TCG Trusted Network Connect
TNC IF-IMV

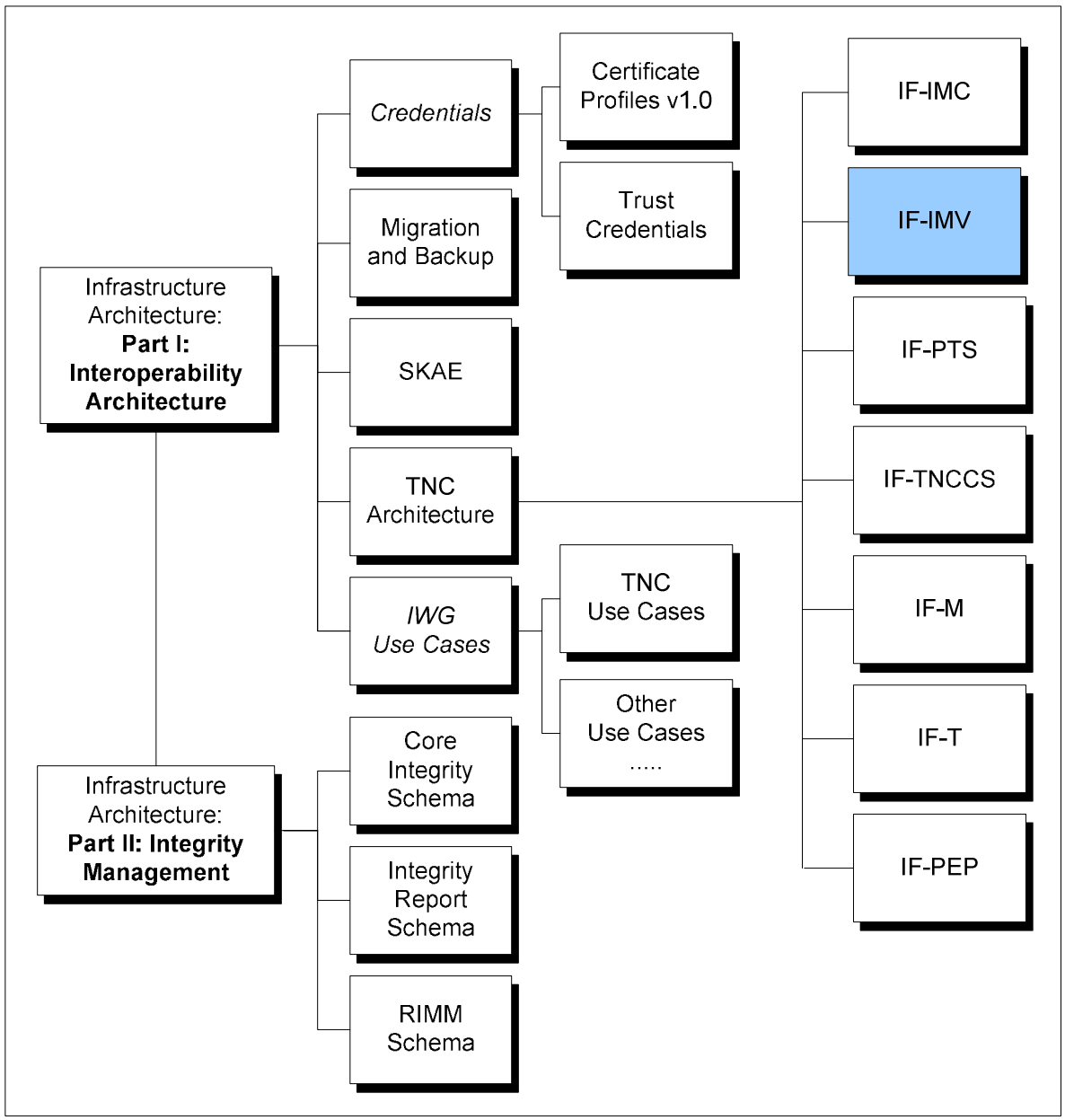
Specification Version 1.2
Revision 8
5 February 2007
Published

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:

Aman Garg	3Com
Bipin Mistry	3Com
Mahalingam Mani	Avaya
Mark Beadles (Editor of IF-IMV 1.0)	Endforce, Inc.
Hidenobu Ito	Fujitsu Limited
Sung Lee	Fujitsu Limited
Kazuaki Nimura	Fujitsu Limited
Boris Balacheff	Hewlett-Packard
Paul Crandell	Hewlett-Packard
Mauricio Sanchez	Hewlett-Packard
Diana Arroyo (Editor)	IBM
Lee Terrell	IBM
Frank Yeh	IBM
Stuart Bailey	Infoblox
Tina Bird	InfoExpress, Inc.
Ravi Sahita	Intel Corporation
Ned Smith	Intel Corporation
Barbara Nelson	iPass
Chris Trytten	iPass
Steve Hanna (Editor, TNC co-chair)	Juniper Networks, Inc.
John Jerrim	Lancope, Inc.
Gene Chang	Meetinghouse Data Communications
Alex Romanyuk	Meetinghouse Data Communications
John Vollbrecht	Meetinghouse Data Communications
Sandilya Garimella	Motorola
Joseph Tardo	Nevis Networks
Pasi Eronen	Nokia Corporation
Jeff Six	National Security Agency
Meenakshi Kaushik	Nortel Networks
Thomas Hardjono	SignaCert, Inc.
Babak Salimi	Sygate Technologies, Inc.
Bryan Kingsford	Symantec
Paul Sangster (TNC co-chair)	Symantec
Rod Murchison	Vernier Networks
Michele Sommerstad	Vernier Networks
Scott Cochrane	Wave Systems
Greg Kazmierczak	Wave Systems

Table of Contents

1 Scope and Audience 8
2 Purpose and Requirements 9
2.1 Purpose of IF-IMV 9
2.2 Summary of Changes since IF-IMV 1.1 9
2.3 Supported Use Cases 9
2.4 Unsupported Use Cases 10
2.5 Requirements 10
2.5.1 Non-Requirements 12
2.6 Assumptions 12
2.7 Keywords 12
2.8 Abstract API Naming Conventions 13
2.9 Features Provided by IF-IMV 13
2.9.1 Integrity Check Handshake 13
2.9.2 Connection Management 13
2.9.3 Remediation and Handshake Retry 14
2.9.4 Message Delivery 14
2.9.5 Reliability 15
2.9.6 Batches 15
2.9.7 IMV Action Recommendation 16
2.9.8 Reason String 16
2.9.9 Stateless IMVs 17
2.9.10 IMVs with Remote Servers 17
3 IF-IMV Abstract API 18
3.1 Platform and Language Independence 18
3.2 Extensibility 18
3.2.1 API Version 18
3.2.2 Dynamic Function Binding 18
3.2.3 Vendor IDs 19
3.2.4 Vendor-Specific Functions 19
3.3 Threading and Reentrancy 19
3.4 Data Types 19
3.4.1 Basic Types 19
3.4.2 Derived Types 20
3.5 Defined Constants 23
3.5.1 Boolean Values 23
3.5.2 Result Code Values 23
3.5.3 Version Numbers 24
3.5.4 Network Connection ID Values 24
3.5.5 Network Connection State Values 24
3.5.6 Handshake Retry Reason Values 25
3.5.7 IMV Action Recommendation Values 25
3.5.8 IMV Evaluation Result Values 26
3.5.9 Vendor ID Values 26
3.5.10 Message Subtype Values 26
3.5.11 Attribute ID Values and Value Definitions 27
3.6 Mandatory and Optional Functions 28
3.7 IMV Functions 28
3.7.1 TNC_IMV_Initialize (MANDATORY) 28
3.7.2 TNC_IMV_NotifyConnectionChange (OPTIONAL) 30
3.7.3 TNC_IMV_ReceiveMessage (OPTIONAL) 30
3.7.4 TNC_IMV_SolicitRecommendation (MANDATORY) 32
3.7.5 TNC_IMV_BatchEnding (OPTIONAL) 33
3.7.6 TNC_IMV_Terminate (OPTIONAL) 34
3.8 TNC Server Functions 34

3.8.1 TNC_TNCS_ReportMessageTypes (MANDATORY) 34
3.8.2 TNC_TNCS_SendMessage (MANDATORY) 35
3.8.3 TNC_TNCS_RequestHandshakeRetry (MANDATORY) 37
3.8.4 TNC_TNCS_ProvideRecommendation (MANDATORY) 37
3.8.5 TNC_TNCS_GetAttribute (OPTIONAL) 39
3.8.6 TNC_TNCS_SetAttribute (OPTIONAL) 40
4 Platform Bindings 40
4.1 Microsoft Windows DLL Platform Binding 40
4.1.1 Finding, Loading, and Unloading IMVs 40
4.1.2 Dynamic Function Binding 40
4.1.3 Threading 40
4.1.4 Platform-Specific Bindings for Basic Types 40
4.1.5 Platform-Specific Bindings for Derived Types 40
4.1.6 Additional Platform-Specific Derived Types 40
4.1.7 Platform-Specific IMV Functions 40
4.1.8 Platform-Specific TNC Server Functions 40
4.1.9 Well-known Registry Key 40
4.2 UNIX/Linux Dynamic Linkage Platform Binding 40
4.2.1 Finding, Loading, and Unloading IMVs 40
4.2.2 Dynamic Function Binding 40
4.2.3 Format of `/etc/tnc_config` 40
4.2.4 Threading 40
4.2.5 Platform-Specific Bindings for Basic Types 40
4.2.6 Platform-Specific Bindings for Derived Types 40
4.2.7 Additional Platform-Specific Derived Types 40
4.2.8 Platform-Specific IMV Functions 40
4.2.9 Platform-Specific TNC Server Functions 40
4.3 Java Platform Binding 40
4.3.1 Object Orientation 40
4.3.2 Exception Handling 40
4.3.3 Limited Privileges 40
4.3.4 Finding, Loading, and Unloading IMVs 40
4.3.5 Dynamic Function Binding 40
4.3.6 Format of the tnc_config file 40
4.3.7 Location of the tnc_config file 40
4.3.8 Threading 40
4.3.9 Attributes 40
4.3.10 Platform-Specific Bindings for Basic Types 40
4.3.11 Platform-Specific Bindings for Derived Types 40
4.3.12 Interface and Class Definitions 40
5 Security Considerations 40
5.1 Threat analysis 40
5.1.1 Registration and Discovery based threats 40
5.1.2 Rogue IMV threats 40
5.1.3 Rogue TNCS threats 40
5.1.4 Man-in-the-Middle Threats 40
5.1.5 Tampering Threats on IMVs and TNCSs 40
5.1.6 Threats Beyond IF-IMV 40
5.2 Suggested remedies 40
6 C Header File 40
7 Use Case Walkthrough 40
7.1 Configuration 40
7.2 TNCS Startup 40
7.3 TNCC Startup 40
7.4 Network Connect 40

7.5 Handshake Retry After Remediation 40
7.6 Handshake Retry Initiated by TNCS 40
7.7 C Binding Sequence Diagrams 40
7.7.1 Sequence Diagram for Network Connect 40
7.7.2 Sequence Diagram for Handshake Retry After Remediation 40
7.7.3 Sequence Diagram for Handshake Retry Initiated by TNCS 40
7.8 Java Binding Sequence Diagrams 40
7.8.1 Sequence Diagram for Network Connect 40
7.8.2 Sequence Diagram for Handshake Retry After Remediation 40
7.8.3 Sequence Diagram for Handshake Retry Initiated by TNCS 40
8 Implementing a Simple IMV 40
8.1 Decide on a Message Type and Format 40
8.2 TNC_IMV_Initialize 40
8.3 TNC_IMV_ProvideBindFunction 40
8.4 TNC_IMV_ReceiveMessage 40
8.5 TNC_IMV_SolicitRecommendation 40
8.6 All Done! 40
9 References 40
9.1 Normative References 40
9.2 Informative References 40

1 Scope and Audience

The Trusted Network Connect Work Group (TNC-WG) has defined 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-IMV, a standard interface between Integrity Measurement Verifiers and the TNC Server. This document defines and specifies IF-IMV.

Architects, designers, developers and technologists who wish to implement, use, or understand IF-IMV 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 Purpose and Requirements

2.1 Purpose of IF-IMV

This document describes and specifies IF-IMV, a critical interface in the Trusted Computing Group's Trusted Network Connect (TNC) architecture. IF-IMV is the interface between Integrity Measurement Verifiers (IMVs) and a TNC Server (TNCS). It is closely related to IF-IMC [6], the interface between Integrity Measurement Clients (IMCs) and a TNC Client (TNCC).

IF-IMV is primarily used to receive integrity measurements sent from client-side Integrity Measurement Collectors (IMCs) to corresponding 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. It also allows IMVs to supply their recommendations to the TNCS.

An API-based approach has been chosen as the preferred embodiment of IF-IMV, similar to IF-IMC [6]. See Section 3 of this document for description of the abstract API and Section 4 for specific platform bindings.

2.2 Summary of Changes since IF-IMV 1.1

The following changes have been made to IF-IMV since the last version (IF-IMV 1.1):

- Documented use cases
- Added Java Platform Binding
- Removed note saying that Linux/UNIX Binding is preliminary
- Added optional support for having IMVs supply reason strings

2.3 Supported Use Cases

Use cases that this version of IF-IMV supports are as follows:

- An IMV and TNCS that support the same platform binding are installed on an endpoint. The TNCS finds and loads the IMV. Then it runs one or more Integrity Check Handshakes. The IMV and TNCS may use any of the features of IF-IMV.
- A TNCS has restricted privileges. It loads IMVs and runs one or more Integrity Check Handshakes.
- A TNCS that supports the Java Platform Binding runs with generous privileges but chooses to run IMVs with restricted privileges for security reasons. It loads IMVs and runs one or more Integrity Check Handshakes.
- An IMV and a TNCS both support the reason string extensions to IF-IMV. An IMV provides a reason string to a TNCS, giving the reason for its IMV Action Recommendation. The TNCS logs this reason and/or passes it on to the TNCC through an extension to IF-TNCCS. At the TNCC, the reason information may be displayed to the user (perhaps in a detailed view). The reason string may be in the endpoint user's preferred language or (if that language is not available) in another language.
- A TNCS is running. When an IMV is installed or uninstalled, the TNCC notices this and loads or unloads the IMV.

Note that several of these use cases were supported by previous versions of IF-IMV. Previous versions of the IF-IMV specification did not document use cases.

2.4 Unsupported Use Cases

Several use cases, including but not limited to this one, are not covered by (but not prevented by) this version of IF-IMV:

- A multi-user endpoint has several active users with different preferred languages. An IMV provides several reason strings, one for each of those different languages.

2.5 Requirements

The following are the requirements which IF-IMV must meet in order to successfully play its role in the TNC architecture. These are stated as general requirements, with specific requirements called out as appropriate.

a. Meets the needs of the TNC architecture

IF-IMV must support all the functions and use cases described in the TNC architecture as they apply to the relationship between the TNC Server and IMV components.

Specific requirements include:

- The API must support multiple overlapping network connections and Integrity Check Handshakes for a single TNCS from multiple TNCCs, and communication between the TNCS and multiple IMVs.
- The API must allow an IMV to act as a front end for one or more back-end applications or remote servers, or not to act as a front end at all, as determined by the IMV implementer.
- IF-IMV must have some mechanism for IMVs to recommend isolation and compliance information to the TNCS, so that isolation can properly be supported on the network. This may stop short of an explicit mechanism for knowing which network to assign for isolation, but there must be a way to pass intelligence from IMVs to the TNCS.
- IMVs MUST be able to recommend initiation of an Integrity Check Handshake retry.

b. 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.

Specific requirements include:

- The security considerations include requirements that unauthorized parties cannot observe communications between the IMV and the TNC Server; that only authorized IMVs can communicate with the TNC Server across IF-IMV and thence to the IMCs; and that no party can cause denial of service to any of the system components. See the Security Considerations section of this document for detailed discussion.

c. 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 in IF-IMV is also important when considering that TNCSs

and IMVs are server-side components which may be required to handle messages from thousands to millions of remote clients.

d. Extensible

IF-IMV needs to expand over time as new features are added to the TNC architecture. For instance, this version of IF-IMV adds support for reason strings. IF-IMV 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.

e. Scalable

IF-IMV is an interface in a critical server-side architecture and must support scalability levels appropriate to this role. Enterprise and service provider deployments of TNC server architectures may be required to support up to millions of clients and a corresponding load of message transactions. IF-IMV should allow TNC Servers to employ load balancing, failover, and other techniques to achieve scalability.

f. Reliable

Reliability of network operations is critical. IF-IMV must support reliable communications, as well as reliable implementations and deployments of TNCSs and IMVs. For example, it should be possible for vendors to implement redundancy features.

g. Easy to use and implement

IF-IMV should be easy for TNC Server and IMV 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. IF-IMV should also make things easy for system administrators and end-users. Components of the TNC architecture should plug together automatically without requiring extensive manual configuration.

h. Platform-independent

Since there is a wide variety of platforms which are deployed in server-side systems, IF-IMV must function on as many server platforms as possible. At least Java, Windows, Linux (most common flavors), and other UNIX variants must be supported. Platform bindings included in this specification describe how platform-specific issues are handled.

i. Language-independent

IF-IMV must support the widest possible variety of programming languages: C, C++, C#, Java, Visual Basic, assembly language, and others. Therefore, this specification defines an abstract API and language-specific bindings. All language-specific bindings are required to support all capabilities of the abstract API.

j. Allow Java IMVs and TNCSs to contain or interface with native code

TNC components that use the Java Platform Binding may need to include or interface with native (non-Java) code. The Java Platform Binding should not include any explicit support for this but it should not prevent it either.

k. Internationalized

IF-IMV must be able to support a wide variety of human languages. In particular, IF-IMV must enable IMVs to provide reason strings in the endpoint user's preferred language.

2.5.1 Non-Requirements

There are certain requirements that IF-IMV explicitly is not required to meet. This list may not be exhaustive (complete).

a. There is **no requirement** that IF-IMV provide *explicit* mechanisms for redundancy and failover. It is acceptable that vendor IMVs and TNCSs are able to provide proprietary redundancy and failover mechanisms.

b. There is **no requirement** that IF-IMV provide support for a single endpoint with multiple users with different preferred languages.

c. There is **no requirement** that IF-IMV provide support for several TNCSs from different vendors running in a single Java Virtual Machine at the same time.

2.6 Assumptions

Here are the assumptions that IF-IMV 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. In the event that reliable delivery cannot be provided, the TNC Client is expected to terminate the connection.

- TNCS provides Action Recommendations for access decision

  It is assumed that the TNCS combines IMV Action Recommendations from multiple IMVs (using whatever logic) and provides a final TNCS Action Recommendation to the entity which makes the access decision. Outside the scope of this specification, there is assumed to be a mechanism or mechanisms for the TNCS to thus communicate with this entity (which may include, for example, NAAs and other PDP components). However, this mechanism is not part of IF-IMV and will not be specified in this document. This may be defined in a future phase of the TNC specifications process. Implementers are encouraged to become familiar with the TNC architecture [1], which includes a detailed discussion of these entities and interactions.

- Statefulness/Statelessness

  TNCSs and IMVs may be stateless with respect to any individual TNCCs/IMCs, or they may keep state. This is an implementation decision, not a requirement of the interface.

2.7 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.

2.8 Abstract API Naming Conventions

To avoid name conflicts, all identifiers in the IF-IMV Abstract API have a name that begins with "`TNC_`". Note that this only pertains to the IF-IMC Abstract API. Since Java includes good support for scoped names, the Java Platform Binding often omits this prefix.

Functions described in this document that are to be implemented by an IMV have a name that begins with "`TNC_IMV_`". 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 Server (known as "callbacks") have a name that begins with "`TNC_TNCS_`". 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.

2.9 Features Provided by IF-IMV

This section documents the features provided by IF-IMV.

2.9.1 Integrity Check Handshake

One of the primary functions of IF-IMV 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 Action Recommendations.

2.9.2 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 TNCS provides connection management features.

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

A TNC MAY maintain the same network connection ID across several 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, avoiding the need to resend data if it was sent in an earlier handshake and has not changed. Second, it allows an IMV to request a handshake retry for a particular connection, as when policies change. The TNCS MAY ensure that connection IDs persist long enough to permit handshake retry but this is purely optional. In contrast, TNCCs SHOULD retain connection IDs so that handshakes can be automatically retried after remediation is complete. It may seem problematic to have a TNCC retain its connection ID for a connection and not have the TNCS retain its connection ID for that connection. This does not actually cause problems since the connection IDs are local identifiers

(like a socket number) and are not shared by the TNCC and TNCS. The TNCS MUST use the same connection ID for all IMVs when referring to a particular connection.

2.9.3 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 TNCS MAY maintain a network connection ID after an Integrity Check Handshake has been completed. This network connection ID can then be used by the TNCS to inform IMVs that it is retrying the handshake or by an IMV to request a retry (due to policy change 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-IMV supports handshake retry and requires IMVs to handle handshake retries (which is usually trivial) but does not require TNCSs to honor IMV requests for handshake retry. In fact, IF-IMV requires an IMV to provide information about the reason for requesting handshake retry so that the 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 (see section 2.9.4, "Message Delivery"). There is no special support in IF-IMV for this feature. IMVs SHOULD send remediation instructions to IMCs before returning an IMV Action Recommendation and IMV Evaluation Result to the TNCS so the instructions are delivered before the handshake is completed.

2.9.4 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 below.

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. All messages sent by an IMV through IF-IMV have a recipient type of IMC. All messages received by an IMV through IF-IMV have a recipient type of IMV. 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-IMV allows an IMV 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 by reference 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 HTTP or another appropriate protocol.

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

A particular example of the message delivery provided by IF-IMV is the communication of remediation instructions from the IMVs through the TNCS to the TNCC/IMCs. This is one application of IMC-IMV message delivery and in all cases follows the normal IMV-IMC communications path. IF-IMV provides support for communicating remediation instructions to an endpoint using this mechanism. Since the normal IMC-IMV communications path is used to communicate remediation instructions, this specification will not address further the details of how remediation itself is done.

2.9.5 Reliability

For successful enterprise deployments, reliability of TNCSs and IMVs is important. To ensure this reliability, organizations may employ redundant TNCSs. Organizations may also require active failover as well as other features that provide a level of high availability for critical networks. Vendors and enterprises wishing to implement their systems incorporating redundancy should see the discussion of this topic in the TNC Architecture document [1].

2.9.6 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 IMV can only send a message in two circumstances: in response to a message received by the IMV in a batch (when `TNC_IMV_ReceiveMessage` is called), and at the end of a batch (when `TNC_IMV_BatchEnding` is called). In either of these circumstances, the IMV MAY send one or more messages by calling `TNC_TNCS_SendMessage` once for each message to be sent and then returning from `TNC_IMV_ReceiveMessage` or `TNC_IMV_BatchEnding`. Note that if the IMV does not call `TNC_TNCS_SendMessage` before returning from

TNC_IMV_ReceiveMessage, or TNC_IMV_BatchEnding, this indicates that it does not want to send any messages at this time. IMCs use a similar mechanism except that they can send messages in three circumstances: during the initial batch, in response to a message received by the IMC in a later batch, and 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 IMV that is not engaged in a dialog with an IMC may well find that the handshake has ended.

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

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

Similarly, an IMV might expect to receive a “working” status message from an IMC during a particular batch, and if so can respond in the next batch with a status query to the IMC to keep that handshake 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 TNCS will return TNC_RESULT_ILLEGAL_OPERATION from TNC_TNCS_SendMessage.

2.9.7 IMV Action Recommendation

One of the assumptions of the TNC architectural model is that IF-IMV provides a means for IMVs to recommend action information to the TNCS, so that isolation can properly be supported on the network. The TNCS then will combine these IMV Action Recommendations using some logic (defined by the TNCS implementers) to come up with an overall TNCS Action Recommendation. Note that the TNCS may choose to ignore any IMV Action Recommendation, but each IMV must be able to recommend an action. Potential choices for IMV Action Recommendations include: recommend full (normal) access; recommend isolation (limited or quarantined access); and recommend denial (no access). The mandatory function TNC_TNCS_ProvideRecommendation is the mechanism within IF-IMV for an IMV to indicate its IMV Action Recommendation.

2.9.8 Reason String

A new feature of IF-IMV 1.2 is that an IMV can supply a reason string to explain its IMV Action Recommendation. In order to preserve backward compatibility with older TNCSs, this is an optional feature. An IMV MUST use dynamic function binding (where present) to determine whether a TNCS supports this feature.

The format of the reason string is not defined. It is simply a UTF-8 string. This provides maximum flexibility to the IMV in creating the reason string. However, it is suggested that the reason string be short but informative. When creating reason strings, remember that the user may not have network access. Some IMVs may allow the system administrator to configure the reason string. Others may provide the reason string themselves.

The TNCC and TNCS MAY choose to log the reason string, ignore it, display it to the endpoint user, combine it with other reason strings, or take any other action with it. They MAY modify the reason string such as removing or display control chars or display or truncating long strings.

The IMV SHOULD try to accommodate the language preferences conveyed via the `TNC_TNCS_GetAttribute` function with the AttributeID set to **TNC_ATTRIBUTEID_PREFERRED_LANGUAGE**. This can allow the endpoint user to view the reason string in their native language. Of course, it is understood that every IMV will have limits to the language preferences it can accommodate. Some IMVs will have only one language supported. This is acceptable although not optimal.

2.9.9 Stateless IMVs

A simple IMV (as described in section 8) can avoid maintaining per-IMC state. Such an IMV (known as a “stateless IMV”) might receive two IMC messages in a single handshake (as when two IMCs that send the same message are configured on one TNCC). This would cause the IMV to provide two IMV Action Recommendations for a single handshake, which might confuse the TNCS. A TNCS SHOULD be prepared to receive more than one IMV Action Recommendation from an IMV for a single handshake. The TNCS MAY handle these multiple IMV Action Recommendations in any way: ignoring the first, ignoring the last, combining them, logging a message, refusing access, or anything else.

2.9.10 IMVs with Remote Servers

As an implementation choice, an IMV may consist of a “stub” DLL located on the TNCS host. This stub can talk a vendor-specific protocol to back-end remote servers which implement, for example, integrity verification or policy management functions. A “stub” IMV presents the full IF-IMV interface, and may convert from IF-IMV interface to the vendor specific protocol. In this case, it is of course the responsibility of the IMV vendors to provide the “stub” as well as any remote server. Any redundancy or failover – indeed, all IMV functionality whatsoever – must be provided within the “stub” and its vendor-specific protocol. IMVs also may, of course, be “standalone” and collocated with the TNCS. In either case, the IMV is defined as that entity which speaks IF-IMV to the TNCS, regardless of whether any remote server also exists.

3 IF-IMV Abstract API

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

3.1 Platform and Language Independence

IF-IMV 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 [5]) for ease of comprehension. Because different languages have different conventions and constructs (functions, objects, etc.), the abstract API may need to be modified for different languages in different bindings. However, this should be avoided as much as possible to increase compatibility between IMVs and TNCSs written in different languages.

Section 6 provides a C header file that serves as a binding for the C language with the Microsoft Windows DLL platform binding. The Java Platform Binding in section 4.3 provides a binding for the Java Programming Language. 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 IMVs and TNC Servers on their platform. This specification does not provide a standard way to mix an IMV written in one language with a TNCS written in another language, beyond the support that may be provided by platform-specific bindings.

IF-IMV is also a platform-independent API. It is designed to support almost any platform. Platform-specific bindings are described in section 3.8.5. The IF-IMV 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-IMV 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-IMV API. Future versions may be incompatible due to removing, adding, or changing functions, types, and constants. However, the `TNC_IMV_Initialize` function and its associated types and constants will not change so that version incompatibilities can be detected. A TNCS or IMV can even support multiple versions of the IF-IMV API for maximum compatibility. See section 3.7.1 for details.

3.2.2 Dynamic Function Binding

Platforms that support IF-IMV SHOULD support dynamic function binding. This feature allows a TNCS or IMV 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 TNCS or IMV 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 TNCSs and IMVs that support different version numbers interact safely.

3.2.3 Vendor IDs

The IF-IMV API supports several forms of vendor extensions. IMV or TNCS vendors can define vendor-specific functions and make them available to the other party. IMV or TNCS vendors can define vendor-specific result codes. And IMV 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, 3.4.2.12, and 3.4.2.7.

3.2.4 Vendor-Specific Functions

The IMV and TNC Server MAY extend the IF-IMV API by defining vendor-specific functions that go beyond those described here. An IMV or TNC Server 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_IMV_Reinstall" is clearly intended to be implemented by IMVs.

3.3 Threading and Reentrancy

The TNCS MUST be reentrant (able to receive and process a function call even when one is already underway). IMV DLLs also MUST be reentrant.

The TNC Server and all IMV DLLs MUST be thread-safe. This means that any IF-IMV function can be called at any time even if other threads are also calling an IF-IMV function. The TNCS and IMVs may employ semaphores or other synchronization mechanisms to protect critical sections of code, but these mechanisms SHOULD be employed sparingly using best practices appropriate to the platform to maintain good performance in a highly multi-threaded server environment.

3.4 Data Types

3.4.1 Basic Types

These types are the most basic ones used by the IF-IMV API. They are defined in a platform-dependent and language-dependent manner to meet the requirements described in this section. Consult section 3.8.5 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_IMVID	TNC_UInt32	IMV ID
TNC_ConnectionID	TNC_UInt32	Network Connection ID
TNC_ConnectionState	TNC_UInt32	Network Connection State
TNC_RetryReason	TNC_UInt32	Handshake retry reason
TNC_IMV_Action_Recommendation	TNC_UInt32	IMV Action Recommendation
TNC_IMV_Evaluation_Result	TNC_UInt32	IMV Evaluation Result
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-IMV API version number
TNC_Result	TNC_UInt32	Result code
TNC_AttributeID	TNC_UInt32	Attribute ID

3.4.2.1 IMV ID

When a TNC Server loads an IMV, it assigns it an IMV ID (represented by the TNC_IMVID type). This allows the IMV to identify itself when calling TNCS functions. The IMV ID is a TNC_UInt32 chosen by the TNCS and passed to the TNC_IMV_Initialize function. It is valid until the TNCS calls TNC_IMV_Terminate for this IMV.

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

3.4.2.2 Network Connection ID

A TNCS will commonly be negotiating with several different TNCCs at once (when several endpoints are simultaneously conducting Integrity Check Handshakes). Each of these TNCC-TNCS pairs is referred to as a “network connection”.

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

The IMV and TNCS 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 IMV messages are sent to the right TNCC and IMCs, helps ensure that the IMV Action Recommendation is associated with the right endpoint, and helps the IMV match up messages from IMCs with any state the IMV may be maintaining from earlier parts of that IMC-IMV conversation (even extending across multiple Integrity Check Handshakes in a single network connection).

The TNCS notifies IMVs of changes in network connection state (handshake success, handshake failure, etc.) by calling the `TNC_IMV_NotifyConnectionChange` function. When a network connection is finished, the TNCS first notifies IMVs of this by calling the `TNC_IMV_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 2.9.3 above, it is sometimes desirable to retry an Integrity Check Handshake (when remediation is complete, for instance). Some TNCSs will not support this but all IMVs MUST do so. To indicate that a network connection retry is beginning, a TNCS notifies the IMVs by calling the `TNC_IMV_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 IMV can ask the TNCS to retry an Integrity Check Handshake by calling the `TNC_TNCS_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 TNCS can choose any other value for a network connection ID that does not conflict with another valid network connection ID for the same TNCS-IMV 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 IMV MUST NOT attach any significance to the value chosen. The network connection ID chosen by a TNCS for a particular network connection need not match the network connection ID chosen by the TNCC for that same connection. This is a local identifier only used between the TNCS and the IMVs.

3.4.2.3 Network Connection State

The TNCS uses the `TNC_IMV_NotifyConnectionChange` function to notify IMVs of changes in network connection state. The network connection state is represented as a `TNC_UInt32`. The TNCS MUST pass one of the values listed in section 3.5.5. The TNCS MUST NOT use any other network connection state value with this version of the IF-IMV API.

3.4.2.4 Handshake Retry Reason

The IMV can ask the TNCS to retry an Integrity Check Handshake by calling the `TNC_TNCS_RequestHandshakeRetry` function. One of the parameters to that function is a `TNC_RetryReason`. This type is represented as a `TNC_UInt32`. The IMV MUST pass one of the values listed in section 3.5.6. The IMV MUST NOT use any other handshake retry reason value with this version of the IF-IMV API.

3.4.2.5 IMV Action Recommendation

After evaluating the endpoint's integrity, each IMV supplies an IMV Action Recommendation and IMV Evaluation Result to the TNCS by calling the `TNC_TNCS_ProvideRecommendation` function. One call to TNC_TNCS_ProvideRecommendation suffices to pass both of these values. The type used to communicate the IMV Action Recommendation is `TNC_IMV_Action_Recommendation`. This type is represented as a `TNC_UInt32`. The IMV MUST pass one of the values listed in section 3.5.7. The IMV MUST NOT use any other IMV Action Recommendation value with this version of the IF-IMV API.

3.4.2.6 IMV Evaluation Result

After evaluating the endpoint's integrity, each IMV supplies an IMV Action Recommendation and IMV Evaluation Result to the TNCS by calling the `TNC_TNCS_ProvideRecommendation` function. One call to TNC_TNCS_ProvideRecommendation suffices to pass both of these values. The type used to communicate the IMV Evaluation Result is `TNC_IMV_Evaluation_Result`. This type is represented as a `TNC_UInt32`. The IMV MUST pass one of the values listed in section 3.5.8. The IMV MUST NOT use any other IMV Evaluation Result value with this version of the IF-IMV API. This document does not specify what the TNCS does with this value. It may log it.

3.4.2.7 Message Type

As described in section 2.9.4, 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-IMV 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 IMV 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 2.9.4).

3.4.2.8 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.

3.4.2.9 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. IMVs may request messages with this vendor ID to indicate that they want to receive messages whose message type includes any vendor ID. However, an IMV MUST NOT send messages whose message type includes this reserved value and a TNCS MUST NOT deliver such messages.

3.4.2.10 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. IMVs may request messages with this message subtype to indicate that they want to receive messages whose message subtype has any value. However, an IMV MUST NOT send messages whose message subtype includes this reserved value and a TNCS MUST NOT deliver such messages.

3.4.2.11 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.

3.4.2.12 Result Code

Each function in the IF-IMV 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 3.5.2) have the reserved value zero (0) in the most significant 24 bits.

IMVs and TNCSs 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-IMV API. Any unknown non-zero result code SHOULD be treated as equivalent to TNC_RESULT_OTHER.

3.4.2.13 Attribute ID

The TNC_AttributeID type identifies a TNC attribute. TNC attributes allow IMVs to set and get attribute values identified by a TNC_AttributeID and associated with a TNCS or network connection. For instance, an IMV can get the attribute value with attribute ID TNC_ATTRIBUTE_PREFERRED_LANGUAGE associated with a particular connection, which identifies the preferred language associated with that connection.

As noted above, an attribute ID is represented as a TNC_UInt32, an unsigned integer of at least 32 bits in length. As with result codes and message types, vendor-specific attribute IDs may be defined by particular vendors by placing 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). The vendor who defines a particular vendor-specific attribute ID should carefully document the format of the attribute value for that attribute ID so that IMVs can properly use the attribute ID. Attribute IDs with a vendor ID of zero are reserved for definition by TCG. Some of these reserved attribute IDs are defined in section 3.5.11 below.

Generally, each TNCS will support only a limited set of attribute IDs. TNCSs MAY support no attribute IDs at all. IMVs MUST be prepared for this.

Note that the Java Platform Binding for IF-IMV uses objects instead of byte arrays for attribute values. Section 4.3.9 documents the objects used to represent attribute values for the reserved attribute IDs. Vendors who define vendor-specific attribute IDs SHOULD define what object is used to represent attribute values for those attribute IDs.

3.5 Defined Constants

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

3.5.1 Boolean Values

There are only two permissible values of the type TNC_Boolean: TNC_TRUE and TNC_FALSE. These values are used to indicate Boolean values.

3.5.2 Result Code Values

Each function in the IF-IMV 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-IMV API. IMVs and TNCSs 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 IMV function returns TNC_RESULT_FATAL, then the IMV has encountered a permanent error. The TNCS SHOULD call TNC_IMV_Terminate as soon as possible. The TNCS MAY then

try to reinitialize the IMV with TNC_IMV_Initialize or try other measures such as unloading and reloading the IMV and then reinitializing it.

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

Result Code	Definition
TNC_RESULT_SUCCESS	Function completed successfully
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_ALREADY_INITIALIZED	TNC_IMV_Initialize was called twice without a call to TNC_IMV_Terminate
TNC_RESULT_NO_COMMON_VERSION	No common IF-IMV API version between IMV and TNC Server
TNC_RESULT_CANT_RETRY	TNCS cannot attempt handshake retry
TNC_RESULT_WONT_RETRY	TNCS refuses to attempt handshake retry
TNC_RESULT_INVALID_PARAMETER	Function parameter is not valid
TNC_RESULT_ILLEGAL_OPERATION	Illegal operation attempted
TNC_RESULT_OTHER	Unspecified error
TNC_RESULT_FATAL	Unspecified fatal error

3.5.3 Version Numbers

As noted in section 3.2.1, this specification defines version 1 of the TNC IF-IMV 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_IMV_VERSION_1	The version of IF-IMV API defined here

3.5.4 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_TNCS_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

3.5.5 Network Connection State Values

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

3.5.6 Handshake Retry Reason Values

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

Handshake Retry Reason Value	Definition
TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE	IMV policy has changed. It recommends handshake retry even if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE	IMV policy has changed. It requests handshake retry but not if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_SERIOUS_EVENT	IMV has detected a serious event and recommends handshake retry even if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_MINOR_EVENT	IMV has detected a minor event. It requests handshake retry but not if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_PERIODIC	IMV wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted

3.5.7 IMV Action Recommendation Values

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

IMV Action Recommendation Value	Definition
TNC_IMV_ACTION_RECOMMENDATION_ALLOW	IMV recommends allowing access
TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS	IMV recommends no access
TNC_IMV_ACTION_RECOMMENDATION_ISOLATE	IMV recommends limited access. This access may be expanded after remediation
TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION	IMV does not have a recommendation

3.5.8 IMV Evaluation Result Values

This is the complete set of permissible values for the TNC_IMV_Evaluation_Result type in this version of the IF-IMV API.

IMV Evaluation Result Value	Definition
TNC_IMV_EVALUATION_RESULT_COMPLIANT	AR complies with policy
TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MINOR	AR is not compliant with policy. Non-compliance is minor.
TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MAJOR	AR is not compliant with policy. Non-compliance is major.
TNC_IMV_EVALUATION_RESULT_ERROR	IMV is unable to determine policy compliance due to error
TNC_IMV_EVALUATION_RESULT_DONT_KNOW	IMV does not know whether AR complies with policy

3.5.9 Vendor ID Values

These are reserved vendor ID values. Other vendor IDs between 1 and 16777214 (0xfffffe) may be used as described in section 3.4.2.9. 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

3.5.10 Message Subtype Values

This is a reserved message subtype value. Other message subtypes between 0 and 254 may be used as described in section 3.4.2.10. Note that message subtypes are assigned by vendors as described in section 3.4.2.7.

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

		subtype

3.5.11 Attribute ID Values and Value Definitions

As described in section 3.4.2.13, attribute IDs with a vendor ID of zero are reserved for definition by TCG. Some of these reserved attribute IDs are defined in this section. After the table of reserved attribute IDs, a description of the format of the corresponding attribute value is provided.

Attribute ID Value	Value	Meaning
TNC_ATTRIBUTEID_PREFERRED_LANGUAGE	0x00000001	Preferred human-readable language(s)
TNC_ATTRIBUTEID_REASON_STRING	0x00000002	Reason for IMV Recommendation
TNC_ATTRIBUTEID_REASON_LANGUAGE	0x00000003	Language for IMV reason

3.5.11.1 Preferred Language Attribute

The Preferred Language attribute indicates which human-readable language(s) are preferred for a particular connection or for a TNCS as a whole. An IMV may get the value of the Preferred Language attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. If the IMV provides a connection ID to `TNC_TNCS_GetAttribute`, the attribute value returned by the TNCS will pertain to that connection. If the IMV provides `TNC_CONNECTIONID_ANY`, the attribute value will pertain to the TNCS as a whole. A TNCS may support only one of these options.

The attribute value for the Preferred Language attribute is a NUL-terminated UTF-8 string containing an Accept-Language header as defined in IETF RFC 3282 [3] (US-ASCII only, no control characters allowed). This header lists the languages preferred for human-readable messages. The TNCS may obtain information about language preferences through IF-TNCCS or in some other manner. If no language preference information is available, a zero length string is used (although the string actually contains one byte, the NUL terminator). An IMV must be able to handle this case, which may be common if a TNCS supports this function but the TNCC does not provide language preference information. Note that the byte length of the Preferred Language attribute always includes the NUL terminator. In fact, it includes every byte in the buffer.

TNCSs are not required to implement the Preferred Language attribute but they SHOULD do so if possible since this feature helps with internationalization and results in a better user experience. Likewise, IMVs SHOULD make use of this function when available but are not required to do so. Many TNCSs do not support this attribute. IMVs MUST work properly if a TNCS does not support it.

3.5.11.2 Reason String Attribute

The Reason String attribute allows an IMV to deliver to the TNCS a reason string explaining its IMV Action Recommendation and IMV Evaluation Result. The TNCS MAY pass the reason string to the TNC Client via IF-TNCCS and either the TNCS or the TNCC MAY log the string, modify it, ignore it, combine it with other strings, or take another action with it (as long as they meet the requirements of this specification).

An IMV may set the value of the Reason String attribute with `TNC_TNCS_SetAttribute` but may not get this value with `TNC_TNCS_GetAttribute`. The IMV MUST provide a valid connection ID when setting this attribute.

The attribute value for the Reason String attribute is a NUL-terminated UTF-8 string which explains the reason for the IMV's IMV Action Recommendation and IMV Evaluation Result. In constructing the reason string, the IMV SHOULD try to accommodate the language preferences conveyed via the `TNC_TNCS_GetAttribute` function with AttributeID set to **TNC_ATTRIBUTEID_REASON_LANGUAGE**. No standard format for the reason string has been

defined. Any format is permissible and MUST be accommodated by the TNCS. However, the TNCS or TNCC MAY modify or ignore the reason string, as noted above.

A zero length string is permissible, although it is generally not necessary since this is semantically equivalent to not setting a value for the Reason String attribute. Note that a zero length reason string actually contains one byte, the NUL terminator. Thus the value for this attribute MUST never have a byte length of zero.

Many TNCSs do not support this attribute. IMVs MUST work properly if a TNCS does not support it.

An IMV that does want to provide a reason string SHOULD do so before providing an IMV Action Recommendation since the TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation.

3.5.11.3 Reason Language Attribute

The Reason Language attribute allows an IMV to indicate to the TNCS which language or languages were used in a reason string. The TNCS MAY pass this language information along with reason string to the TNC Client via IF-TNCCS and either the TNCS or the TNCC MAY log this information, modify it, ignore it, combine it with other language information, or take another action with it (as long as they meet the requirements of this specification).

An IMV may set the value of the Reason Language attribute with `TNC_TNCS_SetAttribute` but may not get this value with `TNC_TNCS_GetAttribute`. The IMV MUST provide a valid connection ID when setting this attribute.

The attribute value for the Reason Language attribute is a NUL-terminated UTF-8 string containing an RFC 3066 language tag. A zero length reason language string (one with only a NUL terminator) is permissible, although it is generally not necessary since this is semantically equivalent to not setting a value for the Reason Language attribute. Note that a zero length reason language string actually contains one byte, the NUL terminator. Thus the value for this attribute MUST never have a byte length of zero.

Many TNCSs do not support this attribute. IMVs MUST work properly if a TNCS does not support it. Likewise, TNCSs MUST work properly if an IMV does not set this attribute.

An IMV that does want to provide a reason language string SHOULD do so before providing an IMV Action Recommendation since the TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation.

3.6 Mandatory and Optional Functions

Some of the functions in the IF-IMV API are marked as mandatory below. Mandatory functions MUST be implemented. The rest are marked as optional and need not be implemented. An IMV or TNC Server 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 IMV Functions

These functions are implemented by the IMV and called by the TNC Server.

3.7.1 TNC_IMV_Initialize (MANDATORY)

```
TNC_Result TNC_IMV_Initialize(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_Version minVersion,
```

```
      /*in*/  TNC_Version maxVersion,
      /*out*/ TNC_Version *pOutActualVersion);
```

Description:

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

The TNC Server MUST NOT call any other IF-IMV API functions for an IMV until it has successfully completed a call to `TNC_IMV_Initialize`(). Once a call to this function has completed successfully, this function MUST NOT be called again for a particular IMV-TNCS pair until a call to `TNC_IMV_Terminate` has completed successfully.

The TNC Server MUST set `minVersion` to the minimum IF-IMV API version number that it supports and MUST set `maxVersion` to the maximum API version number that it supports. The TNC Server also MUST set `pOutActualVersion` so that the IMV 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 IMV MUST check these to determine whether there is an API version number that it supports in this range. If not, the IMV MUST return `TNC_RESULT_NO_COMMON_VERSION`. Otherwise, the IMV SHOULD select a mutually supported version number, store that version number at `pOutActualVersion`, and initialize the IMV. If the initialization completes successfully, the IMV SHOULD return `TNC_RESULT_SUCCESS`. Otherwise, it SHOULD return another result code.

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

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`minVersion`	Minimum API version supported by TNCS
`maxVersion`	Maximum API version supported by TNCS

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 IMV and TNC Server
`TNC_RESULT_ALREADY_INITIALIZED`	`TNC_IMV_Initialize` has already been called and `TNC_IMV_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_IMV_NotifyConnectionChange (OPTIONAL)

```
TNC_Result TNC_IMV_NotifyConnectionChange(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID
      /*in*/  TNC_ConnectionState newState);
```

Description:

The TNC Server calls this function to inform the IMV that the state of the network connection identified by `connectionID` has changed to `newState`. Section 3.5.5 lists all the possible values of `newState` for this version of the IF-IMV API. The TNCS MUST NOT use any other values with this version of IF-IMV.

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

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

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 IMV SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection ID to the TNC Server after this function returns (unless the TNCS later creates another network connection with the same network connection ID).

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks. In the `newState` parameter, the TNCS MUST pass one of the values listed in section 3.5.5.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`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_IMV_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_IMV_ReceiveMessage (OPTIONAL)

```
TNC_Result TNC_IMV_ReceiveMessage(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

The TNC Server calls this function to deliver a message to the IMV. 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 IMC (or a TNCC or other party acting as an IMC).

The IMV 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 IMV functions, the IMV SHOULD NOT wait a long time before returning from `TNC_IMV_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 IMV should implement this function if it wants to receive messages. Most IMVs will do so, since they will base their IMV Action Recommendations on measurements received from the IMC. However, some IMVs may base their IMV Action Recommendations on other data such as reports from intrusion detection systems or scanners. Those IMVs need not implement this function.

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMV_ReceiveMessage` has returned. If the IMV wants to retain the message, it should copy it before returning from `TNC_IMV_ReceiveMessage`.

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID. In the `message` parameter, the TNCS MUST pass a reference to a buffer containing the message being delivered to the IMV. In the `messageLength` parameter, the TNCS MUST pass 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. In the `messageType` parameter, the TNCS MUST pass the type of the message. This value MUST match one of the `TNC_MessageType` values previously supplied by the IMV to the TNCS in the IMV's most recent call to `TNC_TNCS_ReportMessageTypes`. IMVs MAY check these parameters to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`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_IMV_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_IMV_SolicitRecommendation (MANDATORY)

```
TNC_Result TNC_IMV_SolicitRecommendation(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Server calls this function at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered) to solicit recommendations from IMVs that have not yet provided a recommendation. The TNCS SHOULD NOT call this method for an IMV and a particular connection if that IMV has already called TNC_TNCS_ProvideRecommendation with that connection since the TNCS last called TNC_IMV_NotifyConnectionChange for that IMV and connection. If an IMV is not able to provide a recommendation at this time, it SHOULD call TNC_TNCS_ProvideRecommendation with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. If an IMV returns from this function without calling TNC_TNCS_ProvideRecommendation, the TNCS MAY consider the IMV's Action Recommendation to be TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. The TNCS MAY take other actions, such as logging this IMV behavior, which is erroneous.

All IMVs MUST implement this function.

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 TNCS will return TNC_RESULT_ILLEGAL_OPERATION from TNC_TNCS_SendMessage and call TNC_IMV_SolicitRecommendation to elicit IMV Action Recommendations based on the data they have gathered so far.

In the imvID parameter, the TNCS MUST pass the IMV ID value provided to TNC_IMV_Initialize. In the connectionID parameter, the TNCS MUST pass a valid network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID for which a recommendation is requested

Result Code	Condition
TNC_RESULT_SUCCESS	Success

TNC_RESULT_NOT_INITIALIZED	TNC_IMV_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_IMV_BatchEnding (OPTIONAL)

```
TNC_Result TNC_IMV_BatchEnding(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Server calls this function to notify IMVs that all IMC messages received in a batch have been delivered and this is the IMV's last chance to send a message in the batch of IMV messages currently being collected.. An IMV MAY implement this function if it wants to perform some actions after all the IMC messages received during a batch have been delivered (using TNC_IMV_ReceiveMessage). For instance, if an IMV has not received any messages from an IMC it may conclude that its IMC is not installed on the endpoint and may decide to call TNC_TNCS_ProvideRecommendation with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS.

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

In the imvID parameter, the TNCS MUST pass the IMV ID value provided to TNC_IMV_Initialize. In the connectionID parameter, the TNCS MUST pass a valid network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID for which a batch is ending

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_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_IMV_Terminate (OPTIONAL)

```
TNC_Result TNC_IMV_Terminate(
      /*in*/  TNC_IMVID imvID);
```

Description:

The TNC Server calls this function to close down the IMV. For example, this function will typically be called when all work is complete and the TNCS is preparing to shut down or when the IMV reports `TNC_RESULT_FATAL`. Once a call to `TNC_IMV_Terminate` is made, the TNC Server MUST NOT call the IMV except to call `TNC_IMV_Initialize` (which may not succeed if the IMV cannot reinitialize itself). Even if the IMV returns an error from this function, the TNC Server MAY continue with its unload or shutdown procedure.

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. IMVs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_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 Server Functions

These functions are implemented by the TNC Server and called by the IMV.

3.8.1 TNC_TNCS_ReportMessageTypes (MANDATORY)

```
TNC_Result TNC_TNCS_ReportMessageTypes(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_MessageTypeList supportedTypes,
      /*in*/  TNC_UInt32 typeCount);
```

Description:

An IMV calls this function to inform a TNCS about the set of message types the IMV is able to receive. Often, the IMV will call this function from `TNC_IMV_Initialize`. With the Windows DLL binding or UNIX/Linux Dynamic Linkage binding, `TNC_TNCS_ReportMessageTypes` will typically be called from `TNC_IMV_ProvideBindFunction` since an IMV cannot call the TNCS with those platform bindings until `TNC_IMV_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. In the imvID parameter, the IMV MUST pass the value provided to TNC_IMV_Initialize. TNCSs MAY check if imvID matches the value previously passed to TNC_IMV_Initialize and return TNC_RESULT_INVALID_PARAMETER if not, but they are not required to make this check.

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

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

If an IMV 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 IMV 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 IMV calls TNC_TNCS_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
imvID	IMV ID assigned by TNCS
supportedTypes	Reference to list of message types supported by IMV
typeCount	Number of message types supported by IMV

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_TNCS_SendMessage (MANDATORY)

```
TNC_Result TNC_TNCS_SendMessage(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

An IMV calls this function to give a message to the TNCS 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. In the `imvID` parameter, the IMV MUST pass the value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the IMV MUST pass a valid network connection ID. TNCSs MAY check these values to make sure they are valid and return an error if not, but TNCSs are not required to make these checks.

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

The IMV MUST NOT call this function unless it has received a call to `TNC_IMV_ReceiveMessage` or `TNC_IMV_BatchEnding` for this connection and the IMV has not yet returned from that function. If the IMV violates this prohibition, the TNCS SHOULD return `TNC_RESULT_ILLEGAL_OPERATION`. If an IMV really wants to communicate with an IMC at another time, it should call `TNC_TNCS_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 TNCS will return `TNC_RESULT_ILLEGAL_OPERATION` from `TNC_TNCS_SendMessage` and call `TNC_IMV_SolicitRecommendation` to elicit IMV Action Recommendations based on the data they have gathered so far.

The TNC Server MUST support any message type. However, the IMV 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 IMV violates this prohibition, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`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_TNCS_RequestHandshakeRetry (MANDATORY)

```
TNC_Result TNC_TNCS_RequestHandshakeRetry(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_RetryReason reason);
```

Description:

An IMV calls this function to ask a TNCS to retry an Integrity Check Handshake. The IMV MUST pass its IMV ID as the `imvID` parameter, a network connection ID as the `connectionID` parameter, and one of the handshake retry reasons listed in section 3.5.6 as the `reason` parameter. If the network connection ID is `TNC_CONNECTIONID_ANY`, then the IMV requests an Integrity Check Handshake retry on all current network connections.

TNCSs MAY check the parameters to make sure they are valid and return an error if not, but TNCSs are not required to make these checks. The `reason` parameter explains why the IMV is requesting a handshake retry. The TNCS MAY use this in deciding whether to attempt the handshake retry. As noted in section 2.9.3, TNCSs are not required to honor IMV requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMV MAY call this function at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMV suddenly gets important information but has already finished its dialog with the IMC, for instance. As always, the TNCS is not required to honor the request for handshake retry.

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

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID for which handshake retry is requested
`reason`	Reason why handshake retry is requested

Result Code	Condition
`TNC_RESULT_SUCCESS`	TNCS intends to retry the handshake
`TNC_RESULT_CANT_RETRY`	TNCS cannot attempt the handshake retry
`TNC_RESULT_WONT_RETRY`	TNCS 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

3.8.4 TNC_TNCS_ProvideRecommendation (MANDATORY)

```
TNC_Result TNC_TNCS_ProvideRecommendation(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_IMV_Action_Recommendation recommendation,
      /*in*/  TNC_IMV_Evaluation_Result evaluation);
```

Description:

An IMV calls this function to deliver its IMV Action Recommendation and IMV Evaluation Result to the TNCS. The TNCS SHOULD use the `recommendation` value in determining its own TNCS Action Recommendation or decision about endpoint access. The TNC specifications do not specify how the TNCS does the `recommendation` value but it is certainly essential to have a recommendation from the IMV. The TNC specifications also do not specify what the TNCS does with the `evaluation` value. It may log it.

The IMV MUST pass its IMV ID as the `imvID` parameter, a valid network connection ID as the `connectionID` parameter, one of the IMV Action Recommendation values listed in section 3.5.7 as the `recommendation` parameter, and one of the IMV Evaluation Result values listed in section 3.5.8 as the `evaluation` parameter. TNCSs MAY check these values to make sure they are valid and return an error if not, but TNCSs are not required to make these checks.

The IMV should deliver its IMV Action Recommendation as soon as possible so that the TNCS can proceed with determining its own TNCS Action Recommendation. If the IMV receives a message from an IMC and is able to decide on an IMV Action Recommendation and deliver it to the TNCS before returning from `TNC_IMV_ReceiveMessage`, it SHOULD do so. However, as always the IMV SHOULD return promptly to avoid a long delay that might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

An IMV SHOULD NOT expect that it will be able to send IMC-IMV messages after calling `TNC_TNCS_ProvideRecommendation`. The TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation. For instance, IMVs SHOULD send remediation instructions before calling `TNC_TNCS_ProvideRecommendation`.

However, a TNCS MAY continue to deliver messages after an IMV calls `TNC_TNCS_ProvideRecommendation`, especially if other IMVs continue the dialog after the one IMV has rendered its decision. The IMV MUST be prepared for this. It MAY simply ignore these late messages or it MAY consider them and even change its recommendation by calling `TNC_TNCS_ProvideRecommendation` again. In this case, the TNCS SHOULD use the last recommendation received from an IMV during a particular handshake. However, the TNCS is not required to do this.

If an IMV does not provide a recommendation earlier, the TNCS will call `TNC_IMV_SolicitRecommendation` at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered). The IMV SHOULD then call `TNC_TNCS_ProvideRecommendation` to deliver its recommendation. If the IMV calls this function when there is no active handshake on the specified network connection, the TNCS SHOULD return `TNC_RESULT_ILLEGAL_OPERATION`. If an IMV really needs to communicate a recommendation at another time, it should call `TNC_TNCS_RequestHandshakeRetry`.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID for which recommendation is being supplied
`recommendation`	IMV's Action Recommendation
`evaluation`	IMV Evaluation Result

Result Code	Condition
TNC_RESULT_SUCCESS	TNCS intends to retry the handshake
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_ILLEGAL_OPERATION	Recommendation provided at an illegal time
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.5 TNC_TNCS_GetAttribute (OPTIONAL)

```
TNC_Result TNC_TNCS_GetAttribute(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_AttributeID attributeID,
      /*in*/  TNC_UInt32 bufferLength,
      /*out*/ TNC_BufferReference buffer,
      /*out*/ TNC_UInt32 *pOutValueLength);
```

Description:

An IMV calls this function to get the value of an attribute associated with a connection or with the TNCS as a whole. This function is optional. The TNCS is not required to implement it. Since this function was not included in IF-IMV 1.0, many TNCSs do not support it. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass its IMV ID as the `imvID` parameter, a standard or vendor-specific attribute ID as the `attributeID` parameter, and a valid network connection ID as the `connectionID` parameter. If the IMV passes a valid connection ID, the TNCS SHOULD provide the attribute value for the specified connection. If the IMV passes `TNC_CONNECTIONID_ANY`, the TNCS SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS does not recognize the attribute ID or connection ID, it SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS recognizes the attribute ID and connection ID but does not have an attribute value for the requested attribute ID and connection ID, it SHOULD also return `TNC_RESULT_INVALID_PARAMETER`.

The IMV MUST set `pOutValueLength` so that the TNCS can use it as an output parameter to provide the length in bytes of the requested attribute value. With the C binding, this would involve setting `pOutValueLength` to point to a suitable storage location.

If the TNCS returns a result code other than `TNC_RESULT_SUCCESS`, it MUST not store any values via the supplied parameters. But if it returns `TNC_RESULT_SUCCESS`, it MUST provide the length in bytes of the requested attribute value. This length is stored in the manner indicated by the `pOutValueLength` parameter (through a pointer, for the C binding).

If the IMV passes 0 for the `bufferLength` parameter, the TNCS ignores the value of the `buffer` parameter. If the IMV passes a non-zero value for the `bufferLength` parameter, the IMV MUST set the `buffer` parameter so that the TNCS can use it as an output parameter to provide the requested attribute value. The IMV MUST provide enough storage for at least `bufferLength` bytes to be stored via the `buffer` parameter.

The TNCS MUST check the length of the requested attribute value before storing anything via the `buffer` parameter. If the length of the requested attribute value is greater than the `bufferLength` parameter, the TNCS MUST NOT store any data via the `buffer` parameter. Instead, it MUST simply store the length via the `pOutValueLength` parameter. This allows the IMV to recognize that more storage space is needed. In either case, a result code of `TNC_RESULT_SUCCESS` SHOULD be returned.

The TNCS MUST NOT modify the values stored in the `buffer` and `pOutValueLength` parameters after returning from this function. It absolutely MUST NOT store more than `bufferLength` bytes via the `buffer` parameter.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID for which an attribute value is desired (or `TNC_CONNECTIONID_ANY` to get information for the TNCS as a whole)
`attributeID`	Attribute ID for which an attribute value is desired
`bufferLength`	Length in bytes of storage referenced by `buffer` parameter (or 0 if no storage referenced)

Output Parameter	Description
`buffer`	Requested attribute value
`pOutValueLength`	Length in bytes of requested attribute value

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_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

3.8.6 TNC_TNCS_SetAttribute (OPTIONAL)

```
TNC_Result TNC_TNCS_SetAttributeValue(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_AttributeID attributeID,
      /*in*/  TNC_UInt32 bufferLength,
      /*in*/  TNC_BufferReference buffer);
```

Description:

An IMV calls this function to set the value of an attribute associated with a connection or with the TNCS as a whole.

This is an optional function so the TNCS is not required to implement it. Since this function was not included in IF-IMV 1.0, many TNCSs do not implement it. IMVs MUST work properly if a TNCS does not implement this function. The IMV is never required to call this function. The TNCS MUST work with IMVs that don't call this function.The IMV SHOULD use dynamic function binding (on platforms where that is available) to determine whether the TNCS implements this function.

The IMV MUST pass its IMV ID as the `imvID` parameter, a standard or vendor-specific attribute ID as the `attributeID` parameter, and a valid network connection ID as the `connectionID` parameter. If the IMV passes a valid connection ID, the TNCS SHOULD set the attribute value for the specified connection. If the IMV passes `TNC_CONNECTIONID_ANY`, the TNCS SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS does not recognize the attribute ID or connection ID, it SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS recognizes the attribute ID and connection ID but does not support setting an attribute value for the requested attribute ID and connection ID, the TNCS SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code.

The IMV MUST pass an attribute value in the buffer referenced by the `buffer` parameter. This attribute value SHOULD have the exact format specified in the description of the attribute ID (in section 3.5.11 for attributes defined there or in vendor-specific documentation for a vendor-specific attribute ID). The length of the attribute value MUST be exactly the number of octets (bytes) indicated by the `bufferLength` parameter. The TNCS MUST NOT modify the contents of the buffer or even access them after this function returns. Therefore, the TNCS SHOULD copy the attribute value to other storage before returning from this function.

The IMV MAY pass 0 for the `bufferLength` parameter. In this case, the IMV MUST pass NULL for the `buffer` parameter. This indicates a zero-length value for the specified attribute value.

If the IMV passes an attribute value that is not valid for the specified attribute, the TNCS MAY return the `TNC_RESULT_INVALID_PARAMETER` result code to indicate that the attribute value is not valid. The TNCS MAY also forgo checking the validity of the attribute value and return the `TNC_RESULT_SUCCESS` result code but later ignore the attribute value that has been set.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID for which an attribute value is to be set (or `TNC_CONNECTIONID_ANY` to set an attribute value for the TNCS as a whole)
`attributeID`	Attribute ID for attribute to be set
`bufferLength`	Length in bytes of attribute value
`buffer`	Attribute value to set

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_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 Platform Bindings

As noted above, IF-IMV 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-IMV SHOULD be implemented on specific platforms.

Note that if taking the "stub" approach, then IF-IMV represents an API between a TNCS and an IMV stub DLL on the same platform, although the "actual" IMV may be located remotely on a different platform.

It is assumed that platform bindings will only be created for platforms which are appropriate to a role as servers. For example, IF-IMV bindings for handheld and 16-bit consumer operating systems will not be specified as it is assumed that there will be limited (if any) implementation of TNCSs on such systems.

4.1 Microsoft Windows DLL Platform Binding

Microsoft Windows is a popular platform with many variations. This binding is intended to support only 32- or 64-bit Windows versions (e.g., Windows NT, Windows 2000, Windows 2003, or Windows XP). It is not intended to support 16-bit Windows (Windows 3.X and Windows for Workgroups), nor is it directly intended to support Windows CE, Windows 95/98/Me, or other such versions of Windows.

Implementations on one of these platforms SHOULD use this binding when possible for maximum compatibility with other IMVs or TNC Servers 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 IMVs

The loading of IMVs is parallel to the process for loading IMCs within IF-IMC, with only minor differences in behavior. With the Microsoft Windows DLL platform binding, each IMV is implemented as a DLL. This IMV DLL may be either a "stub" IMV DLL or a full IMV; this distinction is immaterial to the operation of the API.

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 (defined in section 4.1.9) that can only be changed by privileged users. The TNC Server gets the value of this key and loads the IMVs using the `LoadLibrary` system call. Then it uses the `GetProcAddress` function call to access the IMV's functions, as described in section 4.1.2. The TNCS MUST always call the `TNC_IMV_Initialize` function first. When it is done using an IMV, the TNC Server calls `TNC_IMV_Terminate` and then unloads the IMV DLL using the `FreeLibrary` system call. The TNCS **MUST** listen for changes to the well-known registry key so that it can load and unload IMVs dynamically. However, the TNCS **SHOULD** delay before making changes based on registry key changes since it is common for these changes to come in batches within a few seconds during an install process. Unlike a TNCC, a TNCS **MUST NOT** ignore such changes.

4.1.2 Dynamic Function Binding

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

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

IMV and TNCS 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 IMV and TNCS functions exposed through this API.

4.1.3 Threading

Unlike IMCs, IMV DLLs are required to be thread-safe. The IMV DLL MAY create threads. The TNC Server MUST be thread-safe. This allows the IMV DLL to do work in background threads and call the TNC Server when a recommendation is ready (for instance).

4.1.4 Platform-Specific Bindings for Basic Types

With the Microsoft Windows DLL platform binding, the basic data types defined in the IF-IMV 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-IMV 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-IMV 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_TNCS_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMV_InitializePointer)(
    TNC_IMVID imvID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);

typedef TNC_Result (*TNC_IMV_NotifyConnectionChangePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionStatus newStatus);

typedef TNC_Result (*TNC_IMV_ReceiveMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
```

```
typedef TNC_Result (*TNC_IMV_SolicitRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_BatchEndingPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_TerminatePointer)(
    TNC_IMVID imvID);

typedef TNC_Result (*TNC_TNCS_ReportMessageTypesPointer)(
    TNC_IMVID imvID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCS_SendMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_TNCS_RequestHandshakeRetryPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCS_ProvideRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_IMV_Action_Recommendation recommendation);

typedef TNC_Result (*TNC_TNCS_GetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);

typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);

typedef TNC_Result (*TNC_TNCS_BindFunctionPointer)(
    TNC_IMVID imvID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMV_ProvideBindFunctionPointer)(
    TNC_IMVID imvID,
    TNC_TNCS_BindFunctionPointer bindFunction);
```

4.1.7 Platform-Specific IMV Functions

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

4.1.7.1 TNC_IMV_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMV_ProvideBindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_TNCS_BindFunctionPointer bindFunction);
```

Description:

IMVs implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. The TNC Server MUST call the function immediately after calling `TNC_IMV_Initialize` to provide a pointer to the TNCS bind function. The IMV can then use the TNCS bind function to obtain pointers to any other TNCS functions.

In the `imvID` parameter, the TNCS MUST pass the value provided to `TNC_IMV_Initialize`. In the `bindFunction` parameter, the TNCS MUST pass a pointer to the TNCS bind function. IMVs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`bindFunction`	Pointer to `TNC_TNCS_BindFunction`

Error Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_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 error codes	Other non-fatal error

4.1.8 Platform-Specific TNC Server Functions

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

4.1.8.1 TNC_TNCS_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCS_BindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Servers implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. An IMV can use this function to obtain pointers to other TNCS functions. To obtain a pointer to a TNCS function, an IMV calls `TNC_TNCS_BindFunction`. The IMV obtains a pointer to `TNC_TNCS_BindFunction` from `TNC_IMV_ProvideBindFunction`.

The IMV MUST set the `imvID` parameter to the IMV ID value provided to `TNC_IMV_Initialize`. TNCSs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check. The IMV MUST set the f`unctionName` parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. "`TNC_TNCS_SendMessage`"). The IMV MUST set the `pOutFunctionPointer` parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCS 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 IMV obtains a pointer to a particular function, the TNCS MUST always return the same function pointer value to that IMV for that function name. This requirement does not apply across IMV termination and reinitialization.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`functionName`	Name of function whose pointer is requested

Output Parameter	Description
`pOutFunctionPointer`	Requested function pointer

Error 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 error codes	Other non-fatal error

4.1.9 Well-known Registry Key

As discussed above, a well-known registry key is used by the TNCS to load IMVs. For Windows platforms, this key is defined within the HKEY_LOCAL_MACHINE hive as follows. (The HKLM hive is used since there will often be no logged on user to give context for any other hive.)

- HKEY_LOCAL_MACHINE
  → Software
  → Trusted Computing Group
  → TNC
  → IMVs
  → [Human readable name of IMV], 0..n

Each IMV 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 IMV DLL to be loaded.
- the optional value "*Description"* is a REG_SZ String which contains a vendor-specific human-readable description of the IMV DLL

The name and description are for ease of administration and may be ignored by the TNCS, 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 Servers and IMVs 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 Server vendor with vendor ID 2 could specify that any IMV can populate its key at install time with a value named 2_SupportPhone and that vendor's TNC Server can read this value and display it in the TNC Server's status panel next to the IMV name. The only requirement, as stated above, is that TNC Servers and IMVs MUST ignore unrecognized values and keys.

4.2 UNIX/Linux Dynamic Linkage Platform 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 IMVs and TNC Servers 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 IMVs

With the UNIX/Linux Dynamic Linkage platform binding, each IMV is implemented as a dynamically loaded executable file (also known as a shared object or DLL). When the IMV 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 Server 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 IMV to be loaded, the TNC Server 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 Server uses the `dlsym` function call to access the IMV's functions, as described in section 4.2.2. The TNCS MUST always call the `TNC_IMV_Initialize` function first. When it is done using an IMV, the TNC Server calls `TNC_IMV_Terminate` and then unloads the IMV executable file using the `dlclose` system call.

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

4.2.2 Dynamic Function Binding

The UNIX/Linux Dynamic Linkage platform binding does support dynamic function binding. To determine whether an IMV function is defined, a TNC Server will pass the function name to

`dlsym`. If the result is `NULL`, the function is not defined. Otherwise, the function is defined and the TNCS can call it using the function pointer returned. This is common practice on UNIX and Linux.

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

IMV0 and TNCS 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 IMV and TNCS functions exposed through this API.

4.2.3 Format of `/etc/tnc_config`

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

The `/etc/tnc_config` file is a UTF-8 file. However, TNCSs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCS encounters a character that is not US-ASCII and the TNCC can not process UTF-8 properly, the TNCS SHOULD indicate an error and not load the file at all. In fact, the TNCS 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 “IMV ” (U+0049, U+004D, U+0056, U+0020) specifies an IMV that may be loaded. The next character MUST be U+0022 (QUOTATION MARK). This MUST be followed by a human-readable IMV name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the IMV 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 IMV executable file (up to but not including the U+000A that terminates the line). The path to the IMV executable file MUST NOT be a partial path.

The `/etc/tnc_config` file must not contain IMVs with the same human-readable name. An IMV that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMV 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 TNCS vendor with vendor ID 2 could specify that any IMV can add a line at install time that begins with 2_SupportPhoneIMV, then the IMV’s human-readable name and the IMV vendor’s support

telephone number. The defining vendor's TNCS (or any other TNCS) can read this phone number and display it in the TNCS's status panel next to the IMV name.

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

A line that does not match the comment, empty, imv, or vendor productions below SHOULD be ignored by a TNCS and IMVs that are using the Linux/UNIX Platform Binding 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 / java-imc / imv / java-imv / vendor /
other) %x0A
comment = %x23 *(%x01-09 / %x0B-22 / %x24-1FFFFF)
empty = ""
imc = %x49.4D.43.20.22 name %x22.20 path
java-imc = %x4a.41.56.41.2d.49.4D.43.20.22 name %x22.20 class %x20 path
imv = %x49.4D.56.20.22 name %x22.20 path
java-imv = %x4a.41.56.41.2d.49.4D.56.20.22 name %x22.20 class %x20 path
name = *(%x01-09 / %x0B-21 / %x23-1FFFFF)
class = *(%x01-09 / %x0B-1F / %x21-1FFFFF)
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
```

Note that lines that match the `imc`, `java-imc`, and `java-imv` productions are ignored for the purposes of the Linux/UNIX Platform Binding for IF-IMV. Note also that the `other` production is only employed if no other production matches a line.

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

```
# Simple TNC config file

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

4.2.4 Threading

Unlike IMC's, IMV executable files are required to be thread-safe. The IMV MAY create threads. The TNC Server MUST be thread-safe. This allows the IMV DLL to do work in background threads and call the TNC Server when messages are ready to send (for instance).

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-IMV 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-IMV 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 DLL platform binding for the IF-IMV 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_TNCS_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMV_InitializePointer)(
    TNC_IMVID imvID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);

typedef TNC_Result (*TNC_IMV_NotifyConnectionChangePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionStatus newStatus);

typedef TNC_Result (*TNC_IMV_ReceiveMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMV_SolicitRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_BatchEndingPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_TerminatePointer)(
    TNC_IMVID imvID);

typedef TNC_Result (*TNC_TNCS_ReportMessageTypesPointer)(
    TNC_IMVID imvID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCS_SendMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
```

```
typedef TNC_Result (*TNC_TNCS_RequestHandshakeRetryPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCS_ProvideRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_IMV_Action_Recommendation recommendation);

typedef TNC_Result (*TNC_TNCS_GetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC AttributeID, attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);

typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);

typedef TNC_Result (*TNC_TNCS_BindFunctionPointer)(
    TNC_IMVID imvID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMV_ProvideBindFunctionPointer)(
    TNC_IMVID imvID,
    TNC_TNCS_BindFunctionPointer bindFunction);
```

4.2.8 Platform-Specific IMV Functions

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

4.2.8.1 TNC_IMV_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMV_ProvideBindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_TNCS_BindFunctionPointer bindFunction);
```

Description:

IMVs implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. The TNC Server MUST call the function immediately after calling `TNC_IMV_Initialize` to provide a pointer to the TNCS bind function. The IMV can then use the TNCS bind function to obtain pointers to any other TNCS functions.

In the `imvID` parameter, the TNCS MUST pass the value provided to `TNC_IMV_Initialize`. In the `bindFunction` parameter, the TNCS MUST pass a pointer to the TNCS bind function. IMVs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description

imvID	IMV ID assigned by TNCS
bindFunction	Pointer to TNC_TNCS_BindFunction

Error Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_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 error codes	Other non-fatal error

4.2.9 Platform-Specific TNC Server Functions

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

4.2.9.1 TNC_TNCS_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCS_BindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Servers implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. An IMV can use this function to obtain pointers to other TNCS functions. To obtain a pointer to a TNCS function, an IMV calls TNC_TNCS_BindFunction. The IMV obtains a pointer to TNC_TNCS_BindFunction from TNC_IMV_ProvideBindFunction.

The IMV MUST set the imvID parameter to the IMV ID value provided to TNC_IMV_Initialize. TNCSs MAY check if imvID matches the value previously passed to TNC_IMV_Initialize and return TNC_RESULT_INVALID_PARAMETER if not, but they are not required to make this check. The IMV MUST set the functionName parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. "TNC_TNCS_SendMessage"). The IMV MUST set the pOutFunctionPointer parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCS 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
imvID	IMV ID assigned by TNCS
functionName	Name of function whose pointer is requested

Output Parameter	Description
pOutFunctionPointer	Requested function pointer

Error 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 error codes	Other non-fatal error

4.3 Java Platform Binding

The Java Platform provides remarkable portability, allowing the same code to run on many operating systems. It also includes features that allow partially trusted code to be loaded with reduced privileges. There is a desire to support IF-IMV on the Java Platform, especially to support loading IMVs with reduced privileges to address security and reliability concerns. Therefore, the Java Platform Binding for IF-IMV has been developed.

At this time, the only versions of the Java Platform that are supported with the Java Platform Binding for IF-IMV are the Java 2 Platform Standard Edition versions 1.4.2 and later. However, other versions may be supported at a later time. Implementations of the IF-IMV specification on the Java 2 Platform SHOULD use this binding when possible for maximum compatibility with other IMCs and TNC Clients on the platform.

4.3.1 Object Orientation

The Java Platform Binding for IF-IMV is designed to take advantage of the Java Platform's support for object orientation. Three Java interfaces have been defined that correspond to the kinds of objects inherent in the IF-IMV Abstract API: `IMV`, `TNCS`, and `IMVConnection`. The functions described in the IF-IMV Abstract API have been mapped to methods in these interfaces. Interfaces were used instead of classes to leave implementers the freedom to use whatever class hierarchy they need or want. An additional `TNCConstants` interface has been defined to contain constant values shared between IF-IMC and IF-IMV.

All IMVs that implement the Java Platform Binding for IF-IMV MUST implement the `IMV` interface. All TNCSs that implement the Java Platform Binding for IF-IMV MUST implement the `TNCS` interface and provide objects that implement the `IMVConnection` interface as needed. The TNCS MUST also define all the interfaces in the IF-IMV API.

4.3.2 Exception Handling

The exception handling capabilities of the Java Platform provide greater robustness than the result code mechanism used by the IF-IMV Abstract API since exceptions must be explicitly ignored while result codes are ignored by default. Therefore, the Java Platform Binding for IF-IMV defines a `TNCException` class that wraps the result codes defined in the IF-IMV Abstract API. This class MUST be defined by all TNCSs.

4.3.3 Limited Privileges

The Java Platform has always included support for running code with limited privileges. With the Java 2 Platform (JDK 1.2 and later), this is implemented with AccessControllers, Permissions, and other components of the Java 2 Platform security architecture. These features are useful for IF-IMV, where it may be desirable to load and run a partially trusted IMV.

The Java Platform Binding for IF-IMV does not define any new Permissions. All IMVs loaded by the TNCS are assumed to be trusted to call any TNCS methods and vice versa. However, this does not mean that the TNCS and IMVs should completely trust each other.

The Java 2 Platform does define many Permissions. Many system methods check to ensure that calling code holds those permissions before allowing access. Since a TNCS may have more or fewer privileges than an IMV if the TNCS and IMV were loaded from different CodeSources, a TNCS and IMV cannot trust each other.

When a TNCS loads an IMV from any external source (one that is not delivered with the TNCS), it MUST use a class loader that will determine and assign the appropriate permissions (such as the URLClassLoader).

When an IMV calls a method of a class included in the TNCS, the TNCS code MUST recognize that the IMC's permissions may be much less than those of the TNCS. The code in the TNCS's called method will run with the intersection of the IMV's permissions and the TNCS's. To perform privileged operations, the TNCS's code MAY use a doPrivileged method to regain its normal permissions and perform privileged actions. Alternatively, the TNCS's code MAY queue data for later processing by code with more permissions. In either case, the TNCS's code MUST check the IMV's request and the arguments supplied very carefully. The IMV's code MUST NOT be trusted unless the TNCS knows that the IMV's privileges are as great as the TNCS's (as when the IMV was loaded from the same CodeSource as the TNCS).

Likewise, when the TNCS calls a method of an IMV, the IMV code MUST recognize that the TNCS's permissions may be much less than those of the IMV. The code in the IMV's called method will run with the intersection of the IMV's permissions and the TNCS's. To perform privileged operations, the IMV's code MAY use a doPrivileged method to regain its normal permissions and perform privileged actions. Alternatively, the IMV's code MAY queue data for later processing by code with more permissions. In either case, the IMV's code MUST check the TNCS's call and the arguments supplied very carefully. The TNCS's code MUST NOT be trusted. The IMV MUST regard IF-M messages as untrusted unless the IMV can authenticate them in some manner or the IMV determines that the TNCS can be trusted enough to perform the operations requested by the IF-M messages. The simplest way to meet this last criterion is for the IMV to only perform operations triggered by IMV messages in its receiveMessage method and to do so without using doPrivileged. This will ensure that the available Permissions are the intersection of the IMV's and TNCS's permissions so the IMV will not accidentally perform any operations that the TNCS is not already trusted to perform.

4.3.4 Finding, Loading, and Unloading IMVs

With the Java platform binding, each IMV is implemented as a jar file. When the IMV is installed and is intended to be usable by any TNCS on the system, its jar file SHOULD be stored in a directory that can be read by any user but can only be modified by privileged users. Then a JAVA-IMV entry SHOULD BE created in the `tnc_config` file giving the full path of the jar file. The privileges of the jar file and the `tnc_config` file should be set so that they can be read by any user but can only be modified by privileged users. See section 4.3.6 for details on the format of the `tnc_config` file.

A TNC Server that wishes to load a Java IMV SHOULD open the `tnc_config` file on the system, read the JAVA-IMV entries in the file, and determine which of them should be loaded (using optional local configuration or any other algorithm). For each IMV to be loaded, the TNC Server SHOULD create a new instance of the IMV class, using the full path of the jar file and the class name for the IMV class as specified in the `tnc_config` file to load the class and call the noargs constructor for that class. When loading an IMV class in this manner, the TNCS MUST use a class loader that will determine and assign the appropriate permissions (such as the URLClassLoader).

The TNCS MUST always call the IMV's `initialize` method first. When it is finished with an IMV, the TNC Server MUST call the IMV's `terminate` method.

As described in the Security Considerations section, loading an IMV into the same JVM as the TNCS can compromise the TNCS and other IMVs if the IMV is later found to be untrustworthy. Also, an unstable IMV can crash the whole TNCS. However, the risk of this is considerably less

with the Java Platform Binding than with the Windows DLL Binding, especially if the Permissions assigned to the IMV are minimal.

4.3.5 Dynamic Function Binding

The Java Platform Binding for IF-IMV does support dynamic function binding. Thus to allow a TNCS or IMV to define methods that go beyond those included in this Abstract API and allow the other party to determine whether the Abstract API optional methods are implemented two techniques are used.

For a TNC Server to determine whether an optional IMV method is implemented the TNC Server should make a call to the method. If the method is not implemented, an `UnsupportedOperationException` is thrown. The same mechanism is employed to handle optional TNCS methods.

Extensions to the IF-IMV API (for new versions of the IF-IMV API and vendor extensions) are handled using interfaces. To define an extension, place the new methods and fields in a new interface. To check whether an IMV or TNCS implements the extension, use the `instanceof` operator. If so, cast the object to the interface type and use the new methods and fields.

For vendor-specific extensions to the IF-IMV API, the name of the new interface must begin with "TNC_XXX_" where XXX is the vendor ID of the vendor defining the extension. This will help avoid name collisions and clarify where the vendor extension came from.

4.3.6 Format of the tnc_config file

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

The `tnc_config` file is a UTF-8 file. However, TNCSs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCS encounters a character that is not US-ASCII and the TNCS can not process UTF-8 properly, the TNCS SHOULD indicate an error and not load the file at all. In fact, the TNCS 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 `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 "JAVA-IMV " (U+004A, U+0041, U+0056, U+0041, U+002D, U+0049, U+004D, U+0043, U+0020) specifies a Java IMV (an IMV that uses the Java Platform Binding) that may be loaded. The next character MUST be U+0022 (QUOTATION MARK). This MUST be followed by a human-readable IMV name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the human-readable IMV 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), the fully qualified class name of the IMV class (which MUST NOT include a space), followed by a space (U+0020). After this space MUST come the full path of the IMV jar file (which runs up to but does not include the U+000A that terminates the line). The path to the IMV jar file MUST NOT be a partial path. For maximum compatibility, the fully qualified class name SHOULD NOT contain any characters that are not US-ASCII characters.

The `tnc_config` file must not contain more than one Java IMV with the same human-readable name. A TNCS that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMV names to make them unique. Identical class names and full paths are permitted but the TNCS MAY ignore entries with identical class names or paths if they will cause problems for it.

An extension mechanism has been defined so that vendors can place vendor-specific data in the `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 TNCS vendor with vendor ID 2 could specify that any IMV can add a line at install time that begins with 2_SupportPhoneIMV, then the IMV's human-readable name and the IMV vendor's support telephone number. The defining vendor's TNCS (or any other TNCS) can read this phone number and display it in the TNCS's status panel next to the IMV name.

TNCSs and IMVs SHOULD ignore unrecognized vendor-specific data. This recommendation is backwards-compatible with the recommendation in IF-IMV 1.0 for TNCSs and IMVs to ignore lines in the `tnc_config` file with unrecognized syntax.

A line that does not match the comment, empty, java-imv, or vendor productions below SHOULD be ignored by the TNCS and IMVs 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 / java-imc / imv / java-imv / vendor /
other) %x0A
comment = %x23 *(%x01-09 / %x0B-22 / %x24-1FFFFF)
empty = ""
imc = %x49.4D.43.20.22 name %x22.20 path
java-imc = %x4a.41.56.41.2d.49.4D.43.20.22 name %x22.20 class %x20 path
imv = %x49.4D.56.20.22 name %x22.20 path
java-imv = %x4a.41.56.41.2d.49.4D.56.20.22 name %x22.20 class %x20 path
name = *(%x01-09 / %x0B-21 / %x23-1FFFFF)
class = *(%x01-09 / %x0B-1F / %x21-1FFFFF)
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
```

Note that lines that match the `imc`, `imv`, and `java-imc` productions are ignored for the purposes of the Java Platform Binding for IF-IMV. Note also that the `other` production is only employed if no other production matches a line.

Here is a sample file specifying one Java IMV named "Example IMV" with a fully qualified class name of com.example.ExampleIIMV and a jar file path of /usr/bin/example_imv.jar.

```
# Simple Java IMV config file

JAVA-IMV "Example IMV" com.example.ExampleIMV /usr/bin/example_imv.jar
```

4.3.7 Location of the tnc_config file

The location of the `tnc_config` file depends on the operating system in use. For Windows operating systems, the file SHOULD go in the C:\WINDOWS directory. For Linux and UNIX and MacOS X operating systems, the file SHOULD go in the /etc directory. This specification does not define a standard location for the `tnc_config` file on other operating systems at this time. IMVs and TNCSs MAY use the os.name property to determine which operating system they are running on and choose the appropriate location for the `tnc_config` file. If the value of the os.name property begins with "Windows", then the operating system is probably a Windows operating system. If not, it's probably a Linux, UNIX, or MacOS X operating system.

If the directory described above does not exist, the IMV or TNCS should not create it. These directories are a basic part of the Windows and Linux/UNIX/MacOS X operating systems. If they do not exist, there is some problem that will probably require administrative intervention.

4.3.8 Threading

With the Java Binding for IF-IMV, IMVs are required to be thread-safe. An IMV MAY create threads. The TNC Server MUST be thread-safe. This allows the IMV DLL to do work in background threads and call the TNC Server when messages are ready to send (for instance).

4.3.9 Attributes

Instead of representing attribute values with a byte array, the Java Platform Binding for IF-IMV uses objects. For each attribute ID defined in section 3.5.11, this section explains the object used by the Java Platform Binding for IF-IMV to represent values for that attribute.

4.3.9.1 Preferred Language Attribute

The Preferred Language attribute indicates which human-readable language(s) are preferred for a particular connection or for a TNCS as a whole.

With the Java Platform Binding for IF-IMV, attribute values for the Preferred Language attribute are represented as a String containing an Accept-Language header as defined in IETF RFC 3282 [3] (US-ASCII only, no control characters allowed). This header lists the languages preferred for human-readable messages. If no language preference information is available, a zero length string is used. The string is not terminated by a NUL character since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.2 Reason String Attribute

The Reason String attribute allows an IMV to deliver to the TNCS a reason string explaining its IMV Action Recommendation and IMV Evaluation Result.

With the Java Platform Binding for IF-IMV, attribute values for the Preferred Language attribute are represented as a String which explains the reason for the IMV's IMV Action Recommendation and IMV Evaluation Result. No standard format for the reason string has been defined. Any format is permissible and MUST be accommodated by the TNCS. The string is not terminated by a NUL character since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.3 Reason Language Attribute

The Reason Language attribute allows an IMV to indicate to the TNCS which language or languages were used in a reason string.

With the Java Platform Binding for IF-IMV, attribute values for the Reason Language attribute are represented as a String containing an RFC 3066 language tag. The string is not terminated by a NUL character since this convention is not employed in the Java programming language. Thus,

the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.10 Platform-Specific Bindings for Basic Types

With the Java Platform Binding, the basic data types defined in the IF-IMV abstract API are mapped as follows:

The `TNC_UInt32` type is mapped to Java's `long` type.

The `TNC_BufferReference` type is mapped to a byte array (`byte []`). The value `NULL` is allowed for a `TNC_BufferReference` only where explicitly permitted in this specification.

Since Java does not have an equivalent of C's `typedef`, the Java types are used in the Java interface definitions.

4.3.11 Platform-Specific Bindings for Derived Types

With the Java Platform Binding, the platform-specific derived data types defined in the IF-IMC abstract API are mapped as follows:

The `TNC_MessageTypeList` type is mapped to an array of longs (`long []`). The value `NULL` is allowed for a `TNC_MessageTypeList` only where explicitly permitted in this specification.

4.3.12 Interface and Class Definitions

Here are interface and class definitions for the Java Platform Binding for the IF-IMV API.

4.3.12.1 TNCException (TNCException.java)

org.trustedcomputinggroup.tnc

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └org.trustedcomputinggroup.tnc.TNCException
```

All Implemented Interfaces:

java.io.Serializable

```
public class TNCException
extends java.lang.Exception
```

An exception that provides information on IF-IMC/IF-IMV errors. This exception class which wraps the result codes defined in the IF-IMC and IF-IMV Abstract API MUST be implemented by all TNCCs and TNCSs.

Each method in the IF-IMC/IF-IMV API throws an exception to indicate reason for failure. IMCs, IMVs, TNCCs and TNCSs MUST be prepared for any method to throw an TNCException.

This class defines a set of standard result codes. Vendor-specific result codes may be used but must be constructed as described in the abstract API. Any unknown result code SHOULD be treated as equivalent to TNC_RESULT_OTHER.

If an IMC or IMV method returns TNC_RESULT_FATAL, then the IMC or IMV has encountered a permanent error. The TNCC or TNCS SHOULD call the IMC or IMV's

`terminate` method as soon as possible. The TNCC or TNCS MAY then try to reinitialize the IMC or IMV with the IMC or IMV's `initialize` method or try other measures.

If a TNCC or TNCS method returns TNC_RESULT_FATAL, then the TNCC or TNCS has encountered a permanent error.

Field Detail

`public static final long` **`TNC_RESULT_NOT_INITIALIZED`**
The IMC or IMV's `initialize` method has not been called.

`public static final long` **`TNC_RESULT_ALREADY_INITIALIZED`**
The IMC or IMV's `initialize` method was called twice without a call to the IMC or IMV's `terminate` method.

`public static final long` **`TNC_RESULT_CANT_RETRY`**
TNCC or TNCS cannot attempt handshake retry.

`public static final long` **`TNC_RESULT_WONT_RETRY`**
TNCC or TNCS refuses to attempt handshake retry.

`public static final long` **`TNC_RESULT_INVALID_PARAMETER`**
Method parameter is not valid.

`public static final long` **`TNC_RESULT_CANT_RESPOND`**
IMC or IMV cannot respond now.

`public static final long` **`TNC_RESULT_ILLEGAL_OPERATION`**
Illegal operation attempted.

`public static final long` **`TNC_RESULT_OTHER`**
Unspecified error.

`public static final long` **`TNC_RESULT_FATAL`**
Unspecified fatal error.

Constructor Detail

`public` **`TNCException`**`()`
Constructs a `TNCException` object; the `resultCode` field defaults to TNC_RESULT_OTHER.

`public` **`TNCException`**`(java.lang.String s,`
`long resultCode)`
Constructs a fully-specified `TNCException` object.
Parameters:
`s` - a description of the exception
`resultCode` - TNC result code

Method Detail

`public long` **`getResultCode`**`()`

Retrieves the TNC result code for this `TNCException` object.
Returns:
the TNC result code

4.3.12.2 TNCConstants (TNCConstants.java)

org.trustedcomputinggroup.tnc

`public interface` **`TNCConstants`**

A collection of well known or common constants to be used by the IMC and IMV packages.

Field Detail

`static final long` **`TNC_CONNECTION_STATE_CREATE`**
Network connection created.

`static final long` **`TNC_CONNECTION_STATE_HANDSHAKE`**
Handshake about to start.

`static final long` **`TNC_CONNECTION_STATE_ACCESS_ALLOWED`**
Handshake completed. TNC Server recommended that requested access be allowed.

`static final long` **`TNC_CONNECTION_STATE_ACCESS_ISOLATED`**
Handshake completed. TNC Server recommended that isolated access be allowed.

`static final long` **`TNC_CONNECTION_STATE_ACCESS_NONE`**
Handshake completed. TNCS Server recommended that no network access be allowed.

`static final long` **`TNC_CONNECTION_STATE_DELETE`**
About to delete network connection . Remove all associated state.

`static final long` **`TNC_VENDORID_ANY`**
Wild card matching any vendor ID.

`static final long` **`TNC_SUBTYPE_ANY`**
Wild card matching any message subtype.

`static final long` **`TNC_ATTRIBUTEID_PREFERRED_LANGUAGE`**
Preferred human-readable language(s) as an Accept-Language header (type String, may get from a TNCS or IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_REASON_STRING`**
Reason for IMV Recommendation (type String, may set for an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_REASON_LANGUAGE`**
Language(s) for Reason String as an RFC 3066 language tag (type String, may set for an IMVConnection)

4.3.12.3 TNCS Interface (TNCS.java)

org.trustedcomputinggroup.tnc.ifimv

`public interface` **`TNCS`**

These methods are implemented by the TNC Server and called by the IMV.

Method Detail

```
void reportMessageTypes(IMV imv,
                        long[] supportedTypes)
                        throws TNCException
```

An IMV calls this method to inform a TNCS about the set of message types the IMV is able to receive. Often, the IMV will call this method from the IMV's initialize method.

A list of message types is contained in the supportedTypes parameter. The supportedTypes parameter may be null to represent no message types.

All TNC Servers MUST implement this method. The TNC Server MUST NOT ever modify the list of message types and MUST NOT access this list after the TNCS' reportMessageTypes method has returned. Generally, the TNC Server will copy the contents of this list before returning from this method. TNC Servers MUST support any message type.

Note that although all TNC Servers must implement this method, some IMVs may never call it if they don't support receiving any message types. This is acceptable. In such a case, the TNC Server MUST NOT deliver any messages to the IMV.

If an IMV 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 IMV 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 IMV calls the TNCS' reportMessageTypes method more than once, the message type list supplied in the latest call supplants the message type lists supplied in earlier calls.

Parameters:
`imv` - the IMV reporting its message types
`supportedTypes` - the message types the IMV wishes to receive
Throws:
TNCException

```
void requestHandshakeRetry(IMV imv,
                           long reason)
                           throws TNCException
```

IMVs can call this method to ask a TNCS to retry an Integrity Check Handshake for all current network connections. The IMV MUST pass itself as the imv parameter and one of the handshake retry reasons listed in IMVConnection as the reason parameter.

TNCSs MAY check the parameters to make sure they are valid and throw an exception if not, but TNCSs are not required to make these checks. The reason parameter explains why the IMV is requesting a handshake retry. The TNCS

MAY use this in deciding whether to attempt the handshake retry. As noted in the Abstract API, TNCSs are not required to honor IMV requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMV MAY call this method at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMV suddenly gets important information but has already finished its dialog with the IMC, for instance. As always, the TNCS is not required to honor the request for handshake retry.

If the TNCS cannot attempt the handshake retry, it SHOULD throw a TNCException with result code TNC_RESULT_CANT_RETRY. If the TNCS could attempt to retry the handshake but chooses not to, it SHOULD throw the TNC_RESULT_WONT_RETRY exception. If the TNCS intends to retry the handshake, it SHOULD throw a TNCException with result code TNC_RESULT_WONT_RETRY. The IMV MAY use this information in displaying diagnostic and progress messages.

Parameters:
`imv` - IMV object
`reason` - reason for retry handshake request
Throws:
`TNCException`

`java.lang.Object` **`getAttribute`**`(long attributeID)`
`throws TNCException`

An IMV calls this method to get the value of the attribute identified by `attributeID` for this `TNCS`.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this TNCS, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not have an attribute value for the requested attribute ID for this `TNCS`, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

The return value is an `Object` that represents the attribute value requested. The IMV must cast this `Object` to the class documented in the description of that specific attribute to get the desired value. All `Object`s returned by this method SHOULD be immutable.

Parameters:
`attributeID` - the attribute ID of the desired attribute
Returns:
the attribute value

Throws:
TNCException

```
void setAttribute(long attributeID,
                  java.lang.Object attributeValue)
                  throws TNCException
```

An IMV calls this method to set the value of the attribute identified by `attributeID` for this `TNCS`.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this TNCS, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not support setting an attribute value for the requested attribute ID for this `TNCS`, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

For the `attributeValue` parameter, the IMV MUST pass an `Object` that represents the new attribute value (or `null` if permitted for the specified attribute). This `Object` must actually be an instance of the class documented in the description of the specified attribute. The `Object` SHOULD be immutable. If the TNCS has any uncertainty about it SHOULD copy the object. The TNCS MAY check the `Object` and throw a `TNCException` if it is not a valid value for the specified attribute.

Parameters:
`attributeID` - the attribute ID of the attribute to be set
`attributeValue` - the new value to be set for this attribute
Throws:
TNCException

4.3.12.4 IMV Interface (IMV.java)

org.trustedcomputinggroup.tnc.ifimv

```
public interface IMV
```

An Integrity Measurement Verifier (IMV). These methods are implemented by the IMV and called by the TNC Server.

Method Detail

```
void initialize(TNCS tncs)
                throws TNCException
```

Initializes the IMV. All IMVs MUST implement this Method. The TNC Server supplies itself as a parameter so the IMV can call the TNCS as needed.

The TNC Server MUST NOT call any other IF-IMV API Methods for an IMV until it has successfully completed a call to the IMV's initialize method. Once a call to this method has completed successfully, this method MUST NOT be called again for a particular IMV-TNCS pair until a call to the IMV's terminate method has completed successfully. the IMV.

Parameters:
`tncs` - the TNC Server
Throws:
`TNCException` - if a TNC error occurs

```
void terminate()
              throws TNCException
```

Closes down the IMV. The TNC Server calls this method when all work is complete and the TNCS is preparing to shut down or when the IMV throws a TNC_RESULT_FATAL exception. Once a call to an IMV's terminate method is made, the TNC Server MUST NOT call the IMV except to call the IMV's initialize method (which may not succeed if the IMV cannot reinitialize itself). Even if the IMV returns an error from this method, the TNC Server MAY continue with its unload or shutdown procedure.
Throws:
`TNCException` - if a TNC error occurs

```
void notifyConnectionChange(IMVConnection c,
                            long newState)
                            throws TNCException
```

Informs the IMV that the state of IMVConnection c has changed to newState. The TNCConstants interface lists all the possible values of the new state for this version of the IF-IMV API. The TNCS MUST NOT use any other values with this version of IF-IMV.

IMVs that want to track the state of network connections or maintain per-connection data structures SHOULD implement this method. Other IMVs MAY implement it. If an IMV chooses to not implement this method it MUST throw an UnsupportedOperationException.

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

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 IMV SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection to the TNC Server after this method returns.

Parameters:
`c` - the IMV Connection
`newState` - new network connection state
Throws:
`TNCException`

```
void receiveMessage(IMVConnection c,
```

```
                long messageType,
                byte[] message)
                throws TNCException
```

The TNC Server calls this method to deliver a message to the IMV. The message is contained in the buffer referenced by message. The type of the message is indicated by messageType. The message MUST be from an IMC (or a TNCC or other party acting as an IMC).

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

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

The IMV should implement this method if it wants to receive messages. Most IMVs will do so, since they will base their IMV Action Recommendations on measurements received from the IMC. However, some IMVs may base their IMV Action Recommendations on other data such as reports from intrusion detection systems or scanners. Those IMVs need not implement this method.

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after the IMV receiveMessage() method has returned. If the IMV wants to retain the message, it should copy it before returning from this method.

The message parameter may be null to represent an empty message. In the messageType parameter, the TNCS MUST pass the type of the message. This value MUST match one of the TNC_MessageType values previously supplied by the IMV to the TNCS in the IMV's most recent call to the TNCS reportMessageTypes method. IMVs MAY check these parameters to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Parameters:
`c` - the IMVConnection
`messageType` - the message type that is being delivered to the IMV
`message` - the message that is being delivered to the IMV
Throws:
TNCException

```
void solicitRecommendation(IMVConnection c)
                           throws TNCException
```

The TNC Server calls this method at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered) to solicit recommendations from IMVs that have not yet provided a recommendation. The TNCS MUST NOT call this method for an IMV for a particular connection if that IMV has already called `provideRecommendation` on that connection since the TNCS last called `notifyConnectionChange` for that IMV with that connection. If an IMV is not able to provide a recommendation at this time, it

SHOULD call the TNCS provideRecommendation() method with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. If an IMV returns from this method without calling the TNCS' provideRecommendation method, the TNCS MAY consider the IMV's Action Recommendation to be TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. The TNCS MAY take other actions, such as logging this IMV behavior, which is erroneous.

All IMVs MUST implement this method.

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 TNCS will return TNC_RESULT_ILLEGAL_OPERATION from sendMessage and call the solicitRecommendation method to elicit IMV Action Recommendations based on the data they have gathered so far.

Parameters:
`c` - IMV connection
Throws:
`TNCException`

`void` **`batchEnding`**`(IMVConnection c)`
`throws TNCException`

The TNC Server calls this method to notify IMVs that all IMC messages received in a batch have been delivered and this is the IMV's last chance to send a message in the batch of IMV messages currently being collected. An IMV MAY implement this method if it wants to perform some actions after all the IMC messages received during a batch have been delivered (using the IMV's receiveMessage method). For instance, if an IMV has not received any messages from an IMC it may conclude that its IMC is not installed on the endpoint and may decide to call the TNCS' provideRecommendation method with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS.

An IMV MAY call the TNCS' sendMessage method from this method. As with all IMV methods, the IMV SHOULD NOT wait a long time before returning from the batchEnding method. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

Parameters:
`c` - IMV connection
Throws:
`TNCException`

4.3.12.5 IMVConnection Interface (IMVConnection.java)

org.trustedcomputinggroup.tnc.ifimv

`public interface` **`IMVConnection`**

The IMV and TNCS use this IMVConnection object to refer to the network connection when delivering messages and performing other operations relevant to the network

connection. This helps ensure that IMV messages are sent to the right TNCC and IMCs, helps ensure that the IMV Action Recommendation is associated with the right endpoint, and helps the IMV match up messages from IMCs with any state the IMV may be maintaining from earlier parts of that IMC-IMV conversation (even extending across multiple Integrity Check Handshakes in a single network connection).

The TNCS MUST create a new IMVConnection object for each combination of an IMV and a connection. IMVConnection objects MUST NOT be shared between multiple IMVs.

Field Detail

`static final long` **`TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE`**
IMV policy has changed. It recommends handshake retry even if network connectivity must be interrupted;

`static final long` **`TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE`**
IMV policy has changed. It requests handshake retry but not if network connectivity must be interrupted;

`static final long` **`TNC_RETRY_REASON_IMV_SERIOUS_EVENT`**
IMV has detected a serious event and recommends handshake retry even if network connectivity must be interrupted.

`static final long` **`TNC_RETRY_REASON_IMV_MINOR_EVENT`**
IMV has detected a minor event. It requests handshake retry but not if network connectivity must be interrupted.

`static final long` **`TNC_RETRY_REASON_IMV_PERIODIC`**
IMV wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_ALLOW`**
IMV recommends allowing access.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS`**
IMV recommends no access.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_ISOLATE`**
IMV recommends limited access. This access may be expanded after remediation.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION`**
IMV does not have a recommendation.

`static final long` **`TNC_IMV_EVALUATION_RESULT_COMPLIANT`**
AR complies with policy.

`static final long` **`TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MINOR`**
AR is not compliant with policy. Non-compliance is minor.

`static final long` **`TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MAJOR`**
AR is not compliant with policy. Non-compliance is major.

`static final long` **`TNC_IMV_EVALUATION_RESULT_ERROR`**

IMV is unable to determine policy compliance due to error.

`static final long` **`TNC_IMV_EVALUATION_RESULT_DONT_KNOW`**
IMV does not know whether AR complies with policy.

Method Detail

`void` **`sendMessage`**`(long messageType, byte[] message) throws TNCException`

Gives a message to the TNCS for delivery. The message is contained in the buffer referenced by the `message` parameter. The `message` parameter may be null which represent an empty message. The type of the message is indicated by the messageType parameter.

All IMVConnections MUST implement this method. An IMVConnection MUST NOT ever modify the buffer contents and MUST NOT access the buffer after the sendMessage method has returned. The IMVConnection will typically copy the message out of the buffer, queue it up for delivery, and return from this method.

The IMV MUST NOT call this method unless it has received a call to the IMV's receiveMessage method or the IMV's batchEnding method for this connection and the IMV has not yet returned from that method. If the IMV violates this prohibition, the TNCS SHOULD throw the TNC_RESULT_ILLEGAL_OPERATION exception. If an IMV really wants to communicate with an IMC at another time, it should call the IMVConnection's requestHandshakeRetry method.

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 IMVConnection's sendMessage method will throw a TNCException with result code TNC_RESULT_ILLEGAL_OPERATION and call the IMVs' solicitRecommendation() to elicit IMV Action Recommendations based on the data they have gathered so far.

The TNC Server MUST support any message type. However, the IMV 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 the Abstract API. If the IMV violates this prohibition, the IMVConnection SHOULD throw a TNCException with result code TNC_RESULT_INVALID_PARAMETER.

Parameters:
`messageType` - the type of message to be delivered
`message` - the message to be delivered
Throws:
`TNCException` - if an error occurs

`void` **`requestHandshakeRetry`**`(long reason) throws TNCException`

Asks a TNCS to retry an Integrity Check Handshake for this IMVConnection. The IMV MUST pass one of the handshake retry reasons listed in the Abstract API as the reason parameter.

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

If the TNCS cannot attempt the handshake retry, the IMVConnection SHOULD throw a TNCException with result code TNC_RESULT_CANT_RETRY. If the TNCS could attempt to retry the handshake but chooses not to, the IMVConnection SHOULD throw a TNCException with result code TNC_RESULT_WONT_RETRY. The IMV MAY use this information in displaying diagnostic and progress messages.

Parameters:
`reason` - handshake retry reason code
Throws:
`TNCException`

```
void provideRecommendation(long recommendation,
                           long evaluation)
                           throws TNCException
```

An IMV calls this method to deliver its IMV Action Recommendation and IMV Evaluation Result to the TNCS. The TNCS SHOULD use the recommendation value in determining its own TNCS Action Recommendation or decision about endpoint access. The TNC specifications do not specify how the TNCS does the recommendation value but it is certainly essential to have a recommendation from the IMV. The TNC specifications also do not specify what the TNCS does with the evaluation value. It may log it.

The IMV MUST pass one of the IMV Action Recommendation values listed in the Abstract API as the recommendation parameter and one of the IMV Evaluation Result values listed in the Abstract API as the evaluation parameter. TNCSs MAY check these values to make sure they are valid and throw an exception if not, but TNCSs are not required to make these checks.

The IMV should deliver its IMV Action Recommendation as soon as possible so that the TNCS can proceed with determining its own TNCS Action Recommendation. If the IMV receives a message from an IMC and is able to decide on an IMV Action Recommendation and deliver it to the TNCS before returning from the IMV receiveMessage method, it SHOULD do so. However, as

always the IMV SHOULD return promptly to avoid a long delay that might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

An IMV SHOULD NOT expect that it will be able to send IMC-IMV messages after calling the IMVConnection's provideRecommendation method. The TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation. For instance, IMVs SHOULD send remediation instructions before calling the IMVConnection's provideRecommendation method.

However, a TNCS MAY continue to deliver messages after an IMV calls the IMVConnection's provideRecommendation method, especially if other IMVs continue the dialog after the one IMV has rendered its decision. The IMV MUST be prepared for this. It MAY simply ignore these late messages or it MAY consider them and even change its recommendation by calling the IMVConnection's provideRecommendation method again. In this case, the TNCS SHOULD use the last recommendation received from an IMV during a particular handshake. However, the TNCS is not required to do this.

If an IMV does not provide a recommendation earlier, the TNCS will call the IMV's solicitRecommendation method at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered). The IMV SHOULD then call the IMVConnection's provideRecommendation method to deliver its recommendation. If the IMV calls this method when there is no active handshake on the specified network connection, the TNCS SHOULD throw the TNC_RESULT_ILLEGAL_OPERATION exception. If an IMV really needs to communicate a recommendation at another time, it should call the IMVConnection's requestHandshakeRetry method.

Parameters:
`recommendation` - action recommendation
`evaluation` - evaluation result
Throws:
`TNCException`

`java.lang.Object` **`getAttribute`**`(long attributeID)`
`throws TNCException`

An IMV calls this method to get the value of the attribute identified by `attributeID` for this `IMVConnection`.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this IMVConnection, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not have an attribute value for

the requested attribute ID for this `IMVConnection`, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

The return value is an `Object` that represents the attribute value requested. The IMV must cast this `Object` to the class documented in the description of that specific attribute to get the desired value. All `Object`s returned by this method SHOULD be immutable.

Parameters:
`attributeID` - the attribute ID of the desired attribute
Returns:
the attribute value
Throws:
`TNCException`

```
void setAttribute(long attributeID,
                  java.lang.Object attributeValue)
                  throws TNCException
```

An IMV calls this method to set the value of the attribute identified by `attributeID` for this `IMVConnection`.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this IMVConnection, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not support setting an attribute value for the requested attribute ID for this `IMVConnection`, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

For the `attributeValue` parameter, the IMV MUST pass an `Object` that represents the new attribute value (or `null` if permitted for the specified attribute). This `Object` must actually be an instance of the class documented in the description of the specified attribute. The `Object` SHOULD be immutable. If the TNCS has any uncertainty about it SHOULD copy the object. The TNCS MAY check the `Object` and throw a `TNCException` if it is not a valid value for the specified attribute.

Parameters:
`attributeID` - the attribute ID of the attribute to be set
`attributeValue` - the new value to be set for this attribute
Throws:
`TNCException`

5 Security Considerations

This section describes the security threats related to IF-IMV and suggests methods to address these threats. The components involved in IF-IMV are one or more Trusted Network Connect Servers (TNCS) and one or more Integrity Measurement Verifiers (IMVs). These are logical components; the TNCS and IMVs reside on the same host. The IF-IMV is the interface between the TNCS and the IMVs.

A multitude of remote distributed endpoints is often more difficult to manage securely than a small number of centralized servers; therefore, it is highly recommended that IMV and TNCS implementers read and understand the Security Considerations of the IF-IMC [6] in addition to the considerations in this document.

5.1 Threat analysis

5.1.1 Registration and Discovery based threats

The TNCS discovers which IMVs can be loaded on a host via a platform-specific binding, for example, on the Windows platform using a windows registry key and on the Linux or Unix platform using a configuration file. On Windows, this implies the registry keys are typically created when the IMVs are installed, requiring the IMV installer to possess sufficient privileges on the platform. Similarly the TNCS must have sufficient privileges to read the relevant keys. Based on the IMVs discovered in the registry, the TNCS 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 IMV (Spoofing)
- It can remove a valid IMV, perhaps replacing it with rogue/modified versions of code (Tamper)

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

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

5.1.2 Rogue IMV threats

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

- Overwrite TNCS or IMV memory
- Violate IF-IMV API requirements such as passing illegal or unexpected argument values
- Perform illegal operations so that the TNCS is terminated by the operating system
- Perform improper operations with the TNCS' privileges
- Attack other components (such as the NAA or remote servers) using the privileges or credentials of the TNCS or other IMVs
- 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 the AR ("IMV Spyware").
- Issue a large number of, or particularly expensive, interface API calls to the TNCS (Denial of service of the TNCS)
- Provide incorrect IMV Action Recommendations, causing valid clients to be rejected or invalid clients to be let on the network
- Provide incorrect IMV Evaluation Results, causing the system state to not reflect the true compliance state of the endpoint
- Spoof TNCS calls to an IMV and provide incorrect handshake or compliance data to IMVs
- Spuriously request handshake retries (Denial of service)

- Lock up TNCS threads by not returning from function calls (Denial of service)
- Use vendor-specific extensions to IF-IMV to perform other attacks

5.1.3 Rogue TNCS threats

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

- Overwrite IMV memory
- Violate IF-IMV API requirements such as passing illegal or unexpected argument values
- Attack other components (such as the NAA or remote servers) using the credentials of an IMV
- 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 the AR
- Issue a large number of, or particularly expensive, interface API calls to an IMV, possibly causing denial of service of a remote server
- Provide incorrect TNCS Action Recommendations to a NAA, causing valid clients to be rejected or invalid clients to be let on the network
- Spuriously request or perform handshake retries (Denial of service)
- Use vendor-specific extensions to IF-IMV to perform other attacks

5.1.4 Man-in-the-Middle Threats

If an attacker injects a man-in-the-middle between an IMV and its corresponding IMC peer entity, between an IMV and its corresponding TNCS, or between a TNCS and a TNCC it may be able to be misused in the following ways:

- Allows the viewing, modification, deletion, or addition, of messages passing between the IMV and the IMC,between the IMV and its corresponding TNCS, or between the TNCS and the TNCC,
- Allow the replay of measurements or other messages that are not reflective of the Access Requestor's current conditions.

5.1.5 Tampering Threats on IMVs and TNCSs

Malicious code (worms, viruses, etc) or another unauthorized application can modify an IMV or TNCS. This allows the attacker to misuse TNC components in the following ways:

- Modify legitimate messages, add new illegitimate messages, or delete legitimate messages.
- Allow the attack to exfiltrate measurements and other data from an Access Requestor.

5.1.6 Threats Beyond IF-IMV

IF-IMV is part of the larger TNC architecture. Successful attacks against other parts of the TNC architecture will generally result in negative effects for IMVs, TNCSs, 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 IMVs be loaded by a TNCS and only authorized TNCSs be allowed to load an IMV. There are well known methods to control what code is loaded by a TNCS:

- 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 IMV registration mechanism (registry or configuration file)
- Control access to IMV code and critical data files
- Employ a TNCS-specific list of authorized IMVs

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

Industry standard best practices for secure coding, software engineering, and code reviews should be followed during the development of IMVs and TNCSs in order to minimize the possibility of incorrect design, incorrect implementation, and software flaws thus mitigating some of the attacks described above.

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. In addition, using PTS for dynamic measuring of TNC components during runtime and also mitigate the attacks related to tampering.

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 IMVs can be monitored.

"IMV spyware" attacks can often be prevented administratively; for example, by prohibiting unknown programs from making unauthorized network connections, or by monitoring the disk for log files created by unknown IMVs which are simply logging messages.
Note that invalid handshake retries can be mitigated by only allowing a retry on a valid session that is associated with each particular IMV ID.

This specification requires that all valid IMVs be installed to a protected system directory. The loading of a rogue IMV 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 IMV to be installed.

Further protection against rogue IMVs (and also against buggy IMVs) can be provided by having the TNCS launch a new "child" process for each IMV, having the child process load the IMV, and then having the TNCS communicate with the child processes carefully. This limits the amount of damage that can be done by a rogue IMV. The TNCS may use this approach but is not required to do so.

IMV implementers who choose a stub-to-backend-server implementation must take care not to make the stub-to-server 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.

One countermeasure for a man-in-the-middle attack is to make use of the PTS described in this section earlier. An additional countermeasure is to have the IF-M protocol (between the IMV and the IMC) and/or the IF-TNCCS (between the TNCS and the TNCC) provide both strong mutual authentication and anti-replay technology in a similar manner to the IF-T protocol. Note: Future enhancements to this specification may be necessary for this type of counter measure.

Protection from many of the identified threats can be provided by housing the IMVs and TNCSs separately from that which is being measured. If a particular component exists within an isolated environment, the chance of it being compromised is far reduced. It is recommended that careful

analysis of the threat environment that a TNC implementation will be deployed into be conducted and the strongest such isolation that makes sense in that environment be applied.

6 C Header File

This section provides a C header file that serves as a binding for the IF-IMV 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 IMVs and TNC Servers on their platform.

```
/* tncifimv.h
 *
 * Trusted Network Connect IF-IMV API version 1.20
 * Microsoft Windows DLL Platform Binding C Header
 * February 5, 2007
 *
 * Copyright(c) 2005-2007, Trusted Computing Group, Inc. All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * • Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * • Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in
 *   the documentation and/or other materials provided with the
 *   distribution.
 * • Neither the name of the Trusted Computing Group nor the names of
 *   its contributors may be used to endorse or promote products
 *   derived from this software without specific prior written
 *   permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Contact the Trusted Computing Group at
 * admin@trustedcomputinggroup.org for information on specification
 * licensing through membership agreements.
 *
 * Any marks and brands contained herein are the property of their
 * respective owners.
 */

#ifndef _TNCIFIMV_H
#define _TNCIFIMV_H

#ifdef __cplusplus
extern "C" {
#endif
```

```
#ifdef WIN32
#ifdef TNC_IMV_EXPORTS
#define TNC_IMV_API __declspec(dllexport)
#else
#define TNC_IMV_API __declspec(dllimport)
#endif
#else
#define TNC_IMV_API
#endif

/* Basic Types */

typedef unsigned long TNC_UInt32;
typedef unsigned char *TNC_BufferReference;

/* Derived Types */

typedef TNC_UInt32 TNC_IMVID;
typedef TNC_UInt32 TNC_ConnectionID;
typedef TNC_UInt32 TNC_ConnectionState;
typedef TNC_UInt32 TNC_RetryReason;
typedef TNC_UInt32 TNC_IMV_Action_Recommendation;
typedef TNC_UInt32 TNC_IMV_Evaluation_Result;
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;
typedef TNC_UInt32 TNC_AttributeID;


/* Function pointers */

typedef TNC_Result (*TNC_IMV_InitializePointer)(
    TNC_IMVID imvID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);
typedef TNC_Result (*TNC_IMV_NotifyConnectionChangePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);
typedef TNC_Result (*TNC_IMV_ReceiveMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_IMV_SolicitRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMV_BatchEndingPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMV_TerminatePointer)(
```

```
    TNC_IMVID imvID);
typedef TNC_Result (*TNC_TNCS_ReportMessageTypesPointer)(
    TNC_IMVID imvID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCS_SendMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_TNCS_RequestHandshakeRetryPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);
typedef TNC_Result (*TNC_TNCS_ProvideRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_IMV_Action_Recommendation recommendation,
    TNC_IMV_Evaluation_Result evaluation);
typedef TNC_Result (*TNC_TNCS_GetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_AttributeId attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);
typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);
typedef TNC_Result (*TNC_TNCS_BindFunctionPointer)(
    TNC_IMVID imvID,
    char *functionName,
    void **pOutfunctionPointer);
typedef TNC_Result (*TNC_IMV_ProvideBindFunctionPointer)(
    TNC_IMVID imvID,
    TNC_TNCS_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
/* reserved for 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_IFIMV_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 */

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

/* IMV Action Recommendation Values */

#define TNC_IMV_ACTION_RECOMMENDATION_ALLOW 0
#define TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS 1
#define TNC_IMV_ACTION_RECOMMENDATION_ISOLATE 2
#define TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION 3

/* IMV Evaluation Result Values */

#define TNC_IMV_EVALUATION_RESULT_COMPLIANT 0
#define TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MINOR 1
#define TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MAJOR 2
#define TNC_IMV_EVALUATION_RESULT_ERROR 3
#define TNC_IMV_EVALUATION_RESULT_DONT_KNOW 4

/* Vendor ID Values */

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

/* Message Subtype Values */

#define ((TNC_MessageSubtype) 0xff)

/* Message Attribute ID Values */

#define TNC_ATTRIBUTEID_PREFERRED_LANGUAGE ((TNC_AttributeID) 0x00000001)
#define TNC_ATTRIBUTEID_REASON_STRING ((TNC_AttributeID) 0x00000002)
#define TNC_ATTRIBUTEID_REASON_LANGUAGE ((TNC_AttributeID) 0x00000003)
```

```
/* IMV Functions */

TNC_IMV_API TNC_Result TNC_IMV_Initialize(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_Version minVersion,
/*in*/  TNC_Version maxVersion,
/*in*/  TNC_Version *pOutActualVersion);

TNC_IMV_API TNC_Result TNC_IMV_NotifyConnectionChange(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_ConnectionState newState);

TNC_IMV_API TNC_Result TNC_IMV_ReceiveMessage(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference messageBuffer,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);

TNC_IMV_API TNC_Result TNC_IMV_SolicitRecommendation(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMV_API TNC_Result TNC_IMV_BatchEnding(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMV_API TNC_Result TNC_IMV_Terminate(
/*in*/  TNC_IMVID imvID);

TNC_IMV_API TNC_Result TNC_IMV_ProvideBindFunction(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_TNCS_BindFunctionPointer bindFunction);

/* TNC Server Functions */

TNC_Result TNC_TNCS_ReportMessageTypes(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_MessageTypeList supportedTypes,
/*in*/  TNC_UInt32 typeCount);

TNC_Result TNC_TNCS_SendMessage(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference message,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);

TNC_Result TNC_TNCS_RequestHandshakeRetry(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_RetryReason reason);

TNC_Result TNC_TNCS_ProvideRecommendation(
/*in*/  TNC_IMVID imvID,
```

```
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_IMV_Action_Recommendation recommendation,
/*in*/  TNC_IMV_Evaluation_Result evaluation);

TNC_Result TNC_TNCS_GetAttribute(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_AttributeID attributeID,
/*in*/  TNC_UInt32 bufferLength,
/*out*/ TNC_BufferReference buffer,
/*out*/ TNC_UInt32 *pOutValueLength);
typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
/*in*/  TNC_IMVID imvID,
/*in*/ TNC_ConnectionID connectionID,

/*in*/ TNC_AttributeID attributeID,
/*in*/ TNC_UInt32 bufferLength,
/*in*/ TNC_BufferReference buffer);

TNC_Result TNC_TNCS_BindFunction(
/*in*/  TNC_IMVID imvID,
/*in*/  char *functionName,
/*in*/  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-IMV supports the use case. The text describing IF-IMV 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

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 manner in which the PEP, NAA, and TNCS find each other is not specified. The client-side components (TNCC and IMCs) find each other automatically using Microsoft Windows registry or a configuration file modified at install time.

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. **[IF-IMV] The details of the load process are platform-specific. With the Microsoft Windows DLL binding, the TNCS reads a protected registry key to find the IMV DLLs, then loads them. During the load process, the TNCS may check the integrity of the IMVs. This is optional.**
2. The TNCS initializes the IMVs through IF-IMV. **[IF-IMV] The TNCS calls `TNC_IMV_Initialize` for each IMV. The IMV performs any initialization it may need to, such as connecting to a remote server process or starting threads. Most IMVs will call `TNC_TNCS_ReportMessageTypes` to indicate which message types they would like to receive. With some platform bindings, this callback must wait until the next step when the Dynamic Function Binding mechanism is functional.**
3. **[IF-IMV] The TNCS performs any other platform-specific initialization needed. With the Microsoft Windows DLL binding, the TNC Server calls the `TNC_IMV_ProvideBindFunction` function to give each IMV a pointer to the bind function (`TNC_TNCS_BindFunction`) used for Dynamic Function Binding.**

7.3 TNCC Startup

1. When the TNCC starts up, the TNCC loads the IMCs.
2. The TNCC initializes the IMCs through IF-IMC.

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 PDP (NAA or TNCS). Depending on configuration, the PEP may contact the NAA first or the TNCS. The ordering of user authentication, platform authentication, and integrity check is also 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 17. 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.
7. The TNCS uses IF-IMV to inform each IMV that an Integrity Check Handshake has started. **[IF-IMV] If this is a new network connection, the TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_CREATE` to indicate that a new network connection has been created. Then the TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_HANDSHAKE`.**
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. **[IF-IMV] The TNCS delivers the IMC messages to the IMVs by calling `TNC_IMV_ReceiveMessage`. The IMVs may call `TNC_TNCS_SendMessage` before returning from `TNC_IMV_ReceiveMessage` if they want to send a response. When the TNCS has delivered all the IMC messages to the IMVs, it calls `TNC_IMV_BatchEnding` to inform them of this fact. The IMVs may call `TNC_TNCS_SendMessage` before returning from `TNC_IMV_BatchEnding` if they want to send a message to an IMV.**
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 and continues with step 11. 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. **[IF-IMV] As described in the previous step, IMVs send messages by calling `TNC_TNCS_SendMessage` before returning from `TNC_IMV_ReceiveMessage` and `TNC_IMV_BatchEnding`. IMVs give their results to the TNCS by calling `TNC_TNCS_ProvideRecommendation` at any time.**
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.
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. **[IF-IMV] The TNCS gives this prompt by calling `TNC_IMV_SolicitRecommendation`. The IMVs provide their recommendations by calling `TNC_TNCS_ProvideRecommendation`.**
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. **[IF-IMV] The TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_ACCESS_ALLOWED`, `TNC_CONNECTION_STATE_ACCESS_ISOLATED`, or `TNC_CONNECTION_STATE_ACCESS_NONE`.**

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.
20. If the IMCs 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.
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 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 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. **[IF-IMV] An IMV requests a handshake retry by calling** `TNC_TNCS_RequestHandshakeRetry`**. The TNCS makes the ultimate decision about whether to retry the handshake. As noted above, the handshake retry may disrupt network connectivity so the TNCS may decide to skip it. In that case, 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 C Binding Sequence Diagrams

7.7.1 Sequence Diagram for Network Connect

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

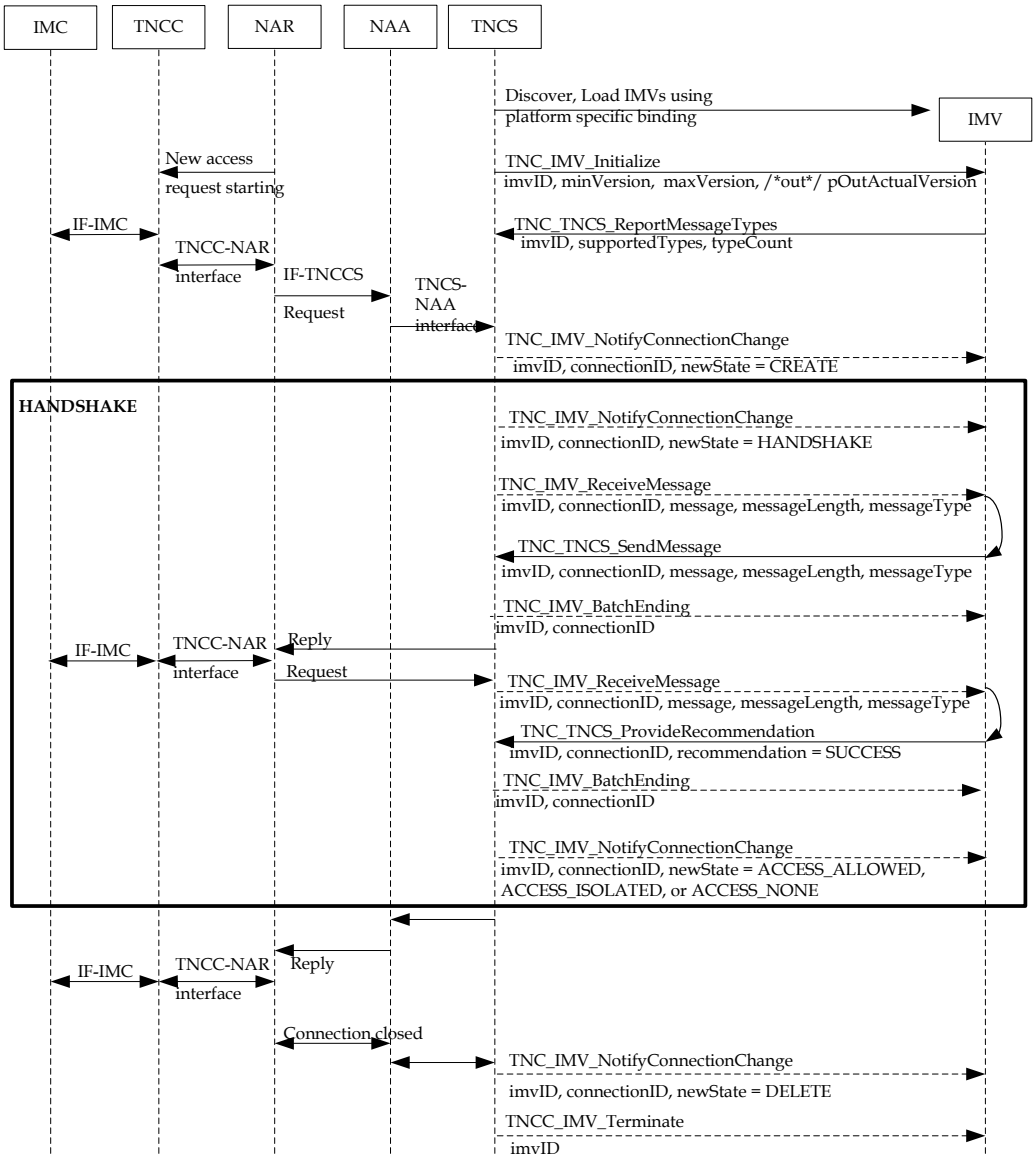


Figure 1 – C Binding: IF-IMV Network Connect Sequence Diagram

7.7.2 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.

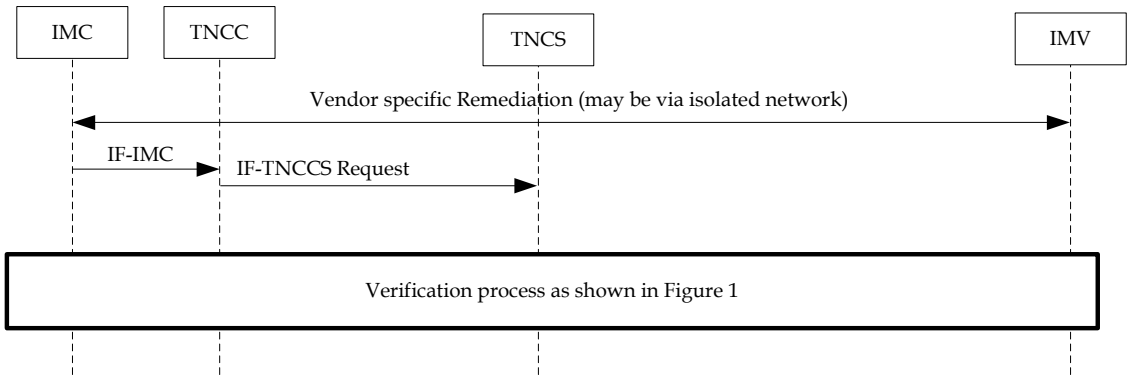


Figure 2 – C Binding: IF-IMV Handshake Retry After Remediation Sequence Diagram

7.7.3 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.

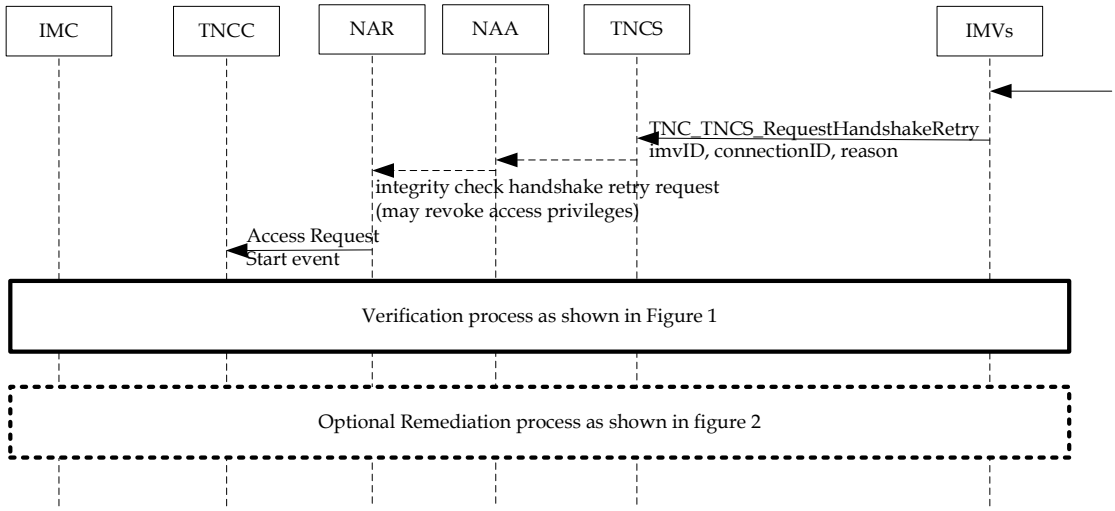


Figure 3 – C Bindng: IF-IMV Handshake Retry Initiated by TNCS

7.8 Java Binding Sequence Diagrams

7.8.1 Sequence Diagram for Network Connect

The following sequence diagram (Figure 4) illustrates the Network Connect use case, as described in section 7.4.

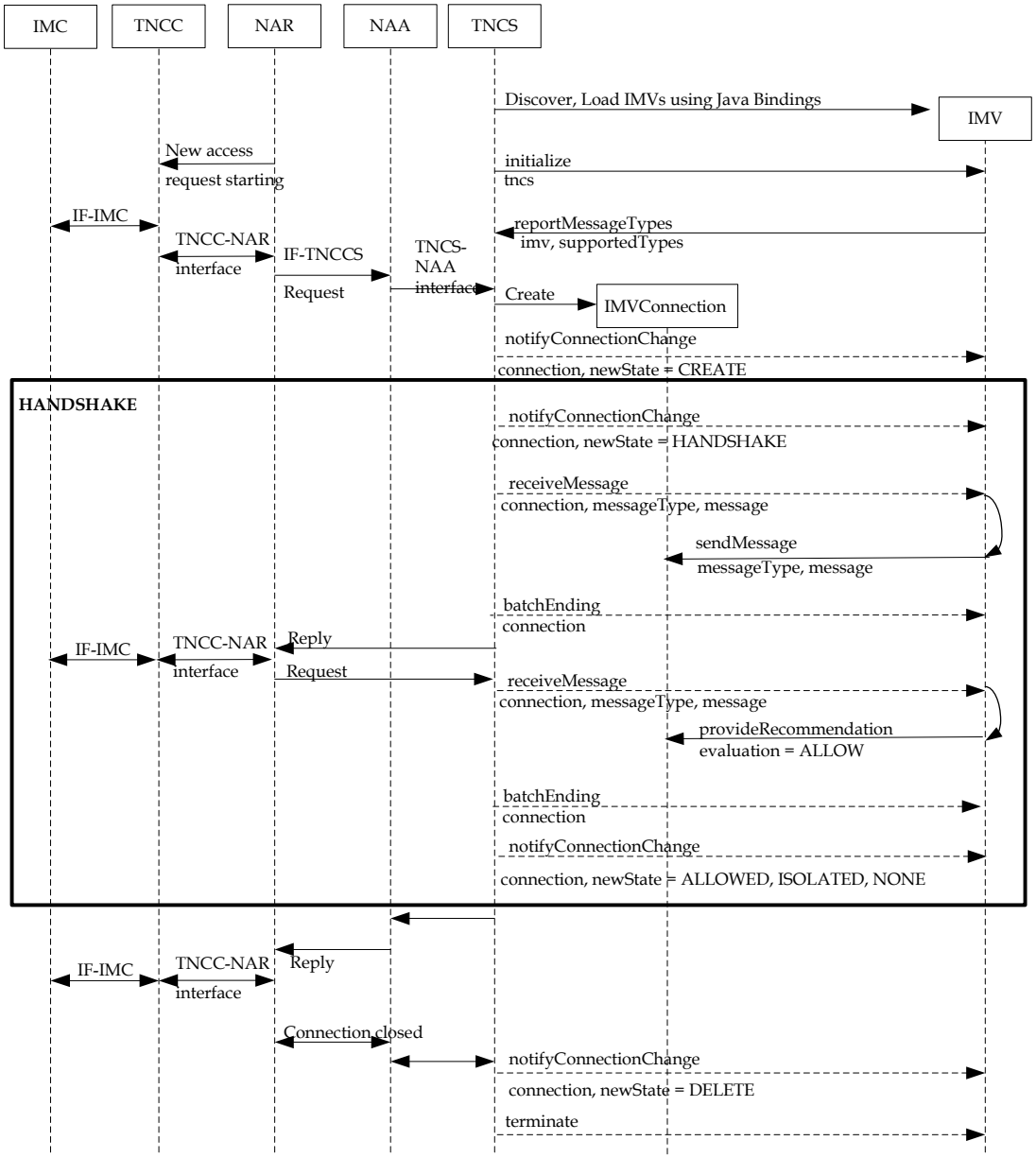


Figure 4 – Java Bindings: IF-IMV Network Connect Sequence Diagram

7.8.2 Sequence Diagram for Handshake Retry After Remediation

The following sequence diagram (Figure 5) illustrates the Handshake Retry After Remediation use case, as described in section 7.5.

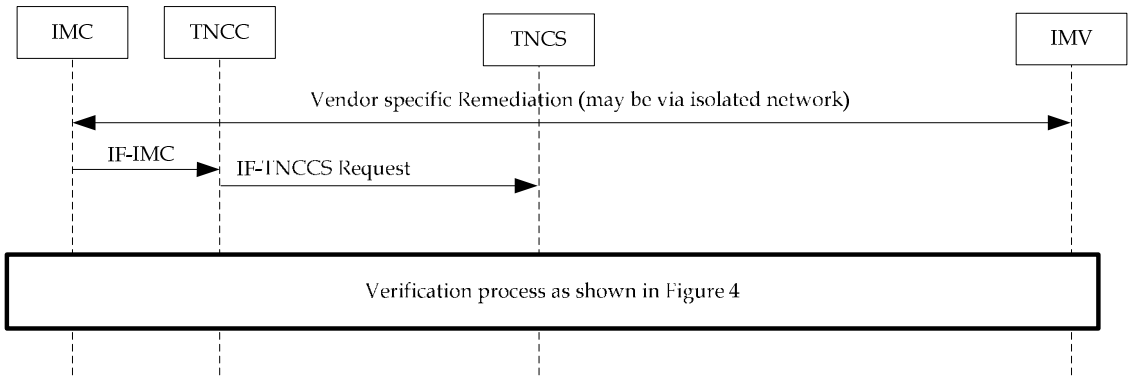


Figure 5– Java Bindings: IF-IMV Handshake Retry After Remediation Sequence Diagram

7.8.3 Sequence Diagram for Handshake Retry Initiated by TNCS

The following sequence diagram (Figure 6) illustrates the Handshake Retry Initiated by TNCS use case, as described in section 7.6.

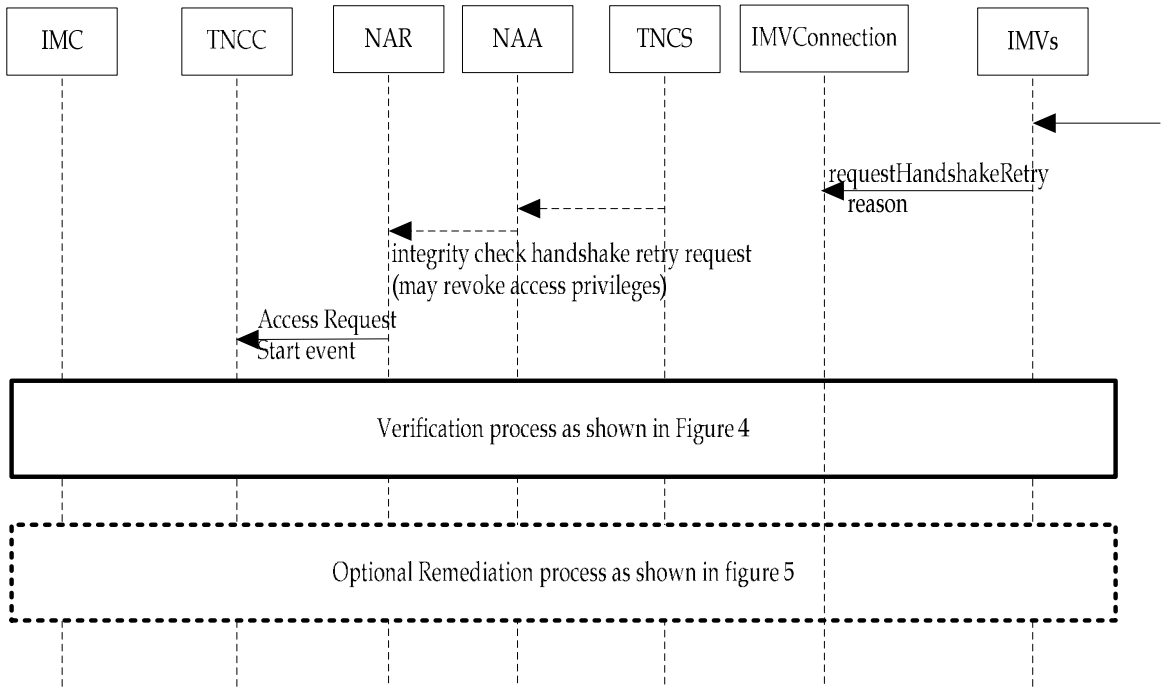


Figure 6 – Java Bindings: IF-IMV Handshake Retry Initiated by TNCS

8 Implementing a Simple IMV

This section provides a brief informative (non-binding) description of how to implement a simple IMV, one that only checks an IMC's integrity report against a policy and decides based on a recommendation based on this.

This example assumes that you're using the Microsoft Windows DLL platform binding. If not, replace the instructions in section 8.3 about `TNC_IMV_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 receive your value from the IMC 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_IMV_Initialize

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

8.3 TNC_IMV_ProvideBindFunction

Use the bind function to get a pointer to `TNC_TNCS_ReportMessageTypes`. Then use this pointer to call `TNC_TNCS_ReportMessageTypes` and report which message types you want to receive. Also use the bind function to get a pointer to `TNC_TNCS_ProvideRecommendation` for later use. This is the only state you need to keep. Return `TNC_RESULT_SUCCESS` unless an error occurs. In that case, return `TNC_RESULT_FATAL`.

8.4 TNC_IMV_ReceiveMessage

When you receive a message from an IMC, evaluate it against your policy and then report your recommendation by calling the `TNC_TNCS_ProvideRecommendation` function using the pointer that you saved earlier. If `TNC_TNCS_ProvideRecommendation` returns an error, then return that. Otherwise, return `TNC_RESULT_SUCCESS`.

8.5 TNC_IMV_SolicitRecommendation

If you never received a message from an IMC, you will be prompted to supply a recommendation by a call to `TNC_IMV_SolicitRecommendation`. Probably you will want to recommend against network access since your IMC is not loaded. In any case, report your recommendation by calling the `TNC_TNCS_ProvideRecommendation` function using the pointer that you saved earlier. If `TNC_TNCS_ProvideRecommendation` returns an error, then return that. Otherwise, return `TNC_RESULT_SUCCESS`.

8.6 All Done!

That's it! You've implemented your first IMV. If you need to do anything special on termination, you can implement `TNC_IMV_Terminate`. But many IMVs 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] Alvestrand, H., “Content Language Headers”, Internet Engineering Task Force RFC 3282, May 2002.

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

9.2 Informative References

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

[6] Trusted Computing Group, *TNC IF-IMC*, Specification Version 1.2, February 2007.