess] returns ok
[eap_tnc] EAP packet type response id 1 length 12
[eap_tnc] No EAP Start, assuming it's an on-going EAP conversation
++[eap_tnc] returns updated
Found Auth-Type = eap_tnc
+- entering group authenticate {...}
[eap_tnc] EAP Identity
[eap_tnc] processing type tnc
tnc_initiate:  1268134592
```

   **Description:**
   The virtual server for the second inner method is used → *inner-tunnel-second*. That one uses the second instance of the EAP-module (*eap_tnc*) to process the incoming request. At the end, the EAP-TNC-module is called the first time.

   **Source-Code:**
   RLM_EAP_TNC.C, *tnc_initiate()*:

```
DEBUG("tnc_initiate: %ld", handler->timestamp);
```

2. **Debug-Output:**

```
++[eap_tnc] returns handled
} # server inner-tunnel-second
[ttls] Got tunneled reply code 11
```

   **Description:**
   The virtual server for the second inner method has finished the processing of the request. Afterwards, the reply is processed by the EAP-TTLS-module.

   **Source-Code:**
   TTLS.C, *eapttls_process()*:

```
if ((debug_flag > 0) && fr_log_fp) {
        fprintf(fr_log_fp, "} # server %s\n",
                (fake->server == NULL) ? "" : fake->server);
        RDEBUG("Got tunneled reply code %d", fake->reply->code);

        debug_pair_list(fake->reply->vps);
}
```

### Finishing EAP-TNC and sending the Access-Accept packet

1. **Debug-Output:**
   `++[eap_tnc] returns ok`

   **Description:**
   Returns *RLM_MODULE_OK*, as the packet has to be further processed by the post-authenticate-section.

   **Source-Code:**
   MAIN/MODCALL.C, *int modcall(int component, modcallable *c, REQUEST *request)*:

```
handle_result:
        RDEBUG2("%.*s[%s] returns %s",
                stack.pointer + 1, modcall_spaces,
                child->name ? child->name : "",
                fr_int2str(rcode_table, myresult, "??"));
```

2. **Debug-Output:**

```
+- entering group post-auth {...}
++?  if (control:TNC-Status == "Access")
?  Evaluating (control:TNC-Status == "Access") -> TRUE
++?  if (control:TNC-Status == "Access") -> TRUE
++- entering if (control:TNC-Status == "Access") {...}
+++[reply] returns noop
++- if (control:TNC-Status == "Access") returns noop
++ ...  skipping elsif for request 10:  Preceding "if" was taken
} # server inner-tunnel-second
```

   **Description:**
   The post-authenticate subsection of the virtual-server for EAP-TNC; as the attribute *TNC-Status* is set to either *access* or *isolate* by the EAP-TNC-module, it sets the VLAN-settings in the Access-Accept.

   **Source-Code:**
   MAIN/MODCALL.C, *modcall(...)*:

```
if (radius_evaluate_condition(request, myresult, 0, &p, TRUE, &condition)) {
        RDEBUG2("%.*s? %s %s -> %s",
                stack.pointer + 1, modcall_spaces,
                child->type == MOD_IF ? "if" : "elsif",
                child->name, (condition != FALSE) ? "TRUE" : "FALSE");
```

3. **Debug-Output:**

```
[ttls] Reply-Code of EAP-TNC as the second inner method was:
2 (PW_AUTHENTICATION_ACK)
EAP-TNC as second inner authentication method stops now
```

**Description:**
The result of the second authentication is success, so both inner authentications were successfull.

**Source-Code:**
RLM_EAP_TTLS/TTLS.C, *eapttls_process(EAP_HANDLER*, tls_session_t*:*

```
} else if (t->doing_tnc == TRUE
        && (fake->reply->code == PW_AUTHENTICATION_ACK || fake->reply->code ==
              PW_AUTHENTICATION_REJECT)) {
                 RDEBUG2("Reply-Code of EAP-TNC as the second inner method was:
                       %d (%s)", fake->reply->code,
                          fake->reply->code == PW_AUTHENTICATION_ACK ? "
                               PW_AUTHENTICATION_ACK" : "
                               PW_AUTHENTICATION_REJECT");
```

RLM_EAP_TTLS/TTLS.C, *stop_tnc(...)*:

```
RDEBUG2("EAP-TNC as second inner authentication method stops now");
```

4. **Debug-Output:**
   `[ttls] Got tunneled Access-Accept`
   `++[eap] returns ok`

   **Description:**
   The EAP-TTLS-module gets the fake-request back and processes the reply-content. Afterwards the EAP-module is finished, as the request was successful.

   **Source-Code:**
   TTLS.C, *process_reply()*:

```
switch (reply->code) {
        case PW_AUTHENTICATION_ACK:
                 RDEBUG("Got tunneled Access-Accept");
                 rcode = RLM_MODULE_OK;
```

5. **Debug-Output:**
   ```
   Sending Access-Accept of id 62 to 192.168.1.6 port 1024
   Message-Authenticator = 0x00000000000000000000000000000000
   User-Name = "tncuser"
   Tunnel-Type:0 = VLAN
   Tunnel-Medium-Type:0 = IEEE-802
   Tunnel-Private-Group-Id:0 = "96"
   EAP-Message = 0x030b0004
   ```

   **Description:**
   The Access-Accept is send to the PEP, with the configured VLAN-settings.

   **Source-Code:**
   LIB/RADIUS.C, *rad_send(...)*:

   ```
   DEBUG("Sending %s of id %d to %s port %d\n", ...)
   ```

## 5.5 Sequence diagrams

The following sequence diagrams show the message flow between the AR, the PEP and the PDP. The IDs of the EAP- and RADIUS-messages correspond to the provided Wireshark-captures.

### Remarks

The (fake) EAP-Response/Identity of the AR in the EAP-packet with ID 26 is created by wpa_supplicant. As a response to the end of the TLS-authentication, a fake EAP-Request/Identity is created inside of wpa_supplicant, to start the inner method. The following debug-output is from wpa_supplicant, showing this behaviour.

```
EAP: Received EAP-Request id=26 method=21 vendor=0 vendorMethod=0
...
EAP-TTLS: TLS done, proceed to Phase 2
EAP-TTLS: Derived key - hexdump(len=64): [REMOVED]
EAP-TTLS: received 0 bytes encrypted data for Phase 2
EAP-TTLS: empty data in beginning of Phase 2 - use fake EAP-Request Identity
EAP-TTLS: Phase 2 EAP Request: type=1
EAP: using real identity - hexdump_ascii(len=7):
     74 6e 63 75 73 65 72                             tncuser
EAP-TTLS: AVP encapsulate EAP Response - hexdump(len=12): 02 00 00 0c 01 74 6e 63
    75 73 65 72
```

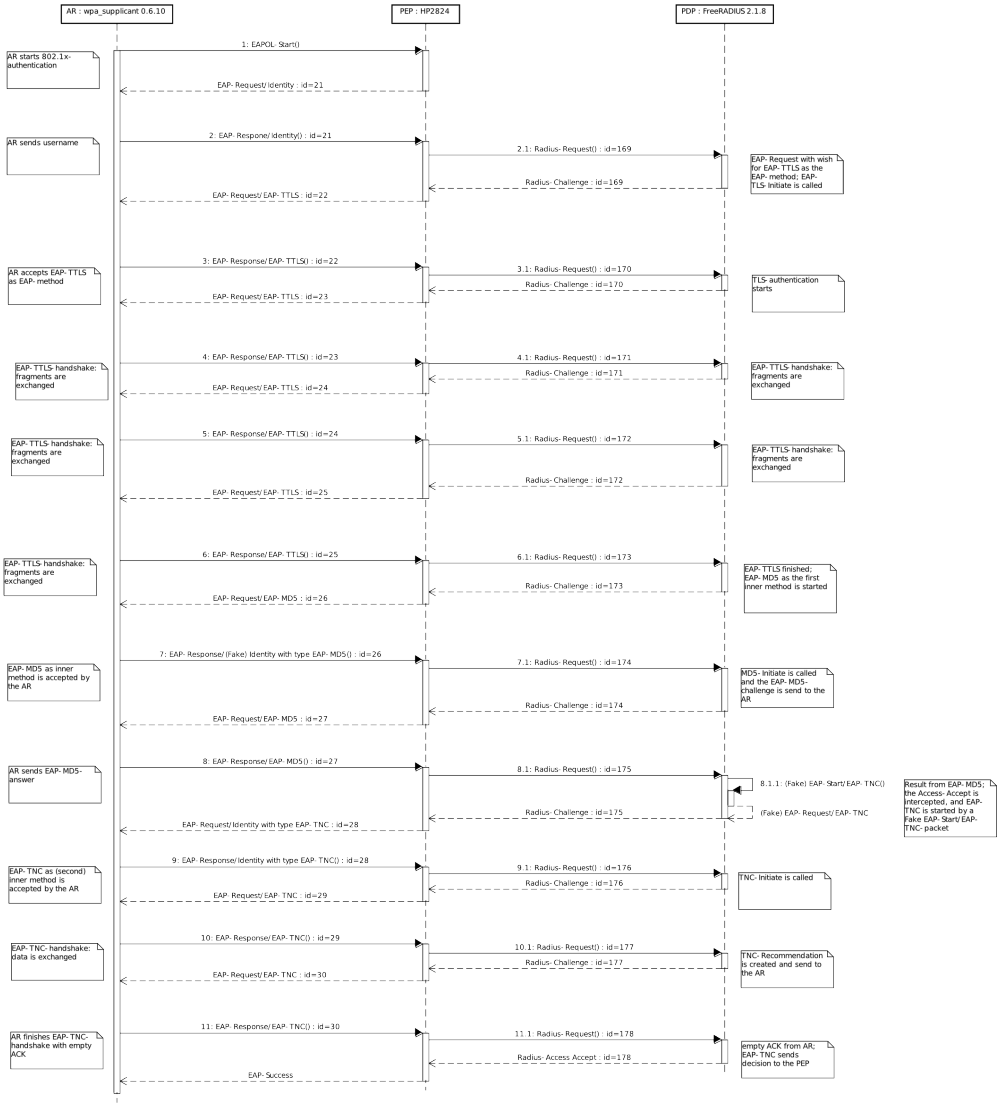


Figure 2: EAP TTLS patch packet flow

## 5.6 Configuration

This chapter describes the configuration of FreeRADIUS and wpa_supplicant for the use of multiple inner authentication methods.

### FreeRADIUS

**/usr/local/etc/raddb/users** The users-file has to contain an entry for a user with a password, so that a user-authentication can take place (EAP-MD5 for example).

```
# User-entry for EAP-MD5
tncuser Cleartext-Password := password
```

**/usr/local/etc/raddb/eap.conf** In the configuration of the EAP-module, several things are important. The default EAP-type has to be TTLS, and there MUST NOT be a configured instance of EAP-TNC. This allows a seperation of the incoming requests, as EAP-TNC-packets are only allowed as the second inner method, and are NOT allowed as the first inner method. Another important setting is the configuration of the virtual-servers for the EAP-TTLS-module. At the end, a second instance of the EAP-module has to be configured, with EAP-TNC as the default EAP-type and a configured EAP-TNC-module.

```
eap {
  # ...
  default_eap_type = ttls
  # ...

  # Comment out the section for TNC
  # tnc {
  # }

  # ...

  ttls {
    default_eap_type = md5
    # ...
    use_tunneled_reply = yes
    #
    virtual_server = "inner-tunnel"
    tnc_virtual_server = "inner-tunnel-second"
  }

  # ...
}

eap eap_tnc {
  default_eap_type = tnc

  tnc {
  }

}
```

**/usr/local/etc/raddb/sites-enabled/inner-tunnel** The configuration of the inner tunnel has to be the same as the default-one delivered with FreeRADIUS.

**/usr/local/etc/raddb/sites-enabled/inner-tunnel-second** This virtual server handles the EAP-TNC-packets of the second inner authentication method. Important is the usage of the second EAP-module instance (eap_tnc) in the *authorize* and *authenticate*-section. In the *post-auth*-section the content is the same as in the default configuration for EAP-TNC, when used as the only inner authenticaton method.

```
# -*- text -*-
######################################################################
#
#       This is a virtual server that handles *only* EAP-TNC
#       requests.
#
#       $Id$
#
######################################################################

server inner-tunnel-second {

  #  Authorization. First preprocess (hints and huntgroups files),
  #  then realms, and finally look in the "users" file.
  #
  #  The order of the realm modules will determine the order that
  #  we try to find a matching realm.
  #
  #  Make *sure* that 'preprocess' comes before any realm if you
  #  need to setup hints for the remote radius server
  authorize {
  #
  #  The preprocess module takes care of sanitizing some bizarre
  #  attributes in the request, and turning them into attributes
  #  which are more standard.
  #
  #  It takes care of processing the 'raddb/hints' and the
  #  'raddb/huntgroups' files.
  #
  #  It also adds the {Client-IP-Address} attribute to the request.
  preprocess

  eap_tnc {
    ok = return
  }
}

#  Authentication.
#
#
#  This section lists which modules are available for authentication.
#  Note that it does NOT mean 'try each module in order'.  It means
#  that a module from the 'authorize' section adds a configuration
#  attribute 'Auth-Type := FOO'.  That authentication type is then
#  used to pick the apropriate module from the list below.
#

#  In general, you SHOULD NOT set the Auth-Type attribute.  The server
#  will figure it out on its own, and will do the right thing.  The
#  most common side effect of erroneously setting the Auth-Type
#  attribute is that one authentication method will work, but the
#  others will not.
#
#  The common reasons to set the Auth-Type attribute by hand
#  is to either forcibly reject the user, or forcibly accept him.
#
authenticate {
  #
  #  Allow EAP-TNC authentication.
```

```
    eap_tnc
}

#  Session database, used for checking Simultaneous-Use. Either the radutmp
#  or rlm_sql module can handle this.
#  The rlm_sql module is *much* faster
session {
  radutmp
}

#  Post-Authentication
#  Once we KNOW that the user has been authenticated, there are
#  additional steps we can take.
post-auth {
  if (control:TNC-Status == "Access") {
    update reply {
      Tunnel-Type := VLAN
      Tunnel-Medium-Type := IEEE-802
      Tunnel-Private-Group-ID := 96
    }
  }
  elsif (control:TNC-Status == "Isolate") {
    update reply {
      Tunnel-Type := VLAN
      Tunnel-Medium-Type := IEEE-802
      Tunnel-Private-Group-ID := 97
    }
  }

# Note that we do NOT assign IP addresses here.
# If you try to assign IP addresses for EAP authentication types,
# it WILL NOT WORK.  You MUST use DHCP.
#
#  Access-Reject packets are sent through the REJECT sub-section of the
#  post-auth section.
#
#  Add the ldap module name (or instance) if you have set
#  'edir_account_policy_check = yes' in the ldap module configuration
#
  Post-Auth-Type REJECT {
    attr_filter.access_reject
  }
}

} # inner-tunnel-second block
```

### wpa_supplicant.conf

The identity and password have to be the same as in the users-configuration on FreeRADIUS.

```
network={
  ssid="eap_ttls"
  key_mgmt=IEEE8021X
  eap=TTLS
  identity="tncuser"
  password="password"
  ca_cert="/home/tncuser/tnc/certs/ca.pem"
  id_str=""
}
```

# 6 naaeap

## 6.1 General Information

The naaeap library processes the TNC specific data that was extracted out of the EAP-TNC packet by the EAP-TNC module of FreeRADIUS / forwards TNC data for encapsulation within EAP to that module. The main purpose of naaeap is to handle fragmentation and establish the communication context to the tncs module.

## 6.2 Architecture

naaeap is provided as shared library. The classes contained are basically separated into classes providing logic and into classes providing simple entities. Since naaeap is used by the EAP-TNC module (which is written in C), naaeap exposes a C interface defined in `naaeap.h`. The interface includes functions for initialization, connection management and the processing of TNC data. Most of the calls (except those that deal with Fragmentation) are forwarded to the tncs module.

# 7 tncs

## 7.1 General Information

The tncs receives incoming TNCCS and IMCIMV messages. TNCCS messages (like preferred language) are directly processed by the tncs. IMCIMV messages are forwarded to the corresponding IMVs. Routing is based upon the respective message types. The tncs is responsible for correctly loading/initializing/calling and terminating all IMVs (those that are specified in the tnc_config file). Currently, tncs supports only IF-TNCCS 1.1. Approaches to integrate IF-TNCCS-SoH have not been finished yet.

## 7.2 Architecture

The TNC server implementation of TNC@FHH. Exposes a C++ interface that is used by naaeap. The C++ interface is defined in `Coordinator.h`. It includes only few methods. The most important ones deal with initialization, connection management and the processing of TNC data.

# 8 imunit

## 8.1 General Information

imunit 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

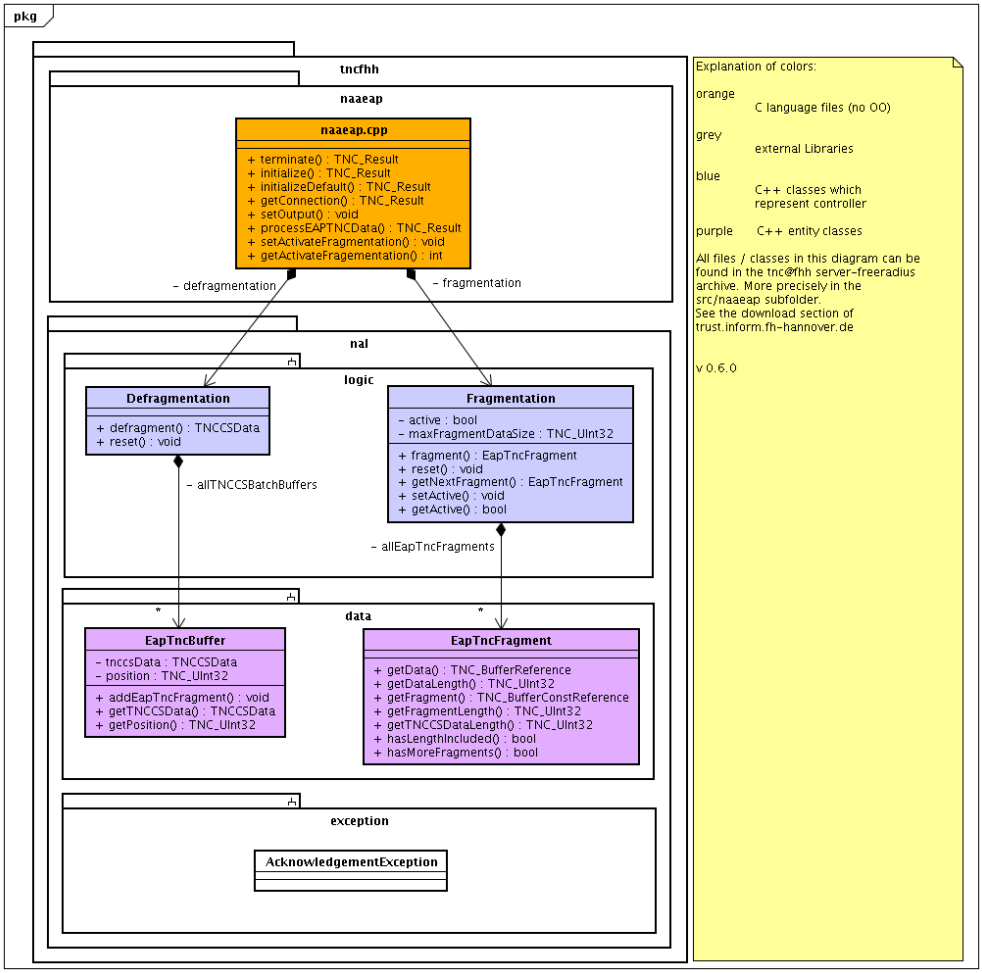


Figure 3: naaeap class diagram (overview)

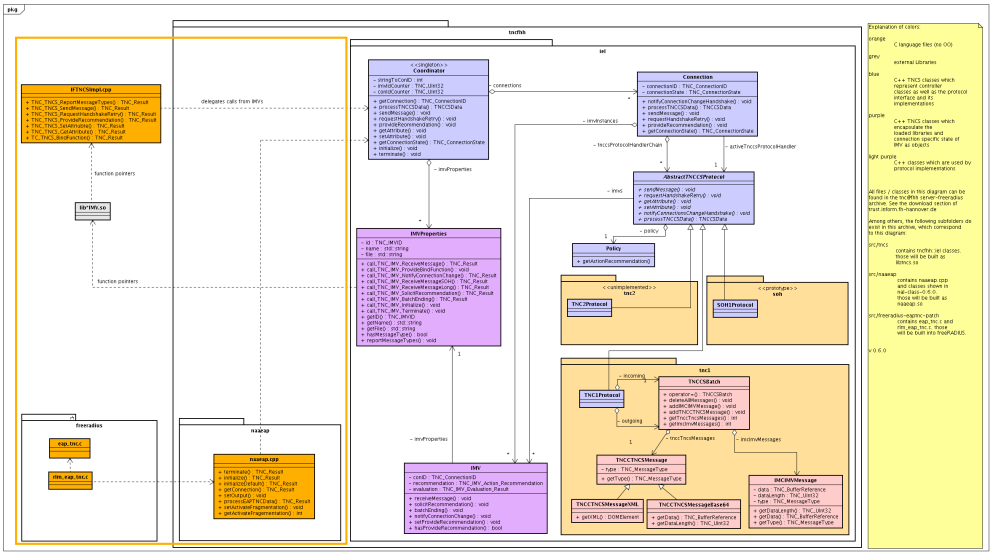

Figure 4: tncs class diagram (overview)

not supported. However, we assume that Windows-support would need only minor modifications of the build process.[5]

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)

To demonstrate the use of imunit for the development of new IMC/V pairs, TNC@FHH contains a very basic exampleimc/v pair.

## 8.2 Architecture

Figure 5 gives an overview of the imunit package. The components and their functions will be described in the following.

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

[5] Appropriate patches are very welcome ...

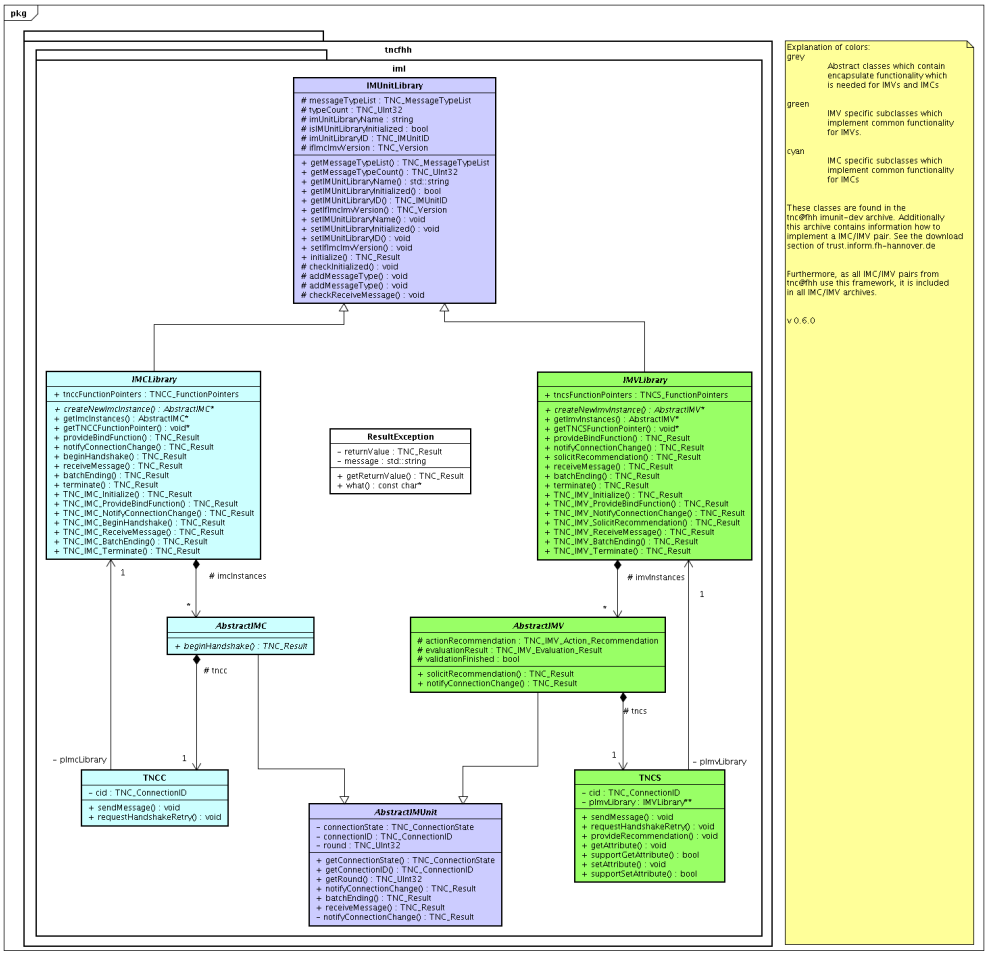


Figure 5: imunit class diagram (overview)

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 9.2.

## 8.3 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.

## 8.4 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.

## 8.5 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.

# 9 IMC/V Pairs

## 9.1 TNC@FHH IMC/V Pairs Message Types

TNC@FHH IMC/V pairs use the following message types:

- vendord id (FHH IANA PEN): 0x0080ab
- example: 0xfe
- dummy: 0x31
- clamav: 0x41
- platid: 0x33
- attestation: 0x34
- hostscanner: 0x30

## 9.2 example

### 9.2.1 General Information

This is a helloworld example for an IMC/V that is implemented based upon imunit. In the remainder of this section, a step-by-step guide that explains how to implement your own IMC/V based upon imunit is given.

### 9.2.2 Architecture

The architecture of the exampleimc/v is very simple. There are just four classes for both the IMC and the IMV:

- ExampleIMCLibrary
- ExampleIMC
- ExampleIMVLibrary
- ExampleIMV

Each class extends the corresponding imunit class. There are no external files, processes or programs used by the exampleimc/v.

### 9.2.3 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[6] (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.

```
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.

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

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.

### 9.2.4 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[7] (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
```

   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);
}
```

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

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.

## 9.3 dummy

### 9.3.1 General Information

This is another helloworld example for an IMC/V that is implemented based upon imunit. The IMC reads a local file, sends the content to the IMV, which provides a recommendation based upon this message.

### 9.3.2 Architecture

The architecture of the dummyimc/v is very simple. The IMC sends the content of a local file to the IMV. The content is either "allow", "isolate", or "none", that correspond to the respective TNC recommendations. The default location of the policy is `/etc/tnc/dummyimc.file`. The IMV uses a policy that defines how many messages it wants to receive before a recommendation is granted. The default location of the policy is `/etc/tnc/dummyimv.policy`.

## 9.4 clamav

### 9.4.1 General Information

The clamav IMC/V checks the status of the anti virus software ClamAV. The IMC/V pair supports the evaluation of the following properties:

- Operational Status of clamd
- Version of clamav
- Version of main
- Version of daily

The baseline for this evaluation is the clamavimv.policy file. This is a simple text file that enables you to specify the desired good values of an endpoint's ClamAV configuration. The file is commented extensively. The policy file is read at the beginning of each new handshake. Remediation is not supported in this version. If the data received from the clamavimc matches the desired values in the policy, an 'ALLOW' recommendation is provided. Otherwise, an 'ISOLATE' recommendation is provided.

### 9.4.2 Architecture

The IMC/V pair consists of four classes that extend those of the imunit framework (like the exampleimc/v from section 9.2). In order to work properly, the clamavimc needs a configuration file. The default location is `/etc/tnc/clamavimc.conf`. The file is commented extensively. The clamavimv needs a policy file where the desired, reference values are specified. The default location of the policy is `/etc/tnc/clamavimv.policy`. The file is commented extensively. The measurement values on the AR are obtained by using the clamconf utility of Clamav. Depending on last clamav update (full or incremental) the filname ending for main and daily can be *.cvd or *.cld. Former versions to 0.7.0 of clamavimc/v did not consider this.

## 9.5 platid

### 9.5.1 General Information

The idea of this IMC/V pair is to enable interoperable platform identification. It implements a simple challenge/response protocol. The IMC signs the challenge with

its RSA private key. The IMV validates if the signature obtained from the IMC is good and if the corresponding public key has been properly registered (e.g. whether it is a known platform or not). The key management is based upon X.509 certificates. As a special feature, the IMC supports to use a TPM protected key. The openssl_tpm_engine package from the TrouSerS project is needed to accomplish this.

The protocol is as follows: The IMV receives a complete X.509 certificate from the IMC. The IMV then sends back a random nonce to the IMC which is encrypted with the RSA private key. The IMV decrypts the encrypted nonce with the public key contained in the X.509 certificate and compares it with the one which was send. Additionally the IMV compares the issuername and fingerprint of the X.509 certificate with entries configured in its policy.

### 9.5.2 Architecture

The IMC is configured via a file. The default location is `/etc/tnc/platidimc.file` It cotains information for the IMC where to find the platform's X.509 certificate and the corresponding RSA private key. The default location of the IMV policy is `/etc/tnc/platidimv.certs`.

The IMC/V pair uses open_ssl for cryptographic operations. If a TPM based key should be used, the IMC additionally needs TrouSerS and the openssl_tpm_engine. Please refer to the TrouSerS documentation for information how to set up the TPM engine.

## 9.6 attestation

### 9.6.1 General Information

The attestation IMC/V pair implements a simple protocol for binary attestation. The integrity of the AR is verified based on its TPM PCR values.

The following steps give a brief overview of the imc/v handshake:

- Step 1: IMC sends its AIK X.509 certificate.
- Step 2: IMV verifies AIK certificate.
- Step 3: IMV sends PCR_SELECTION to IMC.
- Step 4: IMC generates TPM Quote based on PCR_SELECTION and sends it back to IMV.
- Step 5: IMV generates separate TPM Quote based on PCR reference values.
- Step 6: IMV compares own TPM Quote with IMC TPM Quote and verifies IMC TPM Quote signature.
- Step 7: According to operation mode, step 3-6 can be repeated several times.
- Step 8: Depending on validation in step 6, IMV provides "ALLOW" or "NO_ACTION" recommendation.

TPM PCR values are SHA-1 hashes which represent available hard- and software on a computer system, as well as the configuration thereof. These values are usually determined during system start (trusted boot). TrustedGRUB extends the open-source bootloader GNU GRUB with TPM support to measure the binary configuration (i.e., the identity) of modules to be loaded. Further information can be found at http://sourceforge.net/projects/trustedgrub/. Therefore a working TPM is needed as well as a Trusted Software Stack (TSS) for TPM access, e.g. TrouSerS.

Attestation Identity Keys (AIKs) are used to ensure that PCR values (a TPM Quote) come from a trustworthy TPM. Due to the fact that current version of attestationIMC/V is not capable of verifying a CA certificate path, a fingerprint of all used AIK credentials must be made known to the IMV.

### 9.6.2 Architecture

IMV policy is located at `/etc/tnc/attestationimv.policy` and contains a list of PCRs with their associated, desired reference values. Two modes of operation are available. `quoteType=single` requests PCRs sequentially one after another. `quoteType=complete` asks the IMC to quote all given PCRs at once. Single quote mode allows you to determine which PCR value does not match the policy but also causes one batch per PCR value, resulting in a higher amount of network traffic. The TPM Quote is basically a SHA-1 hash of the TCPA_QUOTE_INFO structure. TPM quoting request is performed by the corresponding calls to the TSS, e.g. TrouSerS since version 0.3.4 is needed. This version of attestationIMV does not support different policies (PCR values) per AR. To prevent replay attacks a random 20 byte nonce is also part of each PCR_SELECTION.

IMC configuration is located at `/etc/tnc/platidIMC.file` and contains a path to the X509 certificate and its corresponding private-key (AIK). Currently, tools from http://www.privacyca.com are used to create AIKs. The code from `identity.c` creates a AIK blob and a corresponding X.509 certificate. The SHA-1 hash from this certificate can be extracted by using `x509` tool from OpenSSL.

## 9.7 hostscanner

### 9.7.1 General Information

The hostscannerimc/v scans for opened ports on an endpoint. The ports to be scanned and the expected status of these ports are defined in the IMVs policy file.

### 9.7.2 Architecture

In the IMV's policy, a list of ports with their desired status can be specified. The default location of the policy is `/etc/tnc/hostscannerimv.policy`. The IMC does not require a configuration file.

# 10 tncsim

## 10.1 General Information

tncsim is a simple program that allows you to test IMC/V pairs locally on one endpoint. tncsim loads IMCs and IMVs and initiates a single TNC handshake. Lots of debug information is printed out, including the binary structure of the TNC packets sent from TNCC to TNCS and vice versa.

## 10.2 Architecture

In its default configuration, tncsim uses libtnc as client and TNC@FHH tncs as server. IMC/V pairs that shall be loaded are specified in `/etc/tnc/tncsim_config`. tncsim can be extended to support arbitrary TNCC and TNCS implementations. For details, please refer to the sourcecode.

# 11 Copyright and License



# 12 Acknowledgement

TNC@FHH is implemented by the Trust@FHH research group of the Fachhochschule Hannover, the University of Applied Sciences and Arts, located in Lower Saxony, Germany. Parts of this work have been carried out within the tNAC research project (support code 1704B08) which is funded by the German Federal Ministry of Education and Research (http://www.bmbf.de/en/index.php).