TCG Trusted Network Connect
TNC IF-IMC

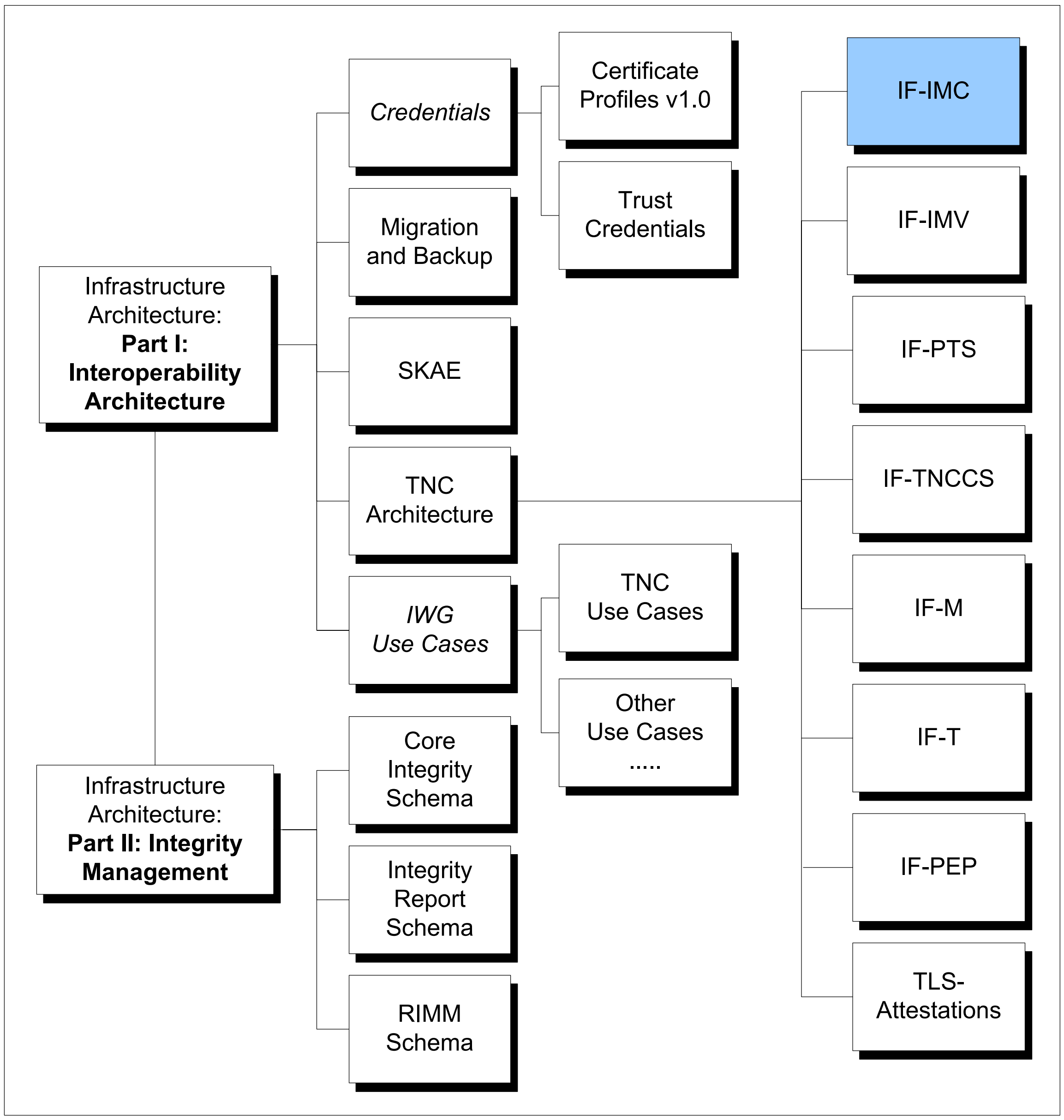
Specification Version 1.1
Revision 5
1 May 2006
Published

Contact: admin@trustedcomputinggroup.org

TCG PUBLISHED

TCG

IWG TNC Document Roadmap



Acknowledgement

The TCG wishes to thank all those who contributed to this specification. This document builds on considerable work done in the various working groups in the TCG.

Special thanks to the members of the TNC contributing to this document:

Mahalingam Mani	Avaya
Mark Beadles	Endforce, Inc.
Kazuaki Nimura	Fujitsu Limited
Boris Balacheff	Hewlett-Packard
Paul Crandell	Hewlett-Packard
Mauricio Sanchez	Hewlett-Packard
Diana Arroyo	IBM
Lee Terrell	IBM
Tina Bird	InfoExpress, Inc.
Ravi Sahita	Intel Corporation
Ned Smith	Intel Corporation
Barbara Nelson	iPass
Steve Hanna (Editor, TNC co-chair)	Juniper Networks, Inc.
Gene Chang	Meetinghouse Data Communications
John Vollbrecht	Meetinghouse Data Communications
Sandilya Garimella	Motorola
Joseph Tardo	Nevis Networks
Thomas Hardjono	SignaCert, Inc.
Babak Salimi	Sygate Technologies, Inc.
Bryan Kingsford	Symantec
Paul Sangster (TNC co-chair)	Symantec
Scott Cochrane	Wave Systems
Greg Kazmierczak	Wave Systems

Table of Contents

1 Scope and Audience

The Trusted Network Connect Sub Group (TNC-SG) is defining an open solution architecture that enables network operators to enforce policies regarding the security state of endpoints in order to determine whether to grant access to a requested network infrastructure. This security assessment of each endpoint is performed using a set of asserted integrity measurements covering aspects of the operational environment of the endpoint. Part of the TNC architecture is IF-IMC, a standard interface between Integrity Measurement Collectors and the TNC Client. This document defines and specifies IF-IMC.

Architects, designers, developers and technologists who wish to implement, use, or understand IF-IMC should read this document carefully. Before reading this document any further, the reader should review and understand the TNC architecture as described in [1].

2 Background

2.1 Purpose of IF-IMC

This document describes and specifies IF-IMC, a critical interface in the Trusted Computing Group's Trusted Network Connect (TNC) architecture. IF-IMC is the interface between Integrity Measurement Collectors (IMCs) and a TNC Client (TNCC). It is closely related to IF-IMV [4], the interface between Integrity Measurement Verifiers (IMVs) and a TNC Server (TNCS).

IF-IMC is primarily used by the TNC Client to gather the security state in the form of integrity measurements from IMCs so they can be communicated to Integrity Measurement Verifiers (IMVs) and to enable message exchanges between the IMCs and the IMVs. These message exchanges occur within Integrity Check Handshakes, each of which is an example of a TCG attestation protocol in the context of the TNC architecture. IF-IMC also allows IMCs to coordinate with the TNC Client as needed. A more detailed description of the features provided by the IF-IMC API is provided in section 2.6.

2.2 Requirements

Here are the requirements that IF-IMC API must meet in order to successfully play its role in the TNC architecture.

- Meets the needs of the TNC architecture

  The API must support all the functions and use cases described in the TNC architecture as they apply to the relationship between the TNC Client and IMC components. The API must support multiple TNCCs on a single AR and multiple overlapping network connections and Integrity Check Handshakes for a single TNCC. The API must allow an IMC to act as a front end for one or more applications or to handle everything within the IMC, as determined by the IMC implementer.

- Secure

  The integrity and confidentiality of communications between an IMC and an IMV must be protected. The TNC Client and TNC Server are assumed to provide a secure communications tunnel between the IMCs and the IMVs. The IMCs and IMVs may choose to add other security mechanisms, but those are out of scope for this document.

  The security requirements for IF-IMC include requirements that unauthorized parties cannot observe communications between the IMC and the TNC Client and that only authorized IMCs can communicate with the TNC Client across IF-IMC. See the Security Consideration section of this document for detailed discussion.

- Efficient

  The TNC architecture delays network access until the endpoint is determined to not pose a security threat to the network based on its asserted integrity information. To minimize user frustration, it is essential to minimize delays and make IMC-IMV communications as rapid and efficient as possible. Efficiency is also important when you consider that some network endpoints are small and low powered.

- Extensible

  IF-IMC will need to expand over time as new features are added to the TNC architecture. For instance, the TNC will soon add support for TPM integration. The IF-IMC API must allow new features to be added easily, providing for a smooth transition and allowing newer and older architectural components to continue to work together.

- Easy to use and implement

  The API should be easy for TNC Client and IMC vendors to use and implement. It should allow them to enhance existing products to support the TNC architecture and integrate legacy code without requiring substantial changes. The API should also make things easy for system administrators and end-users. Components of the TNC architecture should plug together automatically without requiring manual configuration.

- Platform-independent

  Since most or all endpoints on a network will be subject to integrity checks, the IF-IMC API must function on as many platforms as possible. At least Windows, Linux (most common flavors), and other UNIX variants must be supported.

- Language-independent

  The IF-IMC API must support the widest possible variety of languages: C, C++, C#, Java, Visual Basic, assembly language, and others. Therefore, this specification defines an abstract API and language-specific bindings.

2.3 Non-Requirements

Here are certain requirements that the IF-IMC API explicitly is not required to meet. This list is not exhaustive (complete).

- Supporting communications between IMCs and TNCCs written in different languages

  While the IF-IMC API must support the widest possible variety of languages (C, C++, C#, Java, Visual Basic, etc.), it is not required to provide a standard manner for a TNCC written in one language (like C) to load an IMC written in one language (like Java). Depending on the platform and language, this can be quite difficult. When it's not difficult, we will support such interoperation (as with the Microsoft Windows DLL Binding, which supports interoperation among all languages that can easily call and implement a DLL). But support for such cross-language compatibility is not required. Future versions of this API may add such support (perhaps via an RPC mechanism).

2.4 Assumptions

Here are the assumptions that IF-IMC API makes about other components in the TNC architecture.

- Secure Message Transport

  The TNC Client and TNC Server are assumed to provide a secure communications tunnel for messages sent between the IMCs and the IMVs.

- Reliable Message Delivery

  The TNC Client and TNC Server are assumed to provide reliable delivery for messages sent between the IMCs and the IMVs, consistent with the description of message delivery in section 2.7.4 of this specification. In the event that reliable delivery cannot be provided, the TNC Client is expected to terminate the connection.

2.5 Keywords

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [2]. This specification does not distinguish blocks of informative comments and normative requirements. Therefore, for the sake of clarity, note that lower case instances of must, should, etc. do not indicate normative requirements.

2.6 API Naming Conventions

To avoid name conflicts, all identifiers in the IF-IMC API have a name that begins with "`TNC_`".

Functions described in this document that are to be implemented by an IMC have a name that begins with "`TNC_IMC_`". This prefix is followed by words describing the operation performed by the function.

Functions described in this document that are to be implemented by a TNC Client (known as "callbacks") have a name that begins with "`TNC_TNCC_`". This prefix is followed by words describing the operation performed by the function.

Vendor-specific functions MUST have a name that begins with "`TNC_XXX_`" where `XXX` is replaced by the vendor ID of the organization that defined the extension. See section 3.2.4 for more information and requirements on vendor-specific functions.

2.7 Features Provided by IF-IMC

This section documents the features provided by IF-IMC.

2.7.1 Integrity Check Handshake

One of the primary functions of IF-IMC is to enable message exchanges between IMCs and IMVs to share security state allowing the IMVs to factor the integrity of the IMC's security software state into the access control decision. These communications always take place within the context of an Integrity Check Handshake. In such a handshake, the IMCs send a batch of messages (typically, integrity measurements) to the IMVs and the IMVs optionally respond with a batch of messages (remediation instructions, queries for more information, etc.). This dialog may go on for some time until the IMVs decide on their IMV Action Recommendations.

2.7.2 Connection Management

A connection between a TNCC and a TNCS may include several Integrity Check Handshakes: an initial handshake that ends with the endpoint being told to perform remediation such as applying patches (which may involve rebooting the endpoint), a subsequent handshake once the remediation is complete, and sometimes even later handshakes such as when policies change. Handshakes for a given TNCC-TNCS pair cannot be nested. One such handshake must end before another can begin. To optimize and manage handshakes, the TNCC provides connection management features.

When a new TNCC-TNCS relationship is established, the TNCC chooses a network connection ID to refer to that relationship. The TNCC informs the IMCs of the new network connection and updates them whenever the state of the network connection changes. When a network connection is complete, the TNCC notifies the IMCs that the network connection ID will be deleted and then does so. Note that the connection ID is local to the TNCC (like a socket descriptor in UNIX), not shared with the TNCS.

A TNCC SHOULD maintain the same network connection ID across many Integrity Check Handshakes between a particular TNCC-TNCS pair. There are two reasons to maintain a network connection ID beyond a single Integrity Check Handshake. First, this allows the IMCs and IMVs to maintain state information associated with an earlier handshake. Second, it allows an IMC to request a handshake retry for a particular connection, as when the IMC has completed remediation requested by an IMV. A TNCC SHOULD ensure that connection IDs persist long enough to permit handshake retry. Since remediation may require restarting the operating system, power cycling, and other measures, the connection ID SHOULD be remembered even across these measures so that the handshake can be retried after remediation. However, the TNCS MAY refuse to maintain state on old handshakes, forcing a full handshake every time. This is fine. The network connection ID is assigned by the TNCC. Little or no cooperation from the TNCS is required to allow the TNCC to maintain the network connection ID. The TNCC MUST use the same connection ID for all IMCs when referring to a particular connection.

If more than one TNC Client may be running at once on a single machine (rare, but possible) and an IMC is loaded by both TNC Clients, the IMC MUST work properly even if the TNC Clients happen to choose the same network connection ID for different connections. This should not be too hard, since the IMCs will be loaded separately. However, it may become an issue if the IMCs need to communicate with a common application and refer to the network connections.

2.7.3 Remediation and Handshake Retry

In several cases, it is useful to retry an Integrity Check Handshake. First, an endpoint may be isolated until remediation is complete. Once remediation is complete, an IMC can inform the TNCC of this fact and suggest that the TNCC retry the Integrity Check Handshake. Second, a TNCS can initiate a retry of an Integrity Check Handshake (if the TNCS or IMV policies change or as a periodic recheck). Third, an IMC or IMV can request a handshake retry in response to a condition detected by the IMC or IMV (suspicious activity, for instance). In any case, it's generally desirable (but not always possible) to reuse state established by the earlier handshake and to avoid disrupting network connectivity during the handshake retry. IF-TNCCS 1.0 and IF-T 1.0 do not provide any support for handshake retry without disrupting network connectivity but future versions of these specifications will probably do so. In the mean time, proprietary protocols may provide this capability.

To support handshake retries, the TNCC SHOULD maintain a network connection ID after an Integrity Check Handshake has been completed. This network connection ID can then be used by the TNCC to inform IMCs that it is retrying the handshake or by an IMC to request a retry (due to remediation or another reason).

Handshake retry may not always be possible due to limitations in the TNCC, NAR, PEP, or other entities. In other cases, retry may require disrupting network connectivity. For these reasons, IF-IMC supports handshake retry and requires IMCs to handle handshake retries (which is usually trivial) but does not require TNCCs to honor IMC requests for handshake retry. In fact, IF-IMC requires an IMC to provide information about the reason for requesting handshake retry so that the TNCC or TNCS can decide whether it wants to retry (which may disrupt network access).

Note that remediation instructions are delivered from IMVs to IMCs through standard IMV-IMC messages. There is no special support in IF-IMC for this feature. If remediation instructions require network access, IMCs SHOULD NOT follow them until the network connection state changes to success or isolated.

Incompatible IMV policies (more likely with multiple network connections) can cause flip-flopping or other problems if an IMC receives conflicting remediation instructions from different IMVs. IMCs may want to detect this as much as possible and notify the user or administrator or ask them for guidance or refuse to perform remediation that would cause them to flip-flop.

2.7.4 Message Delivery

One of the critical functions of the TNC architecture is conveying messages between IMCs and IMVs. Each message sent in this way consists of a message body, a message type, and a recipient type.

The message body is a sequence of octets (bytes). The TNCC and TNCS SHOULD NOT parse or interpret the message body. They only deliver it as described below. Interpretation of the message body is left to the ultimate recipients of the message, the IMCs or IMVs. A zero length message is perfectly valid and MUST be properly delivered by the TNCC and TNCS just as any other IMC-IMV message would be.

The message type is a four octet number that uniquely identifies the format and semantics of the message. The method used to ensure the uniqueness of message types while providing for vendor extensions is described in section 3.4.2.5. From the perspective of IF-IMC and the TNCC and TNCS, this method is not important. The message type is simply a number.

The recipient type is simply a flag indicating whether the message should be delivered to IMVs or IMCs. Messages sent by IMCs are delivered to IMVs and vice versa so this flag does not appear in IF-IMC. All messages sent by an IMC through IF-IMC have a recipient type of IMV. All messages received by an IMC through IF-IMC have a recipient type of IMC. The recipient type does not show up in IF-IMC or IF-IMV, but it helps in explaining message routing.

The routing and delivery of messages is governed by message type and recipient type. Each IMC and IMV indicates through IF-IMC and IF-IMV which message types it wants to receive. The TNCC and TNCS are then responsible for ensuring that any message sent during an Integrity Check Handshake is delivered to all recipients that have a recipient type matching the message's recipient type and that have indicated the wish to receive messages whose type matches the message's message type. If no recipient has indicated a wish to receive a particular message type, the TNCC and TNCS can handle these messages as they like: ignore, log, etc.

WARNING: The message routing and delivery algorithm just described is not a one-to-one model. A single message may be received by several recipients (for example, two IMVs from a single vendor, two copies of an IMC, or nosy IMVs that monitor all messages). If several of these recipients respond, this may confuse the original sender. IMCs and IMVs MUST work properly in this environment. They MUST NOT assume that only one party will receive and/or respond to a message.

IF-IMC allows an IMC to send and receive messages using this messaging system. Note that this system should not be used to send large amounts of data. The messages will often be sent through PPP or similar protocols that do not include congestion control and are not well suited to bulk data transfer. If an IMC needs to download a patch (for instance), the IMV should indicate this in the remediation instructions. The IMC will process those instructions after network access (perhaps isolated) has been established and can then download the patch via any appropriate protocol.

All messages sent with `TNC_TNCC_SendMessage` and received with `TNC_IMC_ReceiveMessage` are between the IMC and IMV. The IMC communicates with the TNCC by calling functions (standard and vendor-specific) in the IF-IMC, not by sending messages. The TNCC should not interfere with communications between the IMC and IMVs by consuming or blocking IMC-IMV messages.

2.7.5 Batches

IMC-IMV messages will frequently be carried over protocols (like EAP) that require participants to take turns in sending ("half duplex"). To operate well over such protocols, the TNCC sends a batch of messages and the TNCS responds with some messages.

To simplify the development of IMCs and IMVs, IF-IMC always groups IMC-IMV messages into batches. IMCs always send the first batch of messages. IMVs can then respond with a batch of messages, IMCs can respond to those, etc. If the underlying protocol is not half duplex, the TNCC and TNCS still must send IMC-IMV messages in batches and take turns in delivering those messages.

An IMC can only send a message in three circumstances: during the initial batch (when `TNC_IMC_BeginHandshake` is called), in response to a message received by the IMC in a later batch (when `TNC_IMC_ReceiveMessage` is called), and at the end of a batch (when `TNC_IMC_BatchEnding` is called). At any of these times, the IMC MAY send one or more messages by calling `TNC_TNCC_SendMessage` once for each message to be sent and then returning from TNC_IMC_BeginHandshake, TNC_IMC_ReceiveMessage, or TNC_IMC_BatchEnding. Note that if the IMC does not call TNC_TNCC_SendMessage before returning from TNC_IMC_BeginHandshake, TNC_IMC_ReceiveMessage, or TNC_IMC_BatchEnding, this indicates that it does not want to send any messages at this time. IMVs use a similar mechanism except that they can only send messages in response to messages received or at the end of a batch.

If no IMCs want to send a message in a particular batch, the TNCC and TNCS will proceed to complete the handshake. Similarly, if no IMVs want to send a message in a particular batch, the TNCC and TNCS will proceed to complete the handshake. Therefore, an IMC that is not engaged in a dialog with an IMV may well find that the handshake has ended.

When an Integrity Check Handshake is beginning and the TNCC wants to solicit messages from IMCs for the first batch, it calls TNC_IMC_BeginHandshake for each IMC. This indicates to the IMCs that an Integrity Check Handshake is beginning and they should send any IMC-IMV messages. IMCs send those messages by calling the TNC_TNCC_SendMessage function before returning from TNC_IMC_BeginHandshake. Once all IMCs have finished sending their messages for a batch, the TNCC will send those messages to the TNCS and await its response. When this response is received, the TNCC will deliver to IMCs any messages sent by IMVs and start accepting messages from IMCs.

To deliver IMV messages to IMCs, the TNCC calls TNC_IMC_ReceiveMessage. The IMC may process the message immediately or queue it for later processing. However, if the IMC wants to send a message in response, it must do so by calling the TNC_TNCC_SendMessage function before returning from TNC_IMC_ReceiveMessage. Once all IMCs have finished sending their messages for a batch, the TNCC will send those messages to the TNCS and await its response. When this response is received, the TNCC will deliver to IMCs any messages sent by IMVs and start accepting messages from IMCs.

As with all IMC functions, the IMC SHOULD NOT wait a long time before returning from TNC_IMC_BeginHandshake, TNC_IMC_ReceiveMessage, or TNC_IMC_BatchEnding. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise). IMCs that need to perform a lengthy process may want to simply send a status message, indicating that they are working. The IMVs can respond in the next batch with a status query and thus the handshake can be kept going.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCC will return TNC_RESULT_ILLEGAL_OPERATION from TNC_TNCC_SendMessage.

3 IF-IMC Abstract API

The IF-IMC Abstract API defines a small number of standard functions that an IMC can implement. The TNC Client calls these functions when it needs the IMC to perform an action (such as processing a message from an IMV). The API also defines certain functions that the TNC Client implements (known as “callbacks”). The IMC calls these functions when it needs the TNC Client to perform an action (such as sending a message to an IMV).

3.1 Platform and Language Independence

IF-IMC is a language-independent abstract API. It can be mapped to almost any programming language. This section defines the abstract API, using C syntax (as defined in [3]) for ease of comprehension.

Section 6 provides a C header file that serves as a binding for the C language with the Microsoft Windows DLL platform binding. Bindings for other programming languages may be defined in the future. However, many languages can use or implement libraries with C bindings. Implementers SHOULD use the C language binding when possible for maximum compatibility with other IMCs and TNC Clients on their platform. This specification does not provide a standard way to mix an IMC written in one language with a TNCC written in another language, beyond the support that may be provided by platform-specific bindings.

IF-IMC is also a platform-independent API. It is designed to support almost any platform. Platform-specific bindings are described in section 4. The IF-IMC API definition sometimes uses language like “unsigned integer of at least 32 bits.” To see the exact definition of this for a particular platform (operating environment and/or language), see the platform-specific bindings.

3.2 Extensibility

To meet the Extensibility requirement defined above, the IF-IMC API includes several extensibility mechanisms: an API version number, dynamic function binding, and vendor IDs.

3.2.1 API Version

This document defines version 1 of the TNC IF-IMC API. Future versions may be incompatible due to removing, adding, or changing functions, types, and constants. However, the `TNC_IMC_Initialize` function and its associated types and constants will not change so that version incompatibilities can be detected. A TNCC or IMC can even support multiple versions of the IF-IMC API for maximum compatibility. See section 3.7.1 for details.

3.2.2 Dynamic Function Binding

Platforms that support IF-IMC SHOULD support dynamic function binding. This feature allows a TNCC or IMC to define functions that go beyond those included in this API and allows the other party to determine whether those functions are defined, call them if so, and handle their absence gracefully. Dynamic function binding is needed to support optional and vendor-specific functions and so that a TNCC or IMC can support multiple API versions.

On platforms that don’t define a Dynamic Function Binding mechanism, all optional functions MUST be implemented, vendor-specific functions MUST NOT be implemented or used except by private convention, and provisions must be made to insure that TNCCs and IMCs that support different version numbers interact safely.

3.2.3 Vendor IDs

The IF-IMC API supports several forms of vendor extensions. IMC or TNCC vendors can define vendor-specific functions and make them available to the other party. IMC or TNCC vendors can define vendor-specific result codes. And IMC vendors can define vendor-specific message types (for the messages sent between IMCs and IMVs).

In each of these cases, SMI Private Enterprise Numbers are used to provide a separate identifier space for each vendor. IANA provides a registry for SMI Private Enterprise Numbers at http://www.iana.org/assignments/enterprise-numbers. Any organization (including non-profit organizations, governmental bodies, etc.) can obtain one of these numbers at no charge and thousands of organizations have done so. Within this document, SMI Private Enterprise Numbers are known as "vendor IDs". Vendor ID zero (0) is reserved for identifiers defined by the TNC. Vendor ID 16777215 (0xffffff) is reserved for use as a wildcard. For details of how vendor IDs are used to support vendor-specific functions, result codes, and message types, see sections 3.2.4, 3.4.2.10, and 3.4.2.5.

3.2.4 Vendor-Specific Functions

The IMC and TNC client MAY extend the IF-IMC API by defining vendor-specific functions that go beyond those described here. An IMC or TNC Client MUST work properly if a vendor-specific function is not implemented by the other party and MUST ignore vendor-specific functions that it does not understand. To determine whether a vendor-specific function has been implemented, use the dynamic function binding mechanism defined in the platform binding.

Vendor-specific functions MUST have a name that begins with "`TNC_XXX_`" where `XXX` is replaced by the vendor ID of the organization that defined the extension. The vendor ID is converted to ASCII numbers or the equivalent, using a decimal representation whose initial digit MUST NOT be zero (0). For instance, the organization owning the vendor ID 1 could define a vendor-specific function named "`TNC_1_ProcessMapping`". Avoid defining names longer than 31 characters since some platforms do not support such long names well. If a vendor-specific function is designed to be implemented by only one TNC component, then it is helpful to put the name of this component in the function name after the vendor ID. For instance, a function named "`TNC_1_IMC_Reinstall`" is clearly intended to be implemented by IMCs.

3.3 Threading and Reentrancy

Threading is addressed in the platform-specific bindings in section 4.

The TNCC MUST be reentrant (able to receive and process a function call even when one is already underway). IMC DLLs are not required to be reentrant. Therefore, the TNC Client MUST NOT call an IMC DLL from a callback function (like `TNC_TNCC_SendMessage`) and MUST NOT call an IMC DLL from one thread if another thread has an active call into that DLL. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC DLL MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the DLL. An IMC DLL MAY return `TNC_RESULT_CANT_RESPOND` from any function if it is temporarily unable to respond (perhaps because it can only handle one network connection at once). If at all possible, the IMC DLL should avoid doing this since it may cause the TNCC to proceed without the IMC's messages, resulting in denied network access or even unnecessary remediation.

Note that an IMC DLL may just be a stub that communicates with a separate process that processes and responds to IMV messages. This will be fairly common, especially when there is already a background process running (to do real-time virus checking, for instance). Alternatively, the IMC DLL may be very simple, reporting stored values. This will also be very common, especially when integrity checks are fairly static. The checks can run periodically and store their results. The IMC DLL can just read and report these stored results.

3.4 Data Types

This section describes the data types defined and used in the abstract IF-IMC API.

3.4.1 Basic Types

These types are the most basic ones used by the IF-IMC API. They are defined in a platform-dependent and language-dependent manner to meet the requirements described in this section. Consult section 4 to see how these types are defined for a particular platform and language.

Type	Definition
`TNC_UInt32`	Unsigned integer of at least 32 bits
`TNC_BufferReference`	Reference to buffer of octets

3.4.2 Derived Types

These types are defined in terms of the more basic ones defined in section 3.4.1. They are described in the following subsections.

Type	Definition	Usage
`TNC_IMCID`	`TNC_UInt32`	IMC ID
`TNC_ConnectionID`	`TNC_UInt32`	Network Connection ID
`TNC_ConnectionState`	`TNC_UInt32`	Network Connection State
`TNC_RetryReason`	`TNC_UInt32`	Handshake retry reason
`TNC_MessageType`	`TNC_UInt32`	Message type
`TNC_MessageTypeList`	Platform-specific	Reference to list of `TNC_MessageType`
`TNC_VendorID`	`TNC_UInt32`	Vendor ID
`TNC_Subtype`	`TNC_UInt32`	Message subtype
`TNC_Version`	`TNC_UInt32`	IF-IMC API version number
`TNC_Result`	`TNC_UInt32`	Result code

3.4.2.1 IMC ID

When a TNC Client loads an IMC, it assigns it an IMC ID (represented by the `TNC_IMCID` type). This allows the IMC to identify itself when calling TNCC functions. The IMC ID is a `TNC_UInt32` chosen by the TNCC and passed to the `TNC_IMC_Initialize` function. It is valid until the TNCC calls `TNC_IMC_Terminate` for this IMC.

There is no internal structure to an IMC ID and there are no reserved values. The TNCC can choose any value for the IMC ID and the IMC MUST NOT attach any significance to the value chosen.

3.4.2.2 Network Connection ID

A TNCC may be negotiating with several different TNCSs at once (if the endpoint has several network interfaces that are coming up simultaneously, for instance). Each of these TNCC-TNCS pairs is referred to as a "network connection".

To help the IMC track which messages go with which network connection and perform other connection management tasks, the TNCC chooses a network connection ID (represented by the `TNC_ConnectionID` type) that identifies a particular network connection. This connection ID is local to the TNCC and not shared with the TNCS. It's like a socket descriptor in UNIX. When a network connection is created, the TNCC chooses a network connection ID and then passes the network connection ID to the IMC as a parameter to the `TNC_IMC_NotifyConnectionChange` function with a `newState` of `TNC_CONNECTION_STATE_CREATE`. This informs the IMC that a new network connection has begun. The network connection ID then becomes valid.

The IMC and TNCC use this network connection ID to refer to the network connection when delivering messages and performing other operations relevant to the network connection. This helps ensure that IMC messages are sent to the right TNCS and helps the IMC match up messages from IMVs with any state the IMC may be maintaining from earlier parts of that IMC-IMV conversation (even extending across multiple Integrity Check Handshakes in a single network connection).

The TNCC notifies IMCs of changes in network connection state (handshake success, handshake failure, etc.) by calling the `TNC_IMC_NotifyConnectionChange` function. When a network connection is finished, the TNCC first notifies IMCs of this by calling the `TNC_IMC_NotifyConnectionChange` function with the network connection ID and a `newState` of `TNC_CONNECTION_STATE_DELETE`. The network connection ID then becomes invalid and any information associated with it can be deleted. Once a network connection enters the `TNC_CONNECTION_STATE_DELETE` state, it cannot transition to any other state.

As described in section 2.7.3 above, it is sometimes desirable to retry an Integrity Check Handshake (when remediation is complete, for instance). Some TNCCs will not support this but all IMCs MUST do so. To indicate that a network connection retry is beginning, a TNCC notifies the IMCs by calling the `TNC_IMC_NotifyConnectionChange` function with the network connection ID and a `newState` of `TNC_CONNECTION_STATE_HANDSHAKE`. This means that an Integrity Check Handshake will soon begin.

An IMC can ask the TNCC to retry an Integrity Check Handshake by calling the `TNC_TNCC_RequestConnectionRetry` function. For details on this, see the description of that function.

There is no internal structure to a network connection ID. There is one reserved value: `TNC_CONNECTIONID_ANY` (`0xFFFFFFFF`). The TNCC can choose any other value for a network connection ID that does not conflict with another valid network connection ID for the same TNCC-IMC pair. It can even choose a network connection ID that was used by a previous network connection that has now been deleted and is invalid. The IMC MUST NOT attach any significance to the value chosen.

3.4.2.3 Network Connection State

The TNCC uses the `TNC_IMC_NotifyConnectionChange` function to notify IMCs of changes in network connection state. The network connection state is represented as a `TNC_UInt32`. The TNCC MUST pass one of the values listed in section 3.5.3. The TNCC MUST NOT use any other network connection state value with this version of the IF-IMC API.

3.4.2.4 Handshake Retry Reason

The IMC can ask the TNCC to retry an Integrity Check Handshake by calling the `TNC_TNCC_RequestHandshakeRetry` function. One of the parameters to that function is a `TNC_RetryReason`. This type is represented as a `TNC_UInt32`. The IMC MUST pass one of the values listed in section 3.5.5. The IMC MUST NOT use any other handshake retry reason value with this version of the IF-IMC API.

3.4.2.5 Message Type

As described in section 2.7.4, the TNC architecture routes messages between IMCs and IMVs based on their message type. Each message has a message type that uniquely identifies the format and semantics of the message. A message type is a 32-bit number. In the IF-IMC API, this number is represented as a `TNC_UInt32`.

To ensure the uniqueness of message types while providing for vendor extensions, vendor-specific message types are formed by placing a vendor-chosen message subtype in the least significant 8 bits of the message type and the vendor's vendor ID in the most significant 24 bits of the message type. Message types standardized by the TCG will have the reserved value zero (0) in the most significant 24 bits.

The vendor ID `TNC_VENDORID_ANY` (`0xffffff`) and the subtype `TNC_SUBTYPE_ANY` (`0xff`) are reserved as wild cards as described in section 3.8.1. An IMC MUST NOT send messages whose message type includes one of these reserved values.

TNC Clients and TNC Servers MUST properly deliver messages with any message type (as described in section 2.7.4).

3.4.2.6 Message Type List

The `TNC_MessageTypeList` type represents a list of message types. Its exact representation is platform-specific, but will typically be a pointer or reference to an array of `TNC_MessageType`s.

3.4.2.7 Vendor ID

The `TNC_VendorID` type represents a 24-bit vendor ID as described in section 3.2.3. It is represented as a `TNC_UInt32`, but only values from 0 to 16777215 (0xffffff) are valid. This type is used when forming and parsing message types. For a full description of vendor IDs, see section 3.2.3.

The message type `TNC_VENDORID_ANY` (`0xffffff`) is reserved as a wild card as described in section 3.8.1. IMCs may request messages with this vendor ID to indicate that they want to receive messages whose message type includes any vendor ID. However, an IMC MUST NOT send messages whose message type includes this reserved value and a TNCC MUST NOT deliver such messages.

3.4.2.8 Message Subtype

The `TNC_MessageSubtype` type represents an 8-bit message subtype. It is represented as a `TNC_UInt32`, but only values from 0 to 255 are legal. This type is used when forming and parsing message types.

The message subtype `TNC_SUBTYPE_ANY` (`0xff`) is reserved as a wild card as described in section 3.8.1. IMCs may request messages with this message subtype to indicate that they want to receive messages whose message subtype has any value. However, an IMC MUST NOT send messages whose message subtype includes this reserved value and a TNCC MUST NOT deliver such messages.

3.4.2.9 Version

The `TNC_Version` type represents an API version number. See sections 3.2.1 and 3.7.1 for details on how this is used.

3.4.2.10 Result Code

Each function in the IF-IMC API returns a result code of type `TNC_Result` to indicate success or the reason for failure. As noted above, a result code is represented as a `TNC_UInt32`, an unsigned integer of at least 32 bits in length. To form a vendor-specific result code, place a vendor-chosen subcode in the least significant 8 bits of the integer and the vendor's vendor ID in the next most significant 24 bits of the result code (the most significant 24 bits if the integer is 32 bits long). All result codes defined in this specification (listed in section 3.5.1) have the reserved value zero (0) in the most significant 24 bits.

IMCs and TNCCs MUST be prepared for any function to return any result code. Vendor-specific result codes are always permissible and new standard result codes may be defined without changing the version number of the IF-IMC API. Any unknown non-zero result code SHOULD be treated as equivalent to `TNC_RESULT_OTHER`.

3.5 Defined Constants

This section describes the constants defined in the abstract IF-IMC API.

3.5.1 Result Code Values

Each function in the IF-IMC API returns a result code of type `TNC_Result` to indicate success or reason for failure. Here is the set of standard result codes defined by this specification. Vendor-specific result codes are always permissible and new standard result codes may be defined without changing the version number of the IF-IMC API. IMCs and TNCCs MUST be prepared for any function to return any result code. Any unknown non-zero result code SHOULD be treated as equivalent to `TNC_RESULT_OTHER`. IMCs or TNCCs MAY communicate errors to users, log them, ignore them, or handle them in another way that is compliant with this specification.

If an IMC function returns `TNC_RESULT_FATAL`, then the IMC has encountered a permanent error. The TNCC SHOULD call `TNC_IMC_Terminate` as soon as possible. The TNCC MAY then try to reinitialize the IMC with `TNC_IMC_Initialize` or try other measures such as unloading and reloading the IMC and then reinitializing it.

If a TNCC function returns `TNC_RESULT_FATAL`, then the TNCC has encountered a permanent error.

Result Code	Definition
TNC_RESULT_SUCCESS	Function completed successfully
TNC_RESULT_NOT_INITIALIZED	TNC_IMC_Initialize has not been called
TNC_RESULT_ALREADY_INITIALIZED	TNC_IMC_Initialize was called twice without a call to TNC_IMC_Terminate
TNC_RESULT_NO_COMMON_VERSION	No common IF-IMC API version between IMC and TNC Client
TNC_RESULT_CANT_RETRY	TNCC cannot attempt handshake retry
TNC_RESULT_WONT_RETRY	TNCC refuses to attempt handshake retry
TNC_RESULT_INVALID_PARAMETER	Function parameter is not valid
TNC_RESULT_CANT_RESPOND	IMC cannot respond now
TNC_RESULT_ILLEGAL_OPERATION	Illegal operation attempted
TNC_RESULT_OTHER	Unspecified error
TNC_RESULT_FATAL	Unspecified fatal error

3.5.2 Version Numbers

As noted in section 3.2.1, this specification defines version 1 of the TNC IF-IMC API. Future versions of this specification will define other version numbers. See section 3.7.1 for a description of how version numbers are handled.

Version Number	Definition
TNC_IFIMC_VERSION_1	The version of IF-IMC API defined here

3.5.3 Network Connection ID Values

The reserved value `TNC_CONNECTIONID_ANY` MUST NOT be used as a normal network connection ID. Instead, it may be passed to `TNC_TNCC_RequestHandshakeRetry` to indicate that handshake retry is requested for all current network connections.

Network Connection ID Value	Definition

TNC_CONNECTIONID_ANY	All current network connections

3.5.4 Network Connection State Values

This is the complete set of permissible values for the `TNC_Connection_State` type in this version of the IF-IMC API.

Network Connection State Value	Definition
TNC_CONNECTION_STATE_CREATE	Network connection created
TNC_CONNECTION_STATE_HANDSHAKE	Handshake about to start
TNC_CONNECTION_STATE_ACCESS_ALLOWED	Handshake completed. TNCS recommended that requested access be allowed.
TNC_CONNECTION_STATE_ACCESS_ISOLATED	Handshake completed. TNCS recommended that isolated access be allowed.
TNC_CONNECTION_STATE_ACCESS_NONE	Handshake completed. TNCS recommended that no network access be allowed.
TNC_CONNECTION_STATE_DELETE	About to delete network connection ID. Remove all associated state.

3.5.5 Handshake Retry Reason Values

This is the complete set of permissible values for the `TNC_Retry_Reason` type in this version of the IF-IMC API.

Handshake Retry Reason Value	Definition
TNC_RETRY_REASON_IMC_REMEDIATION_COMPLETE	IMC has completed remediation
TNC_RETRY_REASON_IMC_SERIOUS_EVENT	IMC has detected a serious event and recommends handshake retry even if network connectivity must be interrupted
TNC_RETRY_REASON_IMC_INFORMATIONAL_EVENT	IMC has detected an event that it would like to communicate to the IMV. It requests handshake retry but not if network connectivity must be interrupted
TNC_RETRY_REASON_IMC_PERIODIC	IMC wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted

3.5.6 Vendor ID Values

These are reserved vendor ID values. Other vendor IDs between 1 and 16777214 (0xfffffe) may be used as described in section 3.4.2.7. Note that vendor IDs are assigned by IANA as described in section 3.2.3.

Vendor ID Value	Value	Definition
TNC_VENDORID_TCG	0	Reserved for TCG-defined values
TNC_VENDORID_ANY	0xffffff	Wild card matching any vendor ID

3.5.7 Message Subtype Values

This is a reserved message subtype value. Other message subtypes between 0 and 254 may be used as described in section 3.4.2.8. Note that message subtypes are assigned by vendors as described in section 3.4.2.5.

Message Subtype Value	Value	Definition
TNC_SUBTYPE_ANY	0xff	Wild card matching any message subtype

3.6 Mandatory and Optional Functions

Some of the functions in the IF-IMC API are marked as mandatory below. Mandatory functions MUST be implemented. The rest of the functions in the IF-IMC API are marked as optional and need not be implemented. An IMC or TNC Client MUST work properly if one or more optional functions are not implemented by the other party. To determine whether an optional function has been implemented, use the Dynamic Function Binding mechanism defined in most platform bindings. On platforms that don't define a Dynamic Function Binding mechanism, all optional functions MUST be implemented.

3.7 IMC Functions

These functions are implemented by the IMC and called by the TNC Client.

3.7.1 TNC_IMC_Initialize (MANDATORY)

```
TNC_Result TNC_IMC_Initialize(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_Version minVersion,
      /*in*/  TNC_Version maxVersion,
      /*out*/ TNC_Version *pOutActualVersion);
```

Description:

The TNC Client calls this function to initialize the IMC and agree on the API version number to be used. It also supplies the IMC ID, an IMC identifier that the IMC must use when calling TNC Client callback functions. All IMCs MUST implement this function.

The TNC Client MUST NOT call any other IF-IMC API functions for an IMC until it has successfully completed a call to `TNC_IMC_Initialize`(). Once a call to this function has completed successfully, this function MUST NOT be called again for a particular IMC-TNCC pair until a call to `TNC_IMC_Terminate` has completed successfully.

The TNC Client MUST set `minVersion` to the minimum IF-IMC API version number that it supports and MUST set `maxVersion` to the maximum API version number that it supports. The TNC Client also MUST set `pOutActualVersion` so that the IMC can use it as an output parameter to provide the actual API version number to be used. With the C binding, this would involve setting pOutActualVersion to point to a suitable storage location.

The IMC MUST check these to determine whether there is an API version number that it supports in this range. If not, the IMC MUST return TNC_RESULT_NO_COMMON_VERSION. Otherwise, the IMC SHOULD select a mutually supported version number, store that version number at pOutActualVersion, and initialize the IMC. If the initialization completes successfully, the IMC SHOULD return TNC_RESULT_SUCCESS. Otherwise, it SHOULD return another result code.

If an IMC determines that pOutActualVersion is not set properly to allow the IMC to use it as an output parameter, the IMC SHOULD return TNC_RESULT_INVALID_PARAMETER. With the C binding, this might involve checking for a NULL pointer. IMCs are not required to make this check and there is no guarantee that IMCs will be able to perform it adequately (since it is often impossible or very hard to detect invalid pointers).

Input Parameter	**Description**
imcID	IMC ID assigned by TNCC
minVersion	Minimum API version supported by TNCC
maxVersion	Maximum API version supported by TNCC

Output Parameter	**Description**
pOutActualVersion	Mutually supported API version number

Result Code	**Condition**
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NO_COMMON_VERSION	No common API version supported by IMC and TNC Client
TNC_RESULT_ALREADY_INITIALIZED	TNC_IMC_Initialize has already been called and TNC_IMC_Terminate has not
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
Other result codes	Other non-fatal error

3.7.2 TNC_IMC_NotifyConnectionChange (OPTIONAL)

```
TNC_Result TNC_IMC_NotifyConnectionChange(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID
      /*in*/  TNC_ConnectionState newState);
```

Description:

The TNC Client calls this function to inform the IMC that the state of the network connection identified by connectionID has changed to newState. Section 3.5.3 lists all the possible values of newState for this version of the IF-IMC API. The TNCC MUST NOT use any other values with this version of IF-IMC.

IMCs that want to track the state of network connections or maintain per-connection data structures SHOULD implement this function. Other IMCs MAY implement it.

If the state is `TNC_CONNECTION_STATE_CREATE`, the IMC SHOULD note the creation of a new network connection.

If the state is `TNC_CONNECTION_STATE_ACCESS_ALLOWED` or `TNC_CONNECTION_STATE_ACCESS_ISOLATED`, the IMC SHOULD proceed with any remediation instructions received during the Integrity Check Handshake. However, the IMC SHOULD be prepared for delays in network access or even complete denial of network access, even in these cases. Network access will often be delayed for a few seconds while an IP address is acquired. And network access may be denied if the NAA overrides the TNCS Action Recommendation reflected in the newState value.

If the state is `TNC_CONNECTION_STATE_ACCESS_NONE`, the IMC MAY discard any remediation instructions received during the Integrity Check Handshake or it MAY follow them if possible.

If the state is `TNC_CONNECTION_STATE_HANDSHAKE`, an Integrity Check Handshake is about to begin.

If the state is `TNC_CONNECTION_STATE_DELETE`, the IMC SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection ID to the TNC Client after this function returns (unless the TNCC later creates another network connection with the same network connection ID).

The `imcID` parameter MUST contain the IMC ID value provided to `TNC_IMC_Initialize`. The `connectionID` parameter MUST contain a valid network connection ID. IMCs MAY check these values to make sure they are valid and return an error if not, but IMCs are not required to make these checks. The `newState` parameter MUST contain one of the values listed in section 3.5.3.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID whose state is changing
`newState`	New network connection state

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.7.3 TNC_IMC_BeginHandshake (MANDATORY)

```
TNC_Result TNC_IMC_BeginHandshake(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Client calls this function to indicate that an Integrity Check Handshake is beginning and solicit messages from IMCs for the first batch. The IMC SHOULD send any IMC-IMV messages it wants to send as soon as possible after this function is called and then return from this function to indicate that it is finished sending messages for this batch.

As with all IMC functions, the IMC SHOULD NOT wait a long time before returning from `TNC_IMC_BeginHandshake`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

All IMCs MUST implement this function.

The `imcID` parameter MUST contain the IMC ID value provided to `TNC_IMC_Initialize`. The `connectionID` parameter MUST contain a valid network connection ID. IMCs MAY check these values to make sure they are valid and return an error if not, but IMCs are not required to make these checks.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID on which message was received

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.7.4 TNC_IMC_ReceiveMessage (OPTIONAL)

```
TNC_Result TNC_IMC_ReceiveMessage(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

The TNC Client calls this function to deliver a message to the IMC. The message is contained in the buffer referenced by message and contains the number of octets (bytes) indicated by messageLength. The type of the message is indicated by messageType. The message MUST be from an IMV (or a TNCS or other party acting as an IMV).

The IMC SHOULD send any IMC-IMV messages it wants to send as soon as possible after this function is called and then return from this function to indicate that it is finished sending messages in response to this message.

As with all IMC functions, the IMC SHOULD NOT wait a long time before returning from `TNC_IMC_ReceiveMessage`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

The IMC should implement this function if it wants to receive messages. Simple IMCs that only send messages need not implement this function. The IMC MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMC_ReceiveMessage` has returned. If the IMC wants to retain the message, it should copy it before returning from `TNC_IMC_ReceiveMessage`.

The `imcID` parameter MUST contain the IMC ID value provided to `TNC_IMC_Initialize`. The `connectionID` parameter MUST contain a valid network connection ID. The `message` parameter MUST contain a reference to a buffer containing the message being delivered to the IMC. The `messageLength` parameter MUST contain the number of octets in the message. If the value of the `messageLength` parameter is zero (0), the `message` parameter may be `NULL` with platform bindings that have such a value. The `messageType` parameter MUST contain the type of the message. It MUST match one of the `TNC_MessageType` values previously supplied by the IMC to the TNCC in the IMC's most recent call to `TNC_TNCC_ReportMessageTypes`. IMCs MAY check these parameters to make sure they are valid and return an error if not, but IMCs are not required to make these checks.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID on which message was received
`message`	Reference to buffer containing message
`messageLength`	Number of octets in message
`messageType`	Message type of message

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.7.5 TNC_IMC_BatchEnding (OPTIONAL)

```
TNC_Result TNC_IMC_BatchEnding(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Client calls this function to notify IMCs that all IMV messages received in a batch have been delivered and this is the IMC's last chance to send a message in the batch of IMC messages currently being collected. An IMC MAY implement this function if it wants to perform some actions after all the IMV messages received during a batch have been delivered (using `TNC_IMC_ReceiveMessage`). This is especially useful for IMCs that have included a wildcard in the list of message types reported using `TNC_TNCC_ReportMessageTypes`.

An IMC MAY call `TNC_TNCC_SendMessage` from this function. As with all IMC functions, the IMC SHOULD NOT wait a long time before returning from `TNC_IMC_BatchEnding`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

The `imcID` parameter MUST contain the IMC ID value provided to `TNC_IMC_Initialize`. The `connectionID` parameter MUST contain a valid network connection ID. IMCs MAY check these values to make sure they are valid and return an error if not, but IMCs are not required to make these checks.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID for which a batch is ending

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.7.6 TNC_IMC_Terminate (OPTIONAL)

```
TNC_Result TNC_IMC_Terminate(
      /*in*/  TNC_IMCID imcID);
```

Description:

The TNC Client calls this function to close down the IMC when all work is complete or the IMC reports `TNC_RESULT_FATAL`. Once a call to `TNC_IMC_Terminate` is made, the TNC Client MUST NOT call the IMC except to call `TNC_IMC_Initialize` (which may not succeed if the IMC cannot reinitialize itself). Even if the IMC returns an error from this function, the TNC Client MAY continue with its unload or shutdown procedure.

The `imcID` parameter MUST contain the IMC ID value provided to `TNC_IMC_Initialize`. IMCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC

Result Code	**Condition**

`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.8 TNC Client Functions

These functions are implemented by the TNC Client and called by the IMC.

3.8.1 TNC_TNCC_ReportMessageTypes (MANDATORY)

```
TNC_Result TNC_TNCC_ReportMessageTypes(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_MessageTypeList supportedTypes,
      /*in*/  TNC_UInt32 typeCount);
```

Description:

An IMC calls this function to inform a TNCC about the set of message types the IMC is able to receive. Often, the IMC will call this function from `TNC_IMC_Initialize`. With the Windows DLL binding or UNIX/Linux Dynamic Linkage binding, `TNC_TNCC_ReportMessageTypes` will typically be called from `TNC_IMC_ProvideBindFunction` since an IMC cannot call the TNCC with those platform bindings until `TNC_IMC_ProvideBindFunction` is called. A list of message types is contained in the `supportedTypes` parameter. The number of types in the list is contained in the `typeCount` parameter. If the value of the `typeCount` parameter is zero (0), the `supportedTypes` parameter may be `NULL` with platform bindings that have such a value. The imcID MUST contain the value provided to `TNC_IMC_Initialize`. TNCCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

All TNC Clients MUST implement this function. The TNC Client MUST NOT ever modify the list of message types and MUST NOT access this list after `TNC_TNCC_ReportMessageTypes` has returned. Generally, the TNC Client will copy the contents of this list before returning from this function. TNC Clients MUST support any message type.

Note that although all TNC Clients must implement this function, some IMCs may never call it if they don't support receiving any message types. This is acceptable. In such a case, the TNC Client MUST NOT deliver any messages to the IMC.

If an IMC requests a message type whose vendor ID is `TNC_VENDORID_ANY` and whose subtype is `TNC_SUBTYPE_ANY` it will receive all messages with any message type. This message type is `0xffffffff`. If an IMC requests a message type whose vendor ID is NOT `TNC_VENDORID_ANY` and whose subtype is `TNC_SUBTYPE_ANY`, it will receive all messages with the specified vendor ID and any subtype. If an IMC calls `TNC_TNCC_ReportMessageTypes` more than once, the message type list supplied in the latest call supplants the message type lists supplied in earlier calls.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC

supportedTypes	Reference to list of message types supported by IMC
typeCount	Number of message types supported by IMC

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.2 TNC_TNCC_SendMessage (MANDATORY)

```
TNC_Result TNC_TNCC_SendMessage(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

An IMC calls this function to give a message to the TNCC for delivery. The message is contained in the buffer referenced by the message parameter and contains the number of octets (bytes) indicated by the messageLength parameter. If the value of the messageLength parameter is zero (0), the message parameter may be NULL with platform bindings that have such a value. The type of the message is indicated by the messageType parameter. The imcID parameter MUST contain the value provided to TNC_IMC_Initialize and the connectionID parameter MUST contain a valid network connection ID. TNCCs MAY check these values to make sure they are valid and return an error if not, but TNCCs are not required to make these checks.

All TNC Clients MUST implement this function. The TNC Client MUST NOT ever modify the buffer contents and MUST NOT access the buffer after TNC_TNCC_SendMessage has returned. The TNC Client will typically copy the message out of the buffer, queue it up for delivery, and return from this function.

The IMC MUST NOT call this function unless it has received a call to TNC_IMC_BeginHandshake, TNC_IMC_ReceiveMessage, or TNC_IMC_BatchEnding for this connection and the IMC has not yet returned from that function. If the IMC violates this prohibition, the TNCC SHOULD return TNC_RESULT_ILLEGAL_OPERATION. If an IMC really wants to communicate with an IMV at another time, it should call TNC_TNCC_RequestHandshakeRetry.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCC will return TNC_RESULT_ILLEGAL_OPERATION from TNC_TNCC_SendMessage.

The TNC Client MUST support any message type. However, the IMC MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in section 3.8.1. If the IMC violates this prohibition, the TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID on which message should be sent
`message`	Reference to buffer containing message
`messageLength`	Number of octets in message
`messageType`	Message type of message

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_ILLEGAL_OPERATION`	Message send attempted at illegal time
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.3 TNC_TNCC_RequestHandshakeRetry (MANDATORY)

```
TNC_Result TNC_TNCC_RequestHandshakeRetry(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_RetryReason reason);
```

Description:

An IMC calls this function to ask a TNCC to retry an Integrity Check Handshake. The IMC MUST pass its IMC ID as the `imcID` parameter, a network connection ID as the `connectionID` parameter, and one of the handshake retry reasons listed in section 3.5.5 as the `reason` parameter. If the network connection ID is `TNC_CONNECTIONID_ANY`, then the IMC requests an Integrity Check Handshake retry on all current network connections.

TNCCs MAY check the parameters to make sure they are valid and return an error if not, but TNCCs are not required to make these checks. The `reason` parameter explains why the IMC is requesting a handshake retry. The TNCC MAY use this in deciding whether to attempt the handshake retry. As noted in section 2.7.3, TNCCs are not required to honor IMC requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMC MAY call this function at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMC suddenly gets important information but has already finished its dialog with the IMV, for instance. As always, the TNCC is not required to honor the request for handshake retry.

If the TNCC cannot attempt the handshake retry, it SHOULD return the result code `TNC_RESULT_CANT_RETRY`. If the TNCC could attempt to retry the handshake but chooses not to, it SHOULD return the result code `TNC_RESULT_WONT_RETRY`. If the TNCC intends to retry the handshake, it SHOULD return the result code `TNC_RESULT_SUCCESS`. The IMC MAY use this information in displaying diagnostic and progress messages.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID for which handshake retry is requested
`reason`	Reason why handshake retry is requested

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	TNCC intends to retry the handshake
`TNC_RESULT_CANT_RETRY`	TNCC cannot attempt the handshake retry
`TNC_RESULT_WONT_RETRY`	TNCC won't attempt the handshake retry
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

4 Platform Bindings

As noted above, IF-IMC is a platform-independent API. It is designed to support almost any platform. In order to ensure compatibility within a single platform, this section defines how IF-IMC SHOULD be implemented on specific platforms. Additional platform bindings will be defined later.

4.1 Microsoft Windows DLL Platform Binding

Microsoft Windows is a popular platform with many variations. This binding does not support 16-bit Windows (Windows 3.X and Windows for Workgroups). It does support Windows XP, Windows CE, Windows NT, Windows 98, Windows Me, Windows 95, 64-bit Windows, and all other currently known versions of Windows.

Implementations on one of these platforms SHOULD use this binding when possible for maximum compatibility with other IMCs and TNC Clients on the platform. However, some languages (such as Java) cannot easily implement or load DLLs. Implementations in such a language may choose not to use this binding or may write custom code to support this binding.

4.1.1 Finding, Loading, and Unloading IMCs

One factor in Windows' success has been the ease with which software can be installed and configured. However, this can also lead to security problems if untrusted users install unsafe software. We retain this ease of configuration while providing some protection against unsafe software. Use of the Trusted Platform Module will increase the protection against unsafe software configuration.

With the Microsoft Windows DLL platform binding, each IMC is implemented as a DLL. When the DLL is installed, it is stored in a directory that can only be accessed by privileged users. The full path of the DLL is stored in a well-known registry key that can only be changed by privileged users. The TNC Client gets the value of this key and loads the IMCs using the `LoadLibrary` system call. Then it uses the `GetProcAddress` function call to access the IMC's functions, as described in section 4.1.2. The TNCC MUST always call the `TNC_IMC_Initialize` function first. When it is done using an IMC, the TNC Client calls `TNC_IMC_Terminate` and then unloads the IMC DLL using the `FreeLibrary` system call. The TNCC SHOULD listen for changes to the well-known registry key so that it can load and unload IMCs dynamically. However, the TNCC SHOULD delay before making changes based on registry key changes since it's common for these changes to come in batches within a few seconds during an install process. And the TNCC MAY not listen for such changes at all.

On some versions of Windows (including at least Windows 95, Windows 98, and Windows ME), there is no such thing as a privileged user. This means that any code executed with the privileges of any user can modify the registry key that lists the installed IMCs. However, this problem is not unique to the IF-IMC API. It's commonly known that running malicious code on such an operating system may result in complete compromise of the machine. The chances of such an attack can be reduced through best practices like well-patched software, strong host intrusion prevention, and antivirus protection. The TNC architecture supports and encourages such measures. IF-IMC helps ensure that these measures are in place. IF-PTS allows the TNCS to reliably and securely detect compromised machines through use of the TPM. But upgrading to a more secure version of Windows is also recommended.

As described in the Security Considerations section, loading an IMC DLL into the TNCC's address space can compromise the TNCC and other IMCs if the IMC DLL is later found to be untrustworthy. Also, an unstable IMC can crash the whole TNCC. One way to address this problem is to have the TNCC launch a new "child" process for each IMC, have the child process load the IMC DLL, and then have the TNCC communicate with the child processes carefully. If an IMC DLL crashes or is untrustworthy, the damage it can do is limited. The TNCC may use this approach but is not required to do so.

4.1.2 Dynamic Function Binding

The Microsoft Windows DLL platform binding does support dynamic function binding. To determine whether an IMC function is defined, a TNC Client will pass the function name to `GetProcAddress`. If the result is `NULL`, the function is not defined. Otherwise, the function is defined and the TNCC can call it using the function pointer returned. This is common practice on Windows.

A similar mechanism is used to allow an IMC to determine whether a TNCC function is defined. In fact, this mechanism is the only way that the IMC can call a TNCC function with this platform binding. A platform-specific mandatory IMC function named `TNC_IMC_ProvideBindFunction` is defined below. For instructions on how this function is used, see its description.

IMC and TNCC functions can be implemented in and called from many languages. With C++, extern “C” should be used to ensure that C linkage conventions are used for IMC and TNCC functions exposed through this API.

4.1.3 Threading

IMC DLLs are not required to be thread-safe. Therefore, the TNC Client MUST NOT call an IMC DLL from one thread when another TNC Client thread is in the middle of a call to the same IMC DLL. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC DLL MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the DLL.

The IMC DLL MAY create threads. The TNC Client MUST be thread-safe. This allows the IMC DLL to do work in background threads and call the TNC Client when it wants to request an Integrity Check Handshake retry (for instance).

All IMC DLL functions SHOULD return promptly. Otherwise, the TNC Client may get bogged down waiting for a response from the IMC. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

4.1.4 Platform-Specific Bindings for Basic Types

With the Microsoft Windows DLL platform binding, the basic data types defined in the IF-IMC abstract API are mapped as follows:

```
typedef unsigned long TNC_UInt32;
```

The `TNC_UInt32` type is mapped to a four byte unsigned value.

```
typedef unsigned char *TNC_BufferReference;
```

The `TNC_BufferReference` type is mapped to a pointer. The value `NULL` is allowed for a `TNC_BufferReference` only where explicitly permitted in this specification.

4.1.5 Platform-Specific Bindings for Derived Types

With the Microsoft Windows DLL platform binding, the platform-specific derived data types defined in the IF-IMC abstract API are mapped as follows:

```
typedef TNC_MessageType *TNC_MessageTypeList;
```

The `TNC_MessageTypeList` type is mapped to a pointer. The value `NULL` is allowed for a `TNC_MessageTypeList` only where explicitly permitted in this specification.

4.1.6 Additional Platform-Specific Derived Types

The Microsoft Windows DLL platform binding for the IF-IMC API defines several additional derived data types.

4.1.6.1 Function Pointers

Function pointer types are defined for all the functions contained in the abstract API and platform binding. This makes it easy to cast function pointers returned by `GetProcAddress` or `TNC_TNCC_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMC_InitializePointer)(
    TNC_IMCID imcID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);

typedef TNC_Result (*TNC_IMC_NotifyConnectionChangePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);

typedef TNC_Result (*TNC_IMC_BeginHandshakePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_ReceiveMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMC_BatchEndingPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_TerminatePointer)(
    TNC_IMCID imcID);

typedef TNC_Result (*TNC_TNCC_ReportMessageTypesPointer)(
    TNC_IMCID imcID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCC_SendMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_TNCC_RequestHandshakeRetryPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCC_BindFunctionPointer)(
    TNC_IMCID imcID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMC_ProvideBindFunctionPointer)(
    TNC_IMCID imcID,
    TNC_TNCC_BindFunctionPointer bindFunction);
```

4.1.7 Platform-Specific IMC Functions

The Microsoft Windows DLL platform binding for the IF-IMC API defines one additional function that MUST be implemented by IMCs implementing this platform binding.

4.1.7.1 TNC_IMC_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMC_ProvideBindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_TNCC_BindFunctionPointer bindFunction);
```

Description:

IMCs implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. The TNC Client MUST call the function immediately after calling `TNC_IMC_Initialize` to provide a pointer to the TNCC bind function. The IMC can then use the TNCC bind function to obtain pointers to any other TNCC functions.

The `imcID` parameter MUST contain the value provided to `TNC_IMC_Initialize`. The `bindFunction` parameter MUST contain a pointer to the TNCC bind function. IMCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description
`imcID`	IMC ID assigned by TNCC
`bindFunction`	Pointer to `TNC_TNCC_BindFunction`

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

4.1.8 Platform-Specific TNC Client Functions

The Microsoft Windows DLL platform binding for the IF-IMC API defines one additional function that MUST be implemented by TNC Clients implementing this platform binding.

4.1.8.1 TNC_TNCC_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCC_BindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Clients implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. An IMC can use this function to obtain pointers to other

TNCC functions. To obtain a pointer to a TNCC function, an IMC calls `TNC_TNCC_BindFunction`. The IMC obtains a pointer to `TNC_TNCC_BindFunction` from `TNC_IMC_ProvideBindFunction`.

The IMC MUST set the `imcID` parameter to the IMC ID value provided to `TNC_IMC_Initialize`. TNCCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check. The IMC MUST set the `functionName` parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. `"TNC_TNCC_SendMessage"`). The IMC MUST set the `pOutFunctionPointer` parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCC does not define the requested function, `NULL` MUST be stored at pOutFunctionPointer. Otherwise, a pointer to the requested function MUST be stored at pOutFunctionPointer. In either case, `TNC_RESULT_SUCCESS` SHOULD be returned. Once an IMC obtains a pointer to a particular function, the TNCC MUST always return the same function pointer value to that IMC for that function name. This requirement does not apply across IMC termination and reinitialization.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`functionName`	Name of function whose pointer is requested

Output Parameter	**Description**
`pOutFunctionPointer`	Requested function pointer

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
Other result codes	Other non-fatal error

4.1.9 Well-known Registry Key

As discussed above, a well-known registry key is used by the TNCC to load IMCs. For Windows platforms, this key is defined within the HKEY_LOCAL_MACHINE hive as follows. The TNCC should also load IMCs from the equivalent path in HKEY_CURRENT_USER hive in addition to HKLM in order to support per-user software profiles.

- HKEY_LOCAL_MACHINE

  → Software

  → Trusted Computing Group

  → TNC

  → IMCs

  → [Human readable name of IMC], 0..n

Each IMC key contains an (unordered) set of values, as follows:

- the value *"Path"* is a REG_SZ String which contains the fully qualified path to an IMC DLL to be loaded
- the optional value *"Description"* is a REG_SZ String which contains a vendor-specific human-readable description of the IMC DLL

The name and description are for ease of administration and may be ignored by the TNCC, except for human interface purposes; only the Path data matters. Duplicate paths are OK. Additional values or keys may be present within the keys listed above. TNC Clients and IMCs MUST ignore unrecognized values and keys.

An extension mechanism has been defined so that vendors can place vendor-specific keys or values in the TNC key or any subkey without risking name collisions. The name of such a vendor-specific key or value must begin with the vendor ID (as defined in section 3.2.3) of the vendor who defined this extension. The vendor ID must be immediately followed in the name by an underscore which may be followed by any string.

The manner in which these vendor-specific values are used is up to the vendor that defines such a value. For instance, a TNC Client vendor with vendor ID 2 could specify that any IMC can populate its key at install time with a value named 2_SupportPhone and that vendor's TNC Client can read this value and display it in the TNC Client's status panel next to the IMC name. The only requirement, as stated above, is that TNC Clients and IMCs MUST ignore unrecognized values and keys.

4.2 UNIX/Linux Dynamic Linkage Platform Binding

NOTE: This binding is still preliminary and under review. A UNIX/Linux static linkage binding may be defined in addition to or instead of this binding.

UNIX and Linux operating systems are used for servers, desktops, and even embedded devices. There are hundreds of varieties of UNIX and Linux dating back to the 1970s. One platform binding cannot support them all. However, this binding supports all varieties of Linux that conform to the Linux Standard Base 1.0.0 or later and all varieties of UNIX that conform to UNIX 98 or any version of the Single UNIX Specification. This includes most varieties of UNIX and Linux currently in use.

Implementations on one of these platforms SHOULD use this binding when possible for maximum compatibility with other IMCs and TNC Clients on the platform. However, some languages (such as Java) cannot easily implement or load shared libraries. Implementations in such a language may choose not to use this binding or to write custom code to support this binding.

4.2.1 Finding, Loading, and Unloading IMCs

With the UNIX/Linux Dynamic Linkage platform binding, each IMC is implemented as a dynamically loaded executable file (also known as a shared object or DLL). When the IMC is installed, its executable file should be stored in a directory that can only be accessed by privileged users. Then an entry is created in the `/etc/tnc_config` file that gives the full path of the executable file. See section 4.2.3 for details on the format of this file.

The TNC Client opens the `/etc/tnc_config` file, reads the entries in the file, and determines which of them should be loaded (using optional local configuration). For each IMC to be loaded, the TNC Client passes the full path of the executable file to the `dlopen` system call. The value passed as the `mode` parameter to the `dlopen` system call is platform-specific and not specified here. The TNC Client uses the `dlsym` function call to access the IMC's functions, as described in section 4.1.2. The TNCC MUST always call the `TNC_IMC_Initialize` function first. When it is done using an IMC, the TNC Client calls `TNC_IMC_Terminate` and then unloads the IMC executable file using the `dlclose` system call.

If the TNCC receives a HUP signal (which may be sent with the `kill` command), the TNCC SHOULD check the `/etc/tnc_config` file for changes and load or unload IMCs as needed to match the latest list.

As described in the Security Considerations section, loading an IMC into the TNCC's address space can compromise the TNCC and other IMCs if the IMC is later found to be untrustworthy. Also, an unstable IMC can crash the whole TNCC. One way to address this problem is to have the TNCC launch a new "child" process for each IMC, have the child process load the IMC, and then have the TNCC communicate with the child processes carefully. If an IMC crashes or is untrustworthy, the damage it can do is limited. The TNCC may use this approach but is not required to do so.

4.2.2 Dynamic Function Binding

The UNIX/Linux Dynamic Linkage platform binding does support dynamic function binding. To determine whether an IMC function is defined, a TNC Client will pass the function name to `dlsym`. If the result is `NULL`, the function is not defined. Otherwise, the function is defined and the TNCC can call it using the function pointer returned. This is common practice on UNIX and Linux.

A similar mechanism is used to allow an IMC to determine whether a TNCC function is defined. In fact, this mechanism is the only way that the IMC can call a TNCC function with this platform binding. A platform-specific mandatory IMC function named `TNC_IMC_ProvideBindFunction` is defined below. For instructions on how this function is used, see its description.

IMC and TNCC functions can be implemented in and called from many languages. With C++, extern "C" should be used to ensure that C linkage conventions are used for IMC and TNCC functions exposed through this API.

4.2.3 Format of `/etc/tnc_config`

The `/etc/tnc_config` file specifies the set of IMCs available for TNCCs to load. TNCCs are not required to load these IMCs. A TNCC may be configured to ignore this file, load a subset of the IMCs listed here, load a superset of those IMCs, or (most common) load the IMCs in the list. This provides a simple, standard way for the list of IMCs to be specified but allows TNCCs to be configured to only load a particular set of trusted IMCs.

The `/etc/tnc_config` file is a UTF-8 file. However, TNCCs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCC encounters a character that is not US-ASCII and the TNCC can not process UTF-8 properly, the TNCC SHOULD indicate an error and not load the file at all. In fact, the TNCC SHOULD respond to any problem with the file by indicating an error and not loading the file at all.

All characters specified here are specified in standard Unicode notation (U+nnnn where nnnn are hexadecimal characters indicating the code points.

The `/etc/tnc_config` file is composed of zero or more lines. Each line ends in U+000A. No other control characters (characters with the Unicode category Cc) are permitted in the file.

A line that begins with U+0023 is a comment. All other characters on the line should be ignored. A line that does not contain any characters should also be ignored.

A line that begins with "IMC " (U+0049, U+004D, U+0043, U+0020) specifies an IMC that may be loaded. The next character MUST be U+0022 (QUOTATION MARK). This MUST be followed by a human-readable IMC name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the IMC name cannot contain a U+0022 (QUOTATION MARK). But it can contain spaces or other characters. After the U+0022 that terminates the human-readable name MUST come a space (U+0020) and then the full path of the IMC executable file (up to but not including the U+000A that terminates the line). The path to the IMC executable file MUST NOT be a partial path.

The `/etc/tnc_config` file must not contain IMCs with the same human-readable name. An IMC that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMC names to make them unique. Identical full paths are permitted but the TNCC MAY ignore entries with identical paths if they will cause problems for it.

An extension mechanism has been defined so that vendors can place vendor-specific data in the `/etc/tnc_config` file without risking conflicts. A line that contains such vendor-specific data must begin with the vendor ID (as defined in section 3.2.3) of the vendor who defined this extension. The vendor ID must be immediately followed in the name by an underscore which may be followed by any string except control characters until the end of line (U+000A).

The internal format of this vendor-specific data and the manner in which it is to be used should be specified by the vendor whose vendor ID is used to define the extension. For instance, a TNCC vendor with vendor ID 2 could specify that any IMC can add a line at install time that begins with 2_SupportPhoneIMC, then the IMC's human-readable name and the IMC vendor's support telephone number. The defining vendor's TNCC (or any other TNCC) can read this phone number and display it in the TNCC's status panel next to the IMC name.

TNCCs and IMCs SHOULD ignore unrecognized vendor-specific data. This recommendation is backwards-compatible with the recommendation in IF-IMC 1.0 for TNCCs and IMCs to ignore lines in `/etc/tnc_config` with unrecognized syntax.

A line that does not match the comment, empty, imc, or vendor productions below SHOULD be ignored by the TNCC and IMCs unless otherwise specified by a future version of this binding. This provides for future extensions to this file format.

Here is a specification of the file format using ABNF as defined in [3].

```
tnc_config = *line
line = (comment / empty / imc / imv / vendor / other) %x0A
comment = %x23 *(%x01-09 / %x0B-22 / %x24-1FFFFF)
empty = ""
imc = %x49.4D.43.20.22 name %x22.20 path
imv = %x49.4D.56.20.22 name %x22.20 path ; Ignored for IF-IMC
name = *(%x01-09 / %x0B-21 / %x23-1FFFFF)
path = *(%x01-09 / %x0B-1FFFFF)
digit = (%x30-39)
vendor = *digit %x5f *(%x01-09 / %x0B-1FFFFF)
other = 1*(%x01-09 / %x0B-1FFFFF) ; But match more specific rules first
```

Here is a sample file specifying one IMC named "AV" located at /usr/bin/myav/av.so.

```
# Simple TNC config file

IMC "AV" /usr/bin/myav/av.so
```

4.2.4 Threading

IMC executable files are not required to be thread-safe. Therefore, the TNC Client MUST NOT call an IMC from one thread when another TNC Client thread is in the middle of a call to the same IMC. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the IMC.

The IMC MAY create threads. The TNC Client MUST be thread-safe. This allows the IMC to do work in background threads and to call the TNC Client when it wants to request an Integrity Check Handshake retry (for instance). Both the IMC and the TNC Client MUST use POSIX threads (pthreads) for threading and synchronization to ensure compatibility.

All IMC functions SHOULD return promptly (preferably, within 100 ms or less). Otherwise, the TNC Client may get bogged down waiting for a response from the IMC.

4.2.5 Platform-Specific Bindings for Basic Types

With the UNIX/Linux Dynamic Linkage platform binding, the basic data types defined in the IF-IMC abstract API are mapped as follows:

```
typedef unsigned long TNC_UInt32;
```

The `TNC_UInt32` type is mapped to a four byte unsigned value.

```
typedef unsigned char *TNC_BufferReference;
```

The `TNC_BufferReference` type is mapped to a pointer. The value `NULL` is allowed for a `TNC_BufferReference` only where explicitly permitted in this specification.

4.2.6 Platform-Specific Bindings for Derived Types

With the UNIX/Linux Dynamic Linkage platform binding, the platform-specific derived data types defined in the IF-IMC abstract API are mapped as follows:

```
typedef TNC_MessageType *TNC_MessageTypeList;
```

The `TNC_MessageTypeList` type is mapped to a pointer. The value `NULL` is allowed for a `TNC_MessageTypeList` only where explicitly permitted in this specification.

4.2.7 Additional Platform-Specific Derived Types

The UNIX/Linux Dynamic Linkage platform binding for the IF-IMC API defines several additional derived data types.

4.2.7.1 Function Pointers

Function pointer types are defined for all the functions contained in the abstract API and platform binding. This makes it easy to cast function pointers returned by `dlsym` or `TNC_TNCC_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMC_InitializePointer)(
    TNC_IMCID imcID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);

typedef TNC_Result (*TNC_IMC_NotifyConnectionChangePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);

typedef TNC_Result (*TNC_IMC_BeginHandshakePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_ReceiveMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMC_BatchEndingPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);
```

```
typedef TNC_Result (*TNC_IMC_TerminatePointer)(
    TNC_IMCID imcID);
typedef TNC_Result (*TNC_TNCC_ReportMessageTypesPointer)(
    TNC_IMCID imcID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCC_SendMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_TNCC_RequestHandshakeRetryPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);
typedef TNC_Result (*TNC_TNCC_BindFunctionPointer)(
    TNC_IMCID imcID,
    char *functionName,
    void **pOutfunctionPointer);
typedef TNC_Result (*TNC_IMC_ProvideBindFunctionPointer)(
    TNC_IMCID imcID,
    TNC_TNCC_BindFunctionPointer bindFunction);
```

4.2.8 Platform-Specific IMC Functions

The UNIX/Linux Dynamic Linkage platform binding for the IF-IMC API defines one additional function that MUST be implemented by IMCs implementing this platform binding.

4.2.8.1 TNC_IMC_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMC_ProvideBindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_TNCC_BindFunctionPointer bindFunction);
```

Description:

IMCs implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. The TNC Client MUST call the function immediately after calling `TNC_IMC_Initialize` to provide a pointer to the TNCC bind function. The IMC can then use the TNCC bind function to obtain pointers to any other TNCC functions.

The `imcID` parameter MUST contain the value provided to `TNC_IMC_Initialize`. The `bindFunction` parameter MUST contain a pointer to the TNCC bind function. IMCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description
`imcID`	IMC ID assigned by TNCC
`bindFunction`	Pointer to `TNC_TNCC_BindFunction`

Result Code	Condition

TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMC_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

4.2.9 Platform-Specific TNC Client Functions

The UNIX/Linux Dynamic Linkage platform binding for the IF-IMC API defines one additional function that MUST be implemented by TNC Clients implementing this platform binding.

4.2.9.1 TNC_TNCC_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCC_BindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Clients implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. An IMC can use this function to obtain pointers to other TNCC functions. To obtain a pointer to a TNCC function, an IMC calls `TNC_TNCC_BindFunction`. The IMC obtains a pointer to `TNC_TNCC_BindFunction` from `TNC_IMC_ProvideBindFunction`.

The IMC MUST set the `imcID` parameter to the IMC ID value provided to `TNC_IMC_Initialize`. TNCCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check. The IMC MUST set the `functionName` parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. "`TNC_TNCC_SendMessage`"). The IMC MUST set the `pOutFunctionPointer` parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCC does not define the requested function, `NULL` MUST be stored at pOutFunctionPointer. Otherwise, a pointer to the requested function MUST be stored at pOutFunctionPointer. In either case, `TNC_RESULT_SUCCESS` SHOULD be returned.

Input Parameter	**Description**
imcID	IMC ID assigned by TNCC
functionName	Name of function whose pointer is requested

Output Parameter	**Description**
pOutFunctionPointer	Requested function pointer

Result Code	**Condition**
TNC_RESULT_SUCCESS	Success
TNC_RESULT_OTHER	Unspecified non-fatal error

`TNC_RESULT_FATAL`	Unspecified fatal error
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
Other result codes	Other non-fatal error

5 Security Considerations

IF-IMC is a critical component since it enables third party security applications to provide the information that can be used by administrators to make network access control decisions. The security of this interface is critical to ensure that the TNC framework is not itself subject to attack and circumvention. Hostile code has many ways to enter a platform and can eliminate, tamper with or circumvent security applications.

This section describes the security threats related to IF-IMC and suggests methods to address these threats. The components involved in IF-IMC are one or more Trusted Network Connect Clients (TNCC) and one or more Integrity Measurement Collectors (IMCs). All the above components reside on the same endpoint or Access Requestor (AR). IF-IMC is the interface between the TNCC and the IMCs.

5.1 Threat analysis

5.1.1 Registration and Discovery based threats

The TNCC discovers which IMCs are installed on a platform via a platform specific binding, for example, on the Windows platform using a windows registry key and on the Linux or Unix platform, a configuration file. On Windows, the registry keys are typically created when the IMCs are installed, requiring the IMC installer to possess sufficient privileges on the platform. Similarly the TNCC must have sufficient privileges to read the relevant keys. Based on the IMCs discovered in the registry, the TNCC loads the code referenced by the registry entries. On Linux and UNIX, analogous privilege requirements apply for accessing the configuration file. Any party with sufficient privileges to modify the relevant registry key or configuration file can mount the following attacks on the registration process:

- It can add an invalid IMC
- It can remove a valid IMC, perhaps replacing it with rogue/modified versions of code

Similar attacks can also be mounted by modifying the code of an IMC or critical data upon which the IMC depends.

The ability to add an invalid IMC can have considerable impact, as detailed in the next section.

5.1.2 Rogue IMC threats

If a rogue IMC is installed and then loaded by a valid TNCC, it may be able to misuse the IF-IMC API in the following ways:

- Overwrite TNCC or IMC memory
- Violate IF-IMC API requirements such as passing illegal or unexpected argument values
- Perform illegal operations so that the TNCC is terminated by the operating system
- Perform improper operations with the TNCC's privileges
- Attack other components (such as the NAR or applications) using the privileges or credentials of the TNCC or other IMCs
- Send invalid messages to IMCs or IMVs, leading to IMC or IMV crashes or compromise, excessive IMC or IMV resource consumption, or unauthorized or malicious remediation
- Monitor IMC-IMV messages and disclose them or use them for attacks on this or other ARs
- Issue a large number of interface API calls to the TNCC (Denial of service of the TNCC)
- Spoof specific IMCs and provide incorrect information to a TNCC about other IMCs
- Spoof TNCC calls to a IMC and provide incorrect connection notification changes.
- Spoof specific IF-IMC APIs and provide incorrect received messages or request incorrect message types for other IMCs
- Spuriously request handshake retries (Denial of service)
- Lock up TNCC threads by not returning from function calls (Denial of service)
- Cause untimely unloading of IMCs
- Use vendor-specific extensions to IF-IMV to perform other attacks

5.1.3 Rogue TNCC threats

If a rogue TNCC loads a valid IMC, it may be able to misuse IF-IMC in the following ways:

- Overwrite IMC memory
- Violate IF-IMC API requirements such as passing illegal or unexpected argument values
- Attack other components (such as the NAR or applications) using the credentials of an IMC
- Send invalid messages to IMCs or IMVs, leading to IMC or IMV crashes or compromise, excessive IMC or IMV resource consumption, or unauthorized or malicious remediation
- Monitor IMC-IMV messages and disclose them or use them for attacks on this or other ARs
- Issue a large number of, or particularly expensive, interface API calls to an IMC, possibly causing denial of service of a critical security application
- Spuriously request or perform handshake retries (Denial of service)
- Use vendor-specific extensions to IF-IMC to perform other attacks

5.1.4 Threats Beyond IF-IMC

IF-IMC is part of the larger TNC architecture. Successful attacks against other parts of the TNC architecture will generally result in negative effects for IMCs, TNCCs, and the system as a whole. See the Security Considerations section of the TNC Architecture document for an analysis of considerations that pertain to other parts of the TNC architecture.

5.2 Suggested remedies

As demonstrated by the attacks listed above, it is critical that only authorized IMCs be loaded by a TNCC and only authorized TNCCs be allowed to load an IMC. There are well known methods to control what code is loaded by a TNCC:

- Generate a cryptographic hash on the code image and verify it against a list of good hashes
- Verify the software publisher using certificates
- Control access to the IMC registration mechanism (registry or configuration file)
- Control access to IMC code and critical data files
- Employ a TNCC-specific list of authorized IMCs

Similar checks can be performed by the operating system before loading the TNCC.

The addition of a Platform Trust Service (PTS) may provide the above listed services and may also use hardware such as the Trusted Platform Module (TPM) to establish a trusted load path on a platform which is rooted in hardware. In short, every loader entity on the platform is measured before it loads another component, and the measured loaders are expected to log their measurements with corresponding verification signatures in the TPM.

Information disclosure attacks can be prevented by creating security associations between IMCs and IMVs. This does not preclude an additional security association between a NAR and a NAA.

To prevent/detect denial of service attacks, API usage from registered IMCs can be monitored.

However, stronger protection against rogue IMCs can be provided by having the TNCC launch a new “child” process for each IMC, having the child process load the IMC, and then having the TNCC communicate with the child processes carefully. This limits the amount of damage that can be done by a rogue IMC. The TNCC may use this approach but is not required to do so.

This specification requires that all valid IMCs be installed to a protected system directory. The loading of a rogue IMC can be mitigated (not prevented) by requiring privileged access to the registry key or config file. Note, however, that some (usually legacy) operating systems have no concept of a "protected" directory, registry, or file, and thus are provided no protection from this scenario. Note that this approach requires best practices for the use of protected directories and registries; if a user has any administrative access to these objects, they are vulnerable to a social engineering approach to causing a Trojan IMC to be installed.

IMC implementers who choose a stub-to-application implementation must take care not to make the stub-to-application communications the “weak link” in the security chain. They should choose protocols which maintain integrity and confidentiality as required, while taking into account the need for efficiency.

6 C Header File

This section provides a C header file that serves as a binding for the IF-IMC API with the C language and the Microsoft Windows DLL platform binding. As noted in section 3.1, implementers SHOULD use the C language binding when possible for maximum compatibility with other IMCs and TNC Clients on their platform.

```
/* tncifimc.h
 *
 * Trusted Network Connect IF-IMC API version 1.00 Revision 3
 * Microsoft Windows DLL Platform Binding C Header
 * May 3, 2005
 */

#ifndef _TNCIFIMC_H
#define _TNCIFIMC_H

#ifdef __cplusplus
extern "C" {
#endif

#ifdef WIN32
#ifdef TNC_IMC_EXPORTS
#define TNC_IMC_API __declspec(dllexport)
#else
#define TNC_IMC_API __declspec(dllimport)
#endif
#else
#define TNC_IMC_API
#endif

/* Basic Types */

typedef unsigned long TNC_UInt32;
typedef unsigned char *TNC_BufferReference;

/* Derived Types */

typedef TNC_UInt32 TNC_IMCID;
typedef TNC_UInt32 TNC_ConnectionID;
typedef TNC_UInt32 TNC_ConnectionState;
typedef TNC_UInt32 TNC_RetryReason;
typedef TNC_UInt32 TNC_MessageType;
typedef TNC_MessageType *TNC_MessageTypeList;
typedef TNC_UInt32 TNC_VendorID;
typedef TNC_UInt32 TNC_MessageSubtype;
typedef TNC_UInt32 TNC_Version;
typedef TNC_UInt32 TNC_Result;

/* Function pointers */

typedef TNC_Result (*TNC_IMC_InitializePointer)(
    TNC_IMCID imcID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);
typedef TNC_Result (*TNC_IMC_NotifyConnectionChangePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);
typedef TNC_Result (*TNC_IMC_BeginHandshakePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMC_ReceiveMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_IMC_BatchEndingPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMC_TerminatePointer)(
    TNC_IMCID imcID);
typedef TNC_Result (*TNC_TNCC_ReportMessageTypesPointer)(
    TNC_IMCID imcID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCC_SendMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_TNCC_RequestHandshakeRetryPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);
typedef TNC_Result (*TNC_TNCC_BindFunctionPointer)(
    TNC_IMCID imcID,
    char *functionName,
    void **pOutfunctionPointer);
typedef TNC_Result (*TNC_IMC_ProvideBindFunctionPointer)(
    TNC_IMCID imcID,
    TNC_TNCC_BindFunctionPointer bindFunction);

/* Result Codes */

#define TNC_RESULT_SUCCESS 0
#define TNC_RESULT_NOT_INITIALIZED 1
#define TNC_RESULT_ALREADY_INITIALIZED 2
#define TNC_RESULT_NO_COMMON_VERSION 3
#define TNC_RESULT_CANT_RETRY 4
#define TNC_RESULT_WONT_RETRY 5
#define TNC_RESULT_INVALID_PARAMETER 6
#define TNC_RESULT_CANT_RESPOND 7
#define TNC_RESULT_ILLEGAL_OPERATION 8
#define TNC_RESULT_OTHER 9
#define TNC_RESULT_FATAL 10

/* Version Numbers */

#define TNC_IFIMC_VERSION_1 1
```

```
/* Network Connection ID Values */

#define TNC_CONNECTIONID_ANY 0xFFFFFFFF

/* Network Connection State Values */

#define TNC_CONNECTION_STATE_CREATE 0
#define TNC_CONNECTION_STATE_HANDSHAKE 1
#define TNC_CONNECTION_STATE_ACCESS_ALLOWED 2
#define TNC_CONNECTION_STATE_ACCESS_ISOLATED 3
#define TNC_CONNECTION_STATE_ACCESS_NONE 4
#define TNC_CONNECTION_STATE_DELETE 5

/* Handshake Retry Reason Values */

#define TNC_RETRY_REASON_IMC_REMEDIATION_COMPLETE 0
#define TNC_RETRY_REASON_IMC_SERIOUS_EVENT 1
#define TNC_RETRY_REASON_IMC_INFORMATIONAL_EVENT 2
#define TNC_RETRY_REASON_IMC_PERIODIC 3
/* reserved for TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE: 4 */
/* reserved for TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE: 5 */
/* reserved for TNC_RETRY_REASON_IMV_SERIOUS_EVENT: 6 */
/* reserved for TNC_RETRY_REASON_IMV_MINOR_EVENT: 7 */
/* reserved for TNC_RETRY_REASON_IMV_PERIODIC: 8 */

/* Vendor ID Values */

#define TNC_VENDORID_TCG 0
#define TNC_VENDORID_ANY ((TNC_VendorID) 0xffffff)

/* Message Subtype Values */

#define TNC_SUBTYPE_ANY ((TNC_MessageSubtype) 0xff)

/* IMC Functions */

TNC_IMC_API TNC_Result TNC_IMC_Initialize(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_Version minVersion,
/*in*/  TNC_Version maxVersion,
/*out*/ TNC_Version *pOutActualVersion);

TNC_IMC_API TNC_Result TNC_IMC_NotifyConnectionChange(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_ConnectionState newState);

TNC_IMC_API TNC_Result TNC_IMC_BeginHandshake(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMC_API TNC_Result TNC_IMC_ReceiveMessage(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference messageBuffer,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);
```

```
TNC_IMC_API TNC_Result TNC_IMC_BatchEnding(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMC_API TNC_Result TNC_IMC_Terminate(
/*in*/  TNC_IMCID imcID);

TNC_IMC_API TNC_Result TNC_IMC_ProvideBindFunction(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_TNCC_BindFunctionPointer bindFunction);

/* TNC Client Functions */

TNC_Result TNC_TNCC_ReportMessageTypes(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_MessageTypeList supportedTypes,
/*in*/  TNC_UInt32 typeCount);

TNC_Result TNC_TNCC_SendMessage(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference message,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);

TNC_Result TNC_TNCC_RequestHandshakeRetry(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_RetryReason reason);

TNC_Result TNC_TNCC_BindFunction(
/*in*/  TNC_IMCID imcID,
/*in*/  char *functionName,
/*out*/ void **pOutfunctionPointer);

#ifdef __cplusplus
}
#endif

#endif
```

7 Use Case Walkthrough

This section provides an informative (non-binding) walkthrough of a typical TNC use case, showing how IF-IMC supports the use case. The text describing IF-IMC usage is in **bold**. Sequence diagrams that illustrate the main parts of this walkthrough are included at the end of this section.

7.1 Configuration

1. The IT administrator configures any addressing and security information needed for server-side components (PEP, NAA, TNCS, and IMVs) to securely contact each other. The client-side components (TNCC and IMCs) find each other automatically using Microsoft Windows registry or a configuration file modified at install time. The manner in which the NAR and TNCC find each other is not specified.
2. The IT administrator configures policies in the NAA, TNCS, and IMVs for what sorts of user authentication, platform authentication, and integrity checks are required when.

7.2 TNCS Startup

1. When the TNCS starts up, the TNCS loads the IMVs.

   The TNCS initializes the IMVs through IF-IMV.

7.3 TNCC Startup

1. When the TNCC starts up, the TNCC loads the IMCs. **[IF-IMC] The details of the load process are platform-specific. With the Microsoft Windows DLL binding, the TNCC reads a protected registry key to find the IMC DLLs, then loads them.**

   During the load process, the TNCC may check the integrity of the IMCs. This is optional. If a TPM is present, this check will typically involve hashing the IMCs and adding their hashes to a PCR. If no TPM is present, this check may involve checking the signatures on the IMCs. Integrity checks during IMC loading are done completely by the TNCC since there is no TNCS or IMV available. TNCS and IMVs will get a chance to do platform authentication of the endpoint platform later on.

2. The TNCC initializes the IMCs through IF-IMC. **[IF-IMC] The TNCC calls `TNC_IMC_Initialize` for each IMC. The IMC performs any initialization it may need to, such as connecting to a background process or starting threads (if permitted by the platform binding).**

3. **[IF-IMC] The TNCC performs any other platform-specific initialization needed. With the Microsoft Windows DLL binding, the TNCC calls the `TNC_IMC_ProvideBindFunction` function to give each IMC a pointer to the bind function (`TNC_TNCC_BindFunction`) used for Dynamic Function Binding.**

7.4 Network Connect

1. The endpoint's NAR attempts to connect to a network protected by a PEP, thus triggering an Integrity Check Handshake. There are other ways that an Integrity Check Handshake can be triggered, but this will probably be the most common. For those other ways, the next few steps may be significantly different.

2. The PEP sends a network access decision request to the NAA. The ordering of user authentication, platform authentication, and integrity check is subject to configuration. Here we present what will probably be the most common order: first user authentication, then platform authentication, then integrity check.

3. The NAA performs user authentication with the NAR. Based on the NAA's policy, the user identity established through this process may be used to make immediate access decisions

(like deny). If an immediate access decision has been made, skip to step 17. User authentication may also involve having the NAR authenticate the NAA.

4. The NAA informs the TNCS of the connection request, providing the user identity and other useful info (service requested, etc.).
5. The TNCS performs platform authentication with the TNCC, if required by TNCS policy. This includes verifying the IMC hashes collected during TNCC Setup. If an immediate access decision has been made, skip to step 16. Platform authentication may be mutual so the TNCC can be sure it's talking to a secure server.
6. The TNCC uses IF-IMC to fetch IMC messages. **[IF-IMC] If this is a new network connection, the TNCC calls `TNC_IMC_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_CREATE` to indicate that a new network connection has been created. The TNCC calls `TNC_IMC_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_HANDSHAKE` to indicate that a new Integrity Check Handshake is starting. The TNCC calls `TNC_IMC_BeginHandshake` to inform the IMCs that a new Integrity Check Handshake is starting and they should send their messages. The IMCs call `TNC_TNCC_SendMessage` to give their messages to the TNCC and then return from `TNC_IMC_BeginHandshake` to indicate that they are done sending messages for this batch.**
7. The TNCS uses IF-IMV to inform each IMV that an Integrity Check Handshake has started.
8. The TNCC passes the IMC messages to the TNCS. This and all other TNCC-TNCS communications can be sent directly but they will often be relayed through one or more of the NAR, PEP, and NAA.
9. The TNCS passes each IMC message to the matching IMV or IMVs through IF-IMV (using message types associated with the IMC messages to find the right IMV). If there are no IMC messages, skip to step 13.
10. Each IMV analyzes the IMC messages. If an IMV needs to exchange more messages (including remediation instructions) with an IMC, it provides a message to the TNCS through IF-IMV. If an IMV is ready to decide on an IMV Action Recommendation and IMV Evaluation Result, it gives this result to the TNCS through IF-IMV. If there are no more messages to be sent to the IMC from any of the IMVs, skip to step 13.
11. The TNCS sends the messages from the IMVs to the TNCC.
12. The TNCC sends the IMV messages on to the IMCs through IF-IMC so they can process the messages and respond. Skip to step 8. **[IF-IMC] The TNCC delivers the IMV messages to the IMCs via `TNC_IMC_ReceiveMessage`. The IMCs may call `TNC_TNCC_SendMessage` before returning from `TNC_IMC_ReceiveMessage` if they want to send a response. When the TNCC has delivered all the IMV messages to the IMCs, it calls `TNC_IMC_BatchEnding` to inform them of this fact. The IMCs may call `TNC_TNCC_SendMessage` before returning from `TNC_IMC_BatchEnding` if they want to send a message to an IMV.**
13. If there are any IMVs that have not given an IMV Action Recommendation to the TNCS, they are prompted to do so through IF-IMV.
14. The TNCS considers the IMV Action Recommendations supplied by the IMVs and uses an integrity check combining policy to decide what its TNCS Action Recommendation should be.
15. The TNCS sends a copy of its TNCS Action Recommendation to the TNCC. The TNCS also informs the IMVs of its TNCS Action Recommendation via IF-IMV.
16. The TNCS sends its TNCS Action Recommendation to the NAA. The NAA may ignore or modify this recommendation based on its policies but will typically abide by it.

17. The NAA sends its network access decision to the PEP.
18. The PEP implements the network access decision. During this process, the NAR may be informed of the decision. The TNCC may be informed by the NAR or may discover that a new network has come up.
19. If step 6 was not executed, the network connect process is complete. Otherwise, the TNCC informs the IMCs of the TNCS Action Recommendation via IF-IMC. **[IF-IMC] The TNCC signals this change in network connection state through the `TNC_IMC_NotifyConnectionChange` function.**
20. If the IMCs or the applications that they represent need to perform remediation, they perform that remediation. Then they continue with Handshake Retry after Remediation. If no remediation was needed, the use case ends here.

7.5 Handshake Retry After Remediation

1. When an IMC completes remediation, it informs the TNCC that its remediation is complete and requests a retry of the Integrity Check Handshake through IF-IMC. **[IF-IMC] The IMC signals this by calling the `TNC_TNCC_RequestHandshakeRetry` function.**
2. The TNCC decides whether to initiate an Integrity Check Handshake retry (possibly depending on policy, user interaction, etc.). Depending on limitations of the NAR, the TNCC may need to disconnect from the network and reconnect to retry the Integrity Check Handshake. In that case (especially if the previous handshake resulted in full access), it may decide to skip the handshake retry. However, in many cases the TNCC will be able to retry the handshake without disrupting network access. It may even be able to retain the state established in the earlier handshake. If the TNCC decides to skip the Integrity Check Handshake retry, the use case ends here.
3. The TNCC initiates a retry of the handshake. Skip to step 1, 3, or 5 of the Network Connect section above, depending on which steps are needed to initiate the retry.

7.6 Handshake Retry Initiated by TNCS

1. The TNCS can recheck the security state of the AR periodically or when integrity policies change (such as when a new patch is required) by requesting another Integrity Check Handshake with the TNCC. The Integrity Check Handshake retry can be done through the PEP or by communicating directly with the TNCC. State from the previous handshake may be retained or not. An IMV can also request an integrity handshake retry through IF-IMV. If the TNCS decides to skip the Integrity Check Handshake retry, the use case ends here.
2. The TNCS initiates a retry of the handshake. Skip to step 3 or 5 of the Network Connect section above, depending on whether user authentication will be done in the retry.

7.7 Sequence Diagram for Network Connect

The following sequence diagram (Figure 1) illustrates the Network Connect use case, as described in section 7.4.

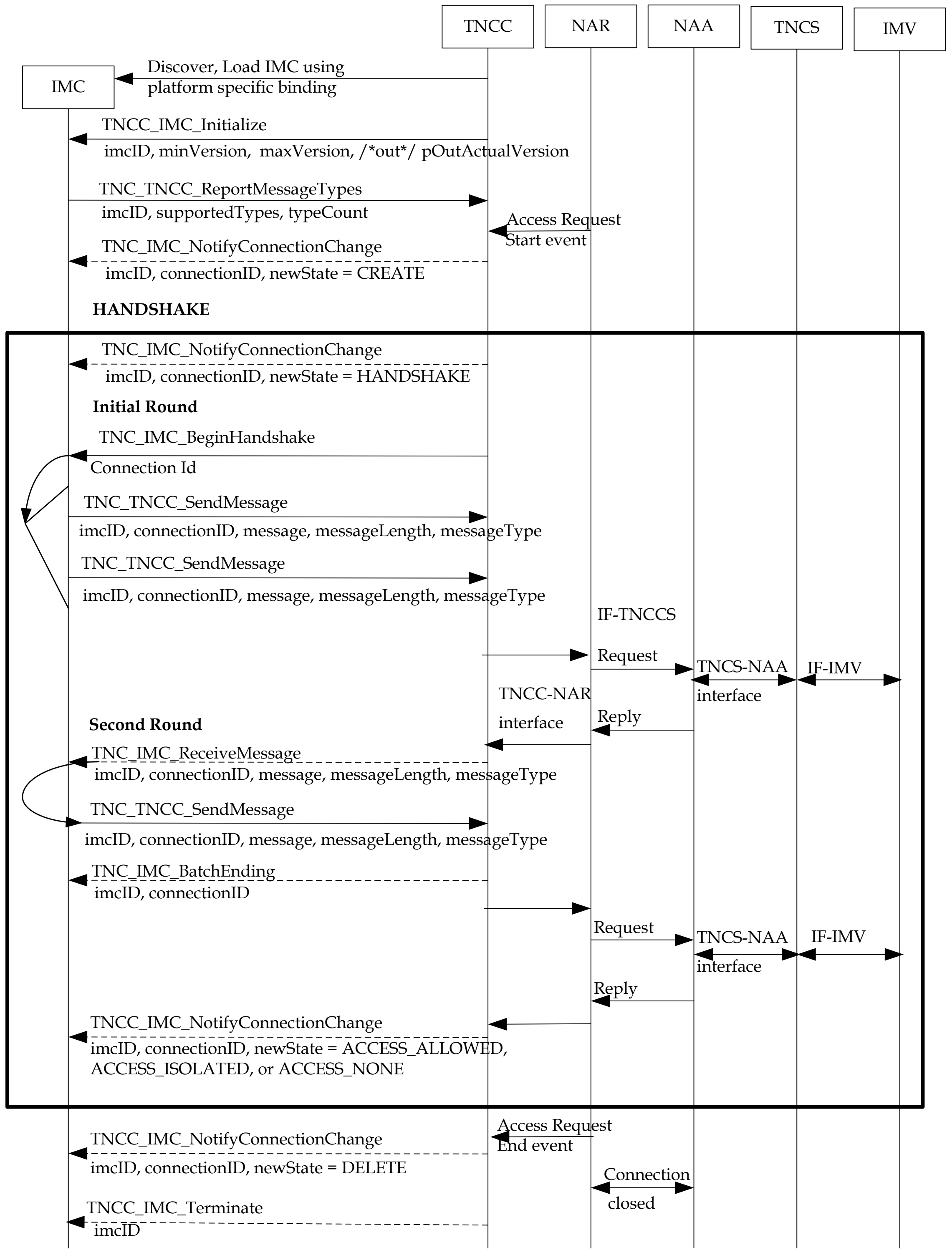

TNCC
NAR
NAA
TNCS
IMV
IMC
Discover, Load IMC using platform specific binding
TNCC_IMC_Initialize
imcID, minVersion, maxVersion, /*out*/ pOutActualVersion
TNC_TNCC_ReportMessageTypes
imcID, supportedTypes, typeCount
Access Request Start event
TNC_IMC_NotifyConnectionChange
imcID, connectionID, newState = CREATE
HANDSHAKE
TNC_IMC_NotifyConnectionChange
imcID, connectionID, newState = HANDSHAKE
Initial Round
TNC_IMC_BeginHandshake
Connection Id
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
IF-TNCCS
Request
TNCS-NAA interface
IF-IMV
TNCC-NAR interface
Reply
Second Round
TNC_IMC_ReceiveMessage
imcID, connectionID, message, messageLength, messageType
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
TNC_IMC_BatchEnding
imcID, connectionID
Request
TNCS-NAA interface
IF-IMV
Reply
TNCC_IMC_NotifyConnectionChange
imcID, connectionID, newState = ACCESS_ALLOWED, ACCESS_ISOLATED, or ACCESS_NONE
Access Request End event
TNCC_IMC_NotifyConnectionChange
imcID, connectionID, newState = DELETE
Connection closed
TNCC_IMC_Terminate
imcID

Figure 1 - IF-IMC Network Connect Sequence Diagram

7.8 Sequence Diagram for Handshake Retry After Remediation

The following sequence diagram (Figure 2) illustrates the Handshake Retry After Remediation use case, as described in section 7.5.

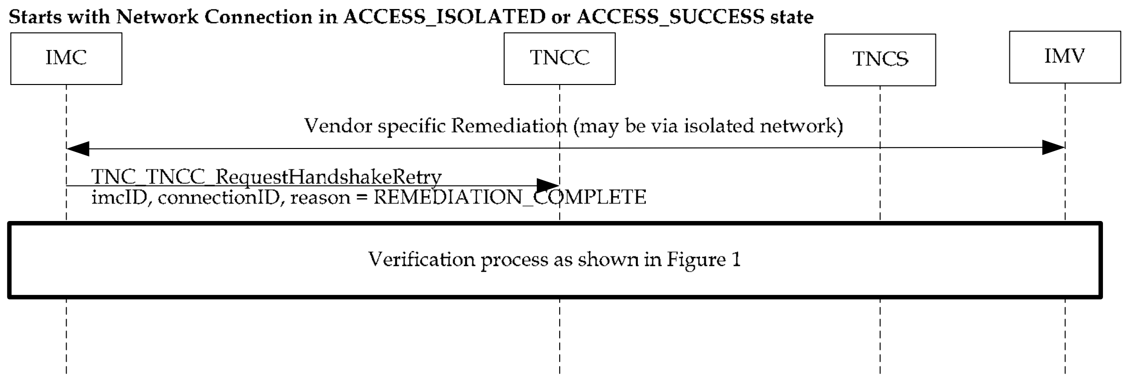

Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMC
TNCC
TNCS
IMV
Vendor specific Remediation (may be via isolated network)
TNC_TNCC_RequestHandshakeRetry
imcID, connectionID, reason = REMEDIATION_COMPLETE
Verification process as shown in Figure 1

Figure 2 - IF-IMC Handshake Retry After Remediation Sequence Diagram

7.9 Sequence Diagram for Handshake Retry Initiated by TNCS

The following sequence diagram (Figure 3) illustrates the Handshake Retry Initiated by TNCS use case, as described in section 7.6.

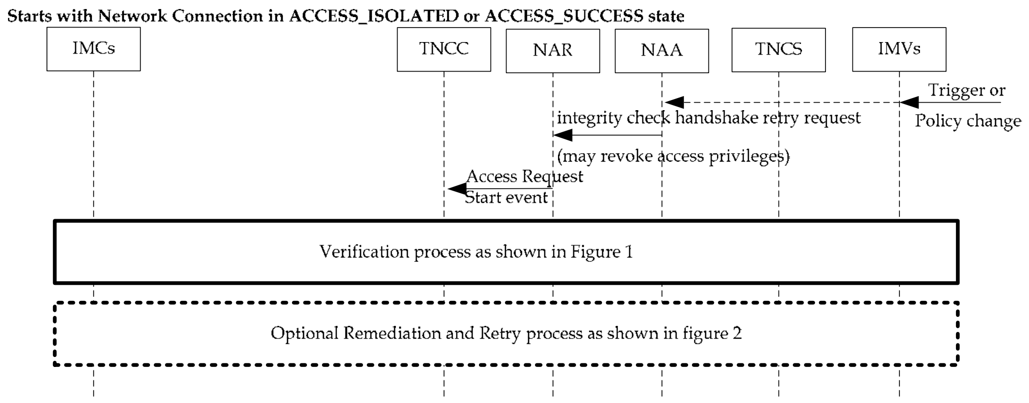

Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMCs
TNCC
NAR
NAA
TNCS
IMVs
Trigger or
Policy change
integrity check handshake retry request
(may revoke access privileges)
Access Request
Start event
Verification process as shown in Figure 1
Optional Remediation and Retry process as shown in figure 2

Figure 3 - IF-IMC Handshake Retry Initiated by TNCS Sequence Diagram

8 Implementing a Simple IMC

This section provides a brief informative (non-binding) description of how to implement a simple IMC, one that only reports a value to an IMV (the operating system version, for instance).

This example assumes that you're using the Microsoft Windows DLL platform binding. If not, replace the instructions in section 8.3 about `TNC_IMC_ProvideBindFunction` with your platform's Dynamic Function Binding mechanism.

8.1 Decide on a Message Type and Format

First, you must decide what message type you will use to send your value to the IMV and what the format of the message will be. This may involve getting a Vendor ID as described in section 3.2.3. Then implement the following functions as described here.

8.2 TNC_IMC_Initialize

All IMCs must implement the `TNC_IMC_Initialize` function. In your implementation, determine whether you support any of the listed IF-IMC API versions. If not, return `TNC_RESULT_NO_COMMON_VERSION`. If so, store the mutually agreed upon version number at `pOutActualVersion` and initialize the IMC. Return `TNC_RESULT_SUCCESS` if all goes well. Normally, you might store your IMC ID for later use but in this example all of your code is called by the TNCC so you have the IMC ID as a parameter to all your functions.

8.3 TNC_IMC_ProvideBindFunction

Use the bind function to get a pointer to `TNC_TNCC_SendMessage` for later use. This is the only state you need to keep. Return `TNC_RESULT_SUCCESS` unless the bind function reports an error. In that case, return `TNC_RESULT_FATAL`.

8.4 TNC_IMC_BeginHandshake

When a new Integrity Check Handshake starts, you just want to send your value and then you're done for the rest of the handshake. To implement this function, call the pointer to `TNC_TNCC_SendMessage` that you saved earlier. Pass in the IMC ID and network connection ID provided to `TNC_IMC_BeginHandshake`, a pointer and length for your message, and the message type you decided on. If `TNC_TNCC_SendMessage` returns an error, then return that. Otherwise, return `TNC_RESULT_SUCCESS`.

8.5 All Done!

That's it! You've implemented your first IMC. If you need to do anything special on termination, you can implement `TNC_IMC_Terminate`. But many IMCs won't need to.

9 References

9.1 Normative References

[1] Trusted Computing Group, *TNC Architecture for Interoperability*, Specification Version 1.1, May 2006.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", Internet Engineering Task Force RFC 2119, March 1997.

[3] Crocker, D., P. Overell, "Augmented BNF for Syntax Specifications: ABNF", Internet Engineering Task Force RFC 2234, November 1997.

9.2 Informative References

[4] Trusted Computing Group, *TNC IF-IMV*, Specification Version 1.1, May 2006.

[5] ISO, ISO/IEC 9899:1999, Programming Languages – C, 1999.