TLS Inner Application Extension
(TLS/IA)

Status of this Memo

This document is an Internet-Draft and is subject to all provisions of section 3 of RFC 3667. By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she become aware will be disclosed, in accordance with RFC 3668.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Copyright Notice

Copyright (C) The Internet Society (2001 - 2004). All Rights Reserved.

Abstract

This document defines a new TLS extension called "Inner Application". When TLS is used with the Inner Application extension (TLS/IA), additional messages are exchanged during the TLS handshake, each of which is an encrypted sequence of Attribute-Value-Pairs (AVPs) from the RADIUS/Diameter namespace. Hence, the AVPs defined in RADIUS and Diameter have the same meaning in TLS/IA;
that is, each attribute code point refers to the same logical attribute in any of these protocols. Arbitrary "applications" may be implemented using the AVP exchange. Possible applications include EAP or other forms of user authentication, client integrity checking, provisioning of additional tunnels, and the like. Use of the RADIUS/Diameter namespace provides natural compatibility between TLS/IA applications and widely deployed AAA infrastructures.

It is anticipated that TLS/IA will be used with and without subsequent protected data communication within the tunnel established by the handshake. For example, TLS/IA may be used to secure an HTTP data connection, allowing more robust password-based user authentication to occur within the TLS handshake than would otherwise be possible using mechanisms available in HTTP. TLS/IA may also be used for its handshake portion alone; for example, EAP-TTLSv1 encapsulates a TLS/IA handshake in EAP as a means to mutually authenticate a client and server and establish keys for a separate data connection.

Table of Contents

1 Introduction......................................................3
1.1 A Bit of History..............................................4
1.2 Handshake-Only vs. Full TLS Usage............................5
2 The InnerApplication Extension to TLS............................5
2.1 TLS/IA Overview............................................6
2.2 Message Exchange..............................................8
2.3 Master Key Permutation........................................8
2.3.1 Application Session Key Material........................10
2.4 Session Resumption...........................................11
2.5 Error Termination............................................12
2.6 Computing Verification Data................................12
2.7 TLS/IA Messages.............................................14
2.8 Negotiating the Inner Application Extension.................14
2.8.1 ClientInnerApplication.................................14
2.8.2 ServerInnerApplication.................................15
2.9 The PhaseFinished Handshake Message.........................16
2.10 The ApplicationPayload Handshake Message..................16
2.11 The InnerApplicationFailure Alert..........................16
3 Encapsulation of AVPs within ApplicationPayload Messages.......16
3.1 AVP Format....................................................17
3.2 AVP Sequences................................................18
3.3 Guidelines for Maximum Compatibility with AAA Servers....18
4 Tunneled Authentication within Application Phases...............19
4.1 Implicit challenge.............................................19
This specification defines the TLS "Inner Application" extension. The term "TLS/IA" refers to the TLS protocol when used with the Inner Application extension.

In TLS/IA, the TLS handshake is extended to allow an arbitrary exchange of information between client and server within a protected tunnel established during the handshake but prior to its completion. The initial phase of the TLS handshake is virtually identical to that of a standard TLS handshake; subsequent phases are conducted under the confidentiality and integrity protection afforded by that initial phase.

The primary motivation for providing such communication is to allow robust user authentication to occur as part of the handshake, in particular, user authentication that is based on password credentials, which is best conducted under the protection of an encrypted tunnel to preclude dictionary attack by eavesdroppers. For example, Extensible Authentication Protocol (EAP) may be used to authenticate using any of a wide variety of methods as part of the TLS handshake. The multi-phase approach of TLS/IA, in which a strong authentication, typically based on a server certificate, is used to protect a password-based authentication, distinguishes it from other TLS variants that rely entirely on a pre-shared key or password for security; for example [TLS-PSK].

The protected exchange accommodates any type of client-server application, not just authentication, though authentication may often be the prerequisite that allows other applications to proceed. For example, TLS/IA may be used to set up HTTP connections,
establish IPsec security associations (as an alternative to IKE), obtain credentials for single sign-on, provide for client integrity verification, and so on.

The new messages that are exchanged between client and server are encoded as sequences of Attribute-Value-Pairs (AVPs) from the RADIUS/Diameter namespace. Use of the RADIUS/Diameter namespace provides natural compatibility between TLS/IA applications and widely deployed AAA infrastructures. This namespace is extensible, allowing new AVPs and, thus, new applications to be defined as needed, either by standards bodies or by vendors wishing to define proprietary applications.

1.1 A Bit of History

The TLS protocol has its roots in the Netscape SSL protocol, which was originally intended to secure HTTP. It provides either one-way or mutual authentication of client and server based on certificates. In its most typical use in HTTP, the client authenticates the server based on the server’s certificate and establishes a tunnel through which HTTP traffic is passed.

For the server to authenticate the client within the TLS handshake, the client must have its own certificate. In cases where the client must be authenticated without a certificate, HTTP, not TLS, mechanisms would have to be employed. For example, HTTP headers have been defined to perform user authentications. However, these mechanisms are primitive compared to other mechanisms, most notably EAP, that have been defined for contexts other than HTTP. Furthermore, any mechanisms defined for HTTP cannot be utilized when TLS is used to protect non-HTTP traffic.

The TLS protocol has also found an important use in authentication for network access, originally within PPP for dial-up access and later for wireless and wired 802.1X access. Several EAP types have been defined that utilize TLS to perform mutual client-server authentication. The first to appear, EAP-TLS, uses the TLS handshake to authenticate both client and server based on the certificate of each.

Subsequent protocols, such EAP-TTLSv0 and EAP-PEAP, utilize the TLS handshake to allow the client to authenticate the server based on the latter’s certificate, then utilize the tunnel established by the TLS handshake to perform user authentication, typically based on password credentials. Such protocols are called "tunneled" EAP protocols. The authentication mechanism used inside the tunnel may itself be EAP, and the tunnel may also be used to convey additional information between client and server.

TLS/IA is in effect a merger of the two types of TLS usage described above, based on the recognition that tunneled authentication would
be useful in other contexts besides EAP. However, the tunneled protocols mentioned above are not directly compatible with a more generic use of TLS, because they utilize the tunneled data portion of TLS, thus precluding its use for other purposes such as carrying HTTP traffic.

The TLS/IA solution to this problem is to fold the tunneled authentication into the TLS handshake itself, making the data portion of the TLS exchange available for HTTP or any other protocol or connection that needs to be secured.

1.2 Handshake-Only vs. Full TLS Usage

It is anticipated that TLS/IA will be used with and without subsequent protected data communication within the tunnel established by the handshake.

For example, TLS/IA may be used to secure an HTTP data connection, allowing more robust password-based user authentication to occur within the TLS handshake than would otherwise be possible using mechanisms available in HTTP.

TLS/IA may also be used for its handshake portion alone. For example, EAP-TTLSv1 encapsulates a TLS/IA handshake in EAP as a means to mutually authenticate a client and server and establish keys for a separate data connection; no subsequent data portion is required. Another example might be use of TLS/IA directly over TCP to provide a user with credentials for single sign-on.

2 The InnerApplication Extension to TLS

The InnerApplication extension to TLS follows the guidelines of RFC 3546. The client proposes use of this extension by including a ClientInnerApplication message in its ClientHello handshake message, and the server confirms its use by including a ServerInnerApplication message in its ServerHello handshake message.

Two new handshake messages are defined for use in TLS/IA:

- The PhaseFinished message. This message is similar to the standard TLS Finished message; it allows the TLS/IA handshake to operate in phases, with message and key confirmation occurring at the end of each phase.

- The ApplicationPayload message. This message is used to carry AVP (Attribute-Value Pair) sequences within the TLS/IA handshake, in support of client-server applications such as authentication.

A new alert code is also defined for use in TLS/IA:

- The InnerApplicationFailure alert. This error alert allows either
party to terminate the handshake due to a failure in an application implemented via AVP sequences carried in ApplicationPayload messages.

2.1 TLS/IA Overview

In TLS/IA, the handshake is divided into phases. The first phase, called the "initial phase", is a standard TLS handshake; it is followed by zero or more "application phases". The last phase is called the "final phase"; this will be an application phase if a such a phase is present, otherwise the standard TLS handshake is both the initial and final phase. Any application phases between the initial and final phase are called "intermediate phases".

A typical handshake consists of an initial phase and a final phase, with no intermediate phases. Intermediate phases are only necessary if interim confirmation key material generated during an application phase is desired.

Each application phase consists of ApplicationPayload handshake messages exchanged by client and server to implement applications such as authentication, plus concluding messages for cryptographic confirmation.

All application phases are encrypted. A new master secret and cipher spec are negotiated at the conclusion of each phase, to be applied in the subsequent phase. The master secret and cipher spec negotiated at the conclusion of the final phase are applied to the data exchange following the handshake.

All phases prior to the final phase use PhaseFinished rather than Finished as the concluding message. The final phase concludes with the Finished message.

Application phases may be omitted entirely only when session resumption is used, provided both client and server agree that no application phase is required. The client indicates in its ClientHello whether it is willing to omit application phases in a resumed session.

In each application phase, the client sends the first ApplicationPayload message. ApplicationPayload messages are then traded one at a time between client and server, until the server concludes the phase by sending, in response to an ApplicationPayload message from the client, a ChangeCipherSpec and PhaseFinished sequence to conclude an intermediate phase, or a ChangeCipherSpec and Finished sequence to conclude the final phase. The client then responds with its own ChangeCipherSpec and PhaseFinished sequence, or ChangeCipherSpec and Finished sequence.
Note that the server MUST NOT send a ChangeCipherSpec plus Finished or PhaseFinished message immediately after sending an ApplicationPayload message. It must allow the client to send an ApplicationPayload message prior to concluding the phase. Thus, within any application phase, there will be one more ApplicationPayload message sent by the client than sent by the server.

The server determines which type of concluding message is used, either PhaseFinished or Finished, and the client MUST echo the same type of concluding message. Each PhaseFinished or Finished message provides cryptographic confirmation of the integrity of all handshake messages and keys generated from the start of the handshake through the current phase.

Each ApplicationPayload message contains opaque data interpreted as an AVP (Attribute-Value Pair) sequence. Each AVP in the sequence contains a typed data element. The exchanged AVPs allow client and server to implement "applications" within a secure tunnel. An application may be any procedure that someone may usefully define. A typical application might be authentication; for example, the server may authenticate the client based on password credentials using EAP. Other possible applications include distribution of keys, validating client integrity, setting up IPsec parameters, setting up SSL VPNs, and so on.

The TLS master secret undergoes multiple permutations until a final master secret is computed at the end of the entire handshake. Each phase of the handshake results in a new master secret; the master secret for each phase is confirmed by the PhaseFinished or Finished message exchange that concludes that phase.

The initial master secret is computed during the initial phase of the handshake, using the standard TLS-defined procedure. This initial master secret is confirmed via the first exchange of ChangeCipherSpec and PhaseFinished messages, or, in the case of a resumed session with no subsequence application phase, the exchange of ChangeCipherSpec and Finished messages.

Each subsequent master secret for an application phase is computed using a PRF based on the current master secret, then mixing into the result any session key material generated during authentications during that phase. Each party computes a new master secret prior to the conclusion of each application phase, and uses that new master secret to compute fresh keying material (that is, a TLS "key_block", consisting of client and server MAC secrets, write keys and IVs). The new master secret and keying material become part of the pending read and write connection states. Following standard TLS procedures, these connection states become current states upon sending or receiving ChangeCipherSpec, and are confirmed via the PhaseFinished or Finished message.
The final master secret, computed during the final handshake phase and confirmed by an exchange of ChangeCipherSpec and Finished messages, becomes the actual TLS master secret that defines the session. This final master secret is the surviving master secret, and each prior master secrets SHOULD be discarded when a new connection state is instantiated. The final master secret is used for session resumption, as well as for any session key derivation that protocols defined over TLS may require.

2.2 Message Exchange

Each intermediate handshake phase consists of ApplicationPayload messages sent alternately by client and server, and a concluding exchange of {ChangeCipherSpec, PhaseFinished} messages. The first and last ApplicationPayload message in each intermediate phase is sent by the client; the first {ChangeCipherSpec, PhaseFinished} message sequence is sent by the server. Thus the client begins the exchange with an ApplicationPayload message and the server determines when to conclude it by sending {ChangeCipherSpec, PhaseFinished}. When it receives the server’s {ChangeCipherSpec, PhaseFinished} messages, the client sends its own {ChangeCipherSpec, PhaseFinished} messages, followed by an ApplicationPayload message to begin the next handshake phase.

The final handshake proceeds in the same manner as the intermediate handshake, except that the Finished message is used rather than the PhaseFinished message, and the client does not send an ApplicationPayload message for the next phase because there is no next phase.

At the start of each application handshake phase, the server MUST wait for the client’s opening ApplicationPayload message before it sends its own ApplicationPayload message to the client. The client MAY NOT initiate conclusion of an application handshake phase by sending the first {ChangeCipherSpec, PhaseFinished} or {ChangeCipherSpec, Finished message} sequence; it MUST allow the server to initiate the conclusion of the phase.

2.3 Master Key Permutation

Each permutation of the master secret from one phase to the next begins with the calculation of a preliminary 48 octet vector (pre_vector) based on the current master secret:

\[
\text{pre}_{\text{vector}} = \text{PRF}(\text{SecurityParameters.master_secret,} \\
\quad \text{"inner application preliminary vector"}, \\
\quad \text{SecurityParameters.server_random +} \\
\quad \text{SecurityParameters.client_random}) [0..48];
\]

Session key material generated by applications during the current application phase are mixed into the preliminary vector by
arithmetically adding each session key to it to compute the new master secret. The preliminary vector is treated as a 48-octet integer in big-endian order; that is, the first octet is of the highest significance. Each session key is also treated as a big-endian integer of whatever size it happens to be. Arithmetic carry past the most significant octet is discarded; that is, the addition is performed modulo $2^{384}$.

Thus, the logical procedure for computing the next master secret (which may also be a convenient implementation procedure) is as follows:

1. At the start of each application handshake phase, use the current master secret to compute pre_vector for the next master secret.

2. Each time session key material is generated from an authentication or other exchange, arithmetically add that session key material to pre_vector.

3. At the conclusion of the application handshake phase, copy the current contents of pre_vector (which now includes addition of all session key material) into the master secret, prior to computing verify_data.

Note that the master secret is the only element of the TLS SecurityParameters that is permuted from phase to phase. The client_random, server_random, bulk_cipher_algorithm, mac_algorithm, etc. remain constant throughout all phases of the handshake.

The purpose of using a PRF to compute a preliminary vector is to ensure that, even in the absence of session keys, the master secret is cryptographically distinct in each phase of the handshake.

The purpose of adding session keys into the preliminary vector is to ensure that the same client entity that negotiated the original master secret also negotiated the inner authentication(s). In the absence of such mixing of keys generated from the standard TLS handshake with keys generated from inner authentication, it is possible for a hostile agent to mount a man-in-the-middle attack, acting as server to an unsuspecting client to induce it to perform an authentication with it, which it can then pass through the TLS tunnel to allow it to pose as that client.

An application phase may include no authentications that produce a session key, may include one such authentication, or may include several. Arithmetic addition was chosen as the mixing method because it is commutative, that is, it does not depend on the order of operations. This allows multiple authentications to proceed concurrently if desired, without having to synchronize the order of master secret updates between client and server.
Addition was chosen rather than XOR in order to avoid what is probably a highly unlikely problem; namely, that two separate authentications produce the same session key, which, if XORed, would mutually cancel. This might occur, for example, if two instances of an authentication method were to be applied against different forms of a user identity that turn out in a some cases to devolve to the same identity.

Finally, it was decided that a more complex mixing mechanism for session key material, such as hashing, besides not being commutative, would not provide any additional security, due to the pseudo-random character of the preliminary vector and the powerful PRF function which is applied to create derivative secrets.

2.3.1 Application Session Key Material

Many authentication protocols used today generate session keys that are bound to the authentication. Such keying material is normally intended for use in a subsequent data connection for encryption and validation. For example, EAP-TLS, MS-CHAP-V2 and its alter ego EAP-MS-CHAP-V2 each generate session keys.

Session keying material generated during an application phase MUST be used to permute the TLS/IA master secret between one phase and the next, and MUST NOT be used for any other purpose. Permuting the master secret based on session keying material is necessary to preclude man-in-the-middle attacks, in which an unsuspecting client is induced to perform an authentication outside a tunnel with an attacker posing as a server; the attacker can then introduce the authentication protocol into a tunnel such as provided by TLS/IA, fooling an authentic server into believing that the attacker is the authentic user.

By mixing keying material generated during application phase authentication into the master secret, such attacks are thwarted, since only a single client identity could both authenticate successfully and have derived the session keying material. Note that the keying material generated during authentication must be cryptographically related to the authentication and not derivable from data exchanged during authentication in order for the keying material to be useful in thwarting such attacks.

In addition, the fact that the master secret cryptographically incorporates keying material from application phase authentications provides additional protection when the master secret is used as a basis for generating additional keys for use outside of the TLS exchange. If the master secret did not include keying material from inner authentications, an eavesdropper who somehow knew the server’s private key could, in an RSA-based handshake, determine the master secret and hence would be able to compute the additional keys that are based on it. When inner authentication keying material is
incorporated into the master secret, such an attack becomes impossible.

The RECOMMENDED amount of keying material to mix into the master secret is 32 octets. Up to 48 octets MAY be used.

Each authentication protocol may define how the keying material it generates is mapped to an octet sequence of some length for the purpose of TLS/IA mixing. However, for protocols which do not specify this (including the multitude of protocols that pre-date TLS/IA) the following rules are defined. The first rule that applies SHALL be the method for determining keying material:

- If the authentication protocol maps its keying material to the RADIUS attributes MS-MPPE-Recv-Key and MS-MPPE-Send-Key [RFC2548], then the keying material for those attributes are concatenated (with MS-MPPE-Recv-Key first), the concatenated sequence is truncated to 32 octets if longer, and the result is used as keying material. (Note that this rule applies to MS-CHAP-V2 and EAP-MS-CHAP-V2.)

- If the authentication protocol uses a pseudo-random function to generate keying material, that function is used to generate 32 octets for use as keying material.

2.4 Session Resumption

A TLS/IA initial handshake phase may be resumed using standard mechanisms defined in RFC 2246. When the initial handshake phase is resumed, client and server may not deem it necessary to exchange AVPs in one or more additional application phases, as the resumption itself may provide all the security needed.

The client indicates within the InnerApplication extension whether it requires AVP exchange when session resumption occurs. If it indicates that it does not, then the server may at its option omit subsequent application phases and complete the resumed handshake in a single phase.

Note that RFC 3546 specifically states that when session resumption is used, the server MUST ignore any extensions in the ClientHello. However, it is not possible to comply with this requirement for the Inner Application extension, since even in a resumed session it may be necessary to include application phases, and whether they must be included is negotiated in the extension message itself. Therefore, the RFC 3546 provision is specifically overridden for the single case of the Inner Application extension, which is considered an exception to this rule.
2.5 Error Termination

The TLS/IA handshake may be terminated by either party sending a fatal alert, following standard TLS procedures.

2.6 Computing Verification Data

In standard TLS, the "verify_data" vector of the Finished message is computed as follows:

\[
\text{PRF(master_secret, finished_label, MD5(handshake_messages) + SHA-1(handshake_messages)) [0..11];}
\]

This allows both parties to confirm the master secret as well as the integrity of all handshake messages that have been exchanged.

In TLS/IA, verify_data for the initial handshake phase is computed in exactly the same manner.

In the subsequent application phases, a slight variation of this formula is used. The data that is hashed is the hash of the handshake messages computed in the previous phase plus all handshake messages that have been exchanged since that previous hash was computed. Thus, for each application phase, the MD5 hash input to the PRF is a hash of the MD5 hash computed in the previous phase concatenated with all subsequent handshake messages through the current phase; the SHA-1 hash is computed in the same way, but using the SHA-1 hash computed for the previous phase.

Also, the master secret used in the PRF computation in each application phase is the new master secret generated at the conclusion of that phase.

For clarity, this is best expressed in formal notation.

Let phases be numbered from 0, where phase 0 is the initial phase.

Let:

- \(\text{Secret}[n]\) be the master secret determined at the conclusion of phase \(n\).
- \(\text{Messages}[n]\) be the additional handshake messages exchanged since the hashes were computed in phase \(n-1\), where \(n > 0\); or all handshake messages exchanged to date starting from ClientHello, where \(n = 0\).
- \(\text{MD5}[n]\) be the MD5 hash of handshake message material for phase \(n\).
- \(\text{SHA-1}[n]\) be the SHA-1 hash of handshake message material for phase \(n\).
PRF[n] be the verify_data generated via PRF in phase n.

Hash computations for phase 0 are as follows:

\[ \text{MD5}[0] = \text{MD5}(\text{Messages}[0]) \]
\[ \text{SHA-1}[0] = \text{SHA-1}(\text{Messages}[0]) \]

Hash computations for phase i, where \( i > 0 \) (i.e. application phases) are as follows:

\[ \text{MD5}[i] = \text{MD5}(\text{MD5}[i-1] + \text{Messages}[i]) \]
\[ \text{SHA-1}[i] = \text{SHA-1}(\text{SHA-1}[i-1] + \text{Messages}[i]) \]

The PRF computation to generate verify_data for any phase i (including \( i = 0 \)) is as follows:

\[ \text{PRF}[i] = \text{PRF}(\text{Secret}[i], \text{finished_label}, \text{MD5}[i] + \text{SHA-1}[i]) \]

Note that for phase 0, the PRF computation is identical to the standard TLS computation. Variations to the algorithm occur only in application phases, in the use of new master secrets and the inclusion of hashes of previous handshake messages as input to the hashing algorithms.

During an application phase, the handshake messages input to the hashing algorithm include all handshake messages exchanged since the last PRF computation was performed. This will always include either one or two PhaseFinished messages from the previous phase. To see why, assume that in the previous phase the client issued its PhaseFinished message first, and the server’s PhaseFinished message in response thus included the client’s PhaseFinished message. This means that the server has not yet fed its PhaseFinished message into the PRF, and the client has fed neither its own PhaseFinished message nor the server’s PhaseFinished response message into the PRF. Therefore these messages from the previous phase must be fed into the PhaseFinished messages along with handshake messages from the current phase into the PRF that validates the current phase.

Note that the only handshake messages that appear in an application phase are InnerApplication messages and Finished or Phase Finished messages. ChangeCipherSpec messages are not handshake messages and are therefore never included in the hash computations.

Note also that for TLS/IA, just as for standard TLS, client and server include a somewhat different set of handshake messages in hash computations. Therefore, both client and server must compute two PRFs for each handshake phase: one to include the verify_data
that it transmits, and one to use to check the verify_data received from the other party.

2.7 TLS/IA Messages

All specifications of TLS/IA messages follow the usage defined in RFC 2246.

TLS/IA defines a new TLS extension, two new handshake messages, and a new alert code. The new types and codes are (decimal):

- "InnerApplication" extension type: 37703
- "PhaseFinished" type: 78
- "ApplicationPayload" type: 79
- "InnerApplicationFailure" code: 208

[Note: I have not checked these types yet against types defined in RFCs or drafts. pf]

2.8 Negotiating the Inner Application Extension

Use of the InnerApplication extension follows RFC 3546. The client proposes use of this extension by including the ClientInnerApplication message in the client_hello_extension_list of the extended ClientHello. If this message is included in the ClientHello, the server MAY accept the proposal by including the ServerInnerApplication message in the server_hello_extension_list of the extended ServerHello. If use of this extension is either not proposed by the client or not confirmed by the server, the variations to the TLS handshake described here MUST NOT be used.

2.8.1 ClientInnerApplication

When the client wishes to propose use of the Inner Application extension, it must include ClientInnerApplication in the "extension_data" vector in the Extension structure in its extended ClientHello message, where:

```c
enum {
    not_required(0), required(1), (255)
} AppPhaseOnResumption;

struct {
    AppPhaseOnResumption app_phase_on_resumption;
} ClientInnerApplication;
```

The AppPhaseOnResumption enumeration allow client and server to negotiate an abbreviated, single-phase handshake when session
resumption is employed. If the server is able to resume a previous session, and if the client sets app_phase_on_resumption to not_required, then the server MAY conclude the initial handshake phase with a Finished message, thus completing the handshake in a single phase. If the client sets app_phase_on_resumption to required, then the server MUST conclude the initial handshake phase with PhaseFinished, thus allowing one or more subsequent application phases to follow the initial handshake phase.

The value of app_phase_on_resumption applies to the current handshake only. For example, it is possible for app_phase_on_resumption to have different values in two handshakes that are both resumed from the same original TLS session.

Note that the server may initiate one or more application phases even if the client sets app_phase_on_resumption to not_required, as the server itself may have reason to proceed with one or more application phases.

Note also that if session resumption does not occur, the app_phase_on_resumption variable is ignored, the server MUST conclude the initial phase with a PhaseFinished message and one or more application phases MUST follow the initial handshake phase.

2.8.2 ServerInnerApplication

When the server wishes to confirm use of the Inner Application extension that has been proposed by the client, it must include ServerInnerApplication in the "extension_data" vector in the Extension structure in its extended ServerHello message, where:

struct {
    } ServerInnerApplication;

Note that the ServerInnerApplication message contains no data; however, it’s presence is required to confirm use of the Inner Application extension when proposed by the client.

If the client set app_phase_on_resumption to not_required and the server agrees and will not initiate an application phase, the server MUST NOT include ServerInnerApplication in its ServerHello and it must conclude the initial (and only) handshake phase with the Finished message. If, the server includes ServerInnerApplication, it MUST conclude the initial handshake phase with PhaseFinished, indicating that one or more application phases will follow the initial handshake phase.
2.9 The PhaseFinished Handshake Message

The PhaseFinished message concludes all handshake phases prior to the final handshake phase. It MUST be immediately preceded by a ChangeCipherSpec message. It is defined as follows:

```c
struct {
    opaque verify_data[12];
} PhaseFinished;
```

2.10 The ApplicationPayload Handshake Message

The ApplicationPayload message carries an AVP sequence during an application handshake phase. It is defined as follows:

```c
struct {
    opaque avps[Handshake.length];
} ApplicationPayload;
```

where Handshake.length is the 24-bit length field in the encapsulating Handshake message.

Note that the "avps" element has its length defined in square bracket rather than angle bracket notation, implying a fixed rather than variable length vector. This avoids the having the length of the AVP sequence specified redundantly both in the encapsulating Handshake message and as a length prefix in the avps element itself.

2.11 The InnerApplicationFailure Alert

An InnerApplicationFailure error alert may be sent by either party during an application phase. This indicates that the sending party considers the negotiation to have failed due to an application carried in the AVP sequences, for example, a failed authentication.

The AlertLevel for an InnerApplicationFailure alert MUST be set to "fatal".

Note that other alerts are possible during an application phase; for example, decrypt_error. The InnerApplicationFailure alert relates specifically to the failure of an application implemented via AVP sequences; for example, failure of an EAP or other authentication method, or information passed within the AVP sequence that is found unsatisfactory.

3 Encapsulation of AVPs within ApplicationPayload Messages

During application phases of the TLS handshake, information is exchanged between client and server through the use of attribute-value pairs (AVPs). This data is encrypted using the then-current cipher state established during the preceding handshake phase.
The AVP format chosen for TLS/IA is compatible with the Diameter AVP format. This does not in any way represent a requirement that Diameter be supported by any of the devices or servers participating in the TLS/IA conversation, whether directly as client or server or indirectly as a backend authenticator. Use of this format is merely a convenience. Diameter is a superset of RADIUS and includes the RADIUS attribute namespace by definition, though it does not limit the size of an AVP as does RADIUS. RADIUS, in turn, is a widely deployed AAA protocol and attribute definitions exist for all commonly used password authentication protocols, including EAP.

Thus, Diameter is not considered normative except as specified in this document. Specifically, the AVP Codes used in TLS/IA are semantically equivalent to those defined for Diameter, and, by extension, RADIUS.

Use of the RADIUS/Diameter namespace allows a TLS/IA server to easily translate between AVPs it uses to communicate with clients and the protocol requirements of AAA servers that are widely deployed. Plus, it provides a well-understood mechanism to allow vendors to extend that namespace for their particular requirements.

3.1 AVP Format

The format of an AVP is shown below. All items are in network, or big-endian, order; that is, they have most significant octet first.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           AVP Code                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V M r r r r r|                  AVP Length                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Vendor-ID (opt)                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                Data ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

AVP Code

The AVP Code is four octets and, combined with the Vendor-ID field if present, identifies the attribute uniquely. The first 256 AVP numbers represent attributes defined in RADIUS. AVP numbers 256 and above are defined in Diameter.

AVP Flags

The AVP Flags field is one octet, and provides the receiver with information necessary to interpret the AVP.
The 'V' (Vendor-Specific) bit indicates whether the optional Vendor-ID field is present. When set to 1, the Vendor-ID field is present and the AVP Code is interpreted according to the namespace defined by the vendor indicated in the Vendor-ID field.

The 'M' (Mandatory) bit indicates whether support of the AVP is required. If this bit is set to 0, this indicates that the AVP may be safely ignored if the receiving party does not understand or support it. If set to 1, this indicates that the receiving party must fail the negotiation if it does not understand the AVP; for a server, this would imply returning EAP-Failure, for a client, this would imply abandoning the negotiation.

The 'r' (reserved) bits are unused and must be set to 0.

**AVP Length**

The AVP Length field is three octets, and indicates the length of this AVP including the AVP Code, AVP Length, AVP Flags, Vendor-ID (if present) and Data.

**Vendor-ID**

The Vendor-ID field is present if and only if the 'V' bit is set in the AVP Flags field. It is four octets, and contains the vendor’s IANA-assigned "SMI Network Management Private Enterprise Codes" [RFC1700] value. Vendors defining their own AVPs must maintain a consistent namespace for use of those AVPs within RADIUS, Diameter and TLS/IA.

A Vendor-ID value of zero is semantically equivalent to absence of the Vendor-ID field altogether.

### 3.2 AVP Sequences

Data encapsulated within the TLS Record Layer must consist entirely of a sequence of zero or more AVPs. Each AVP must begin on a 4-octet boundary relative to the first AVP in the sequence. If an AVP is not a multiple of 4 octets, it must be padded with 0s to the next 4-octet boundary.

Note that the AVP Length does not include the padding.

### 3.3 Guidelines for Maximum Compatibility with AAA Servers

When maximum compatibility with AAA servers is desired, the following guidelines for AVP usage are suggested:

- Non-vendor-specific AVPs should be selected from the set of attributes defined for RADIUS; that is, attributes with codes less than 256. This provides compatibility with both RADIUS and
Diameter.

- Vendor-specific AVPs should be defined in terms of RADIUS. Vendor-specific RADIUS attributes translate to Diameter automatically; the reverse is not true. RADIUS vendor-specific attributes use RADIUS attribute 26 and include vendor ID, vendor-specific attribute code and length; see [RFC2865] for details.

4 Tunneled Authentication within Application Phases

TLS/IA permits user authentication information to be tunneled within an application phase between client and server, protecting the security of the authentication information against active and passive attack.

Any type of password or other authentication may be tunneled. Also, multiple tunneled authentications may be performed. Normally, tunneled authentication is used when the client has not been issued a certificate and the TLS handshake provides only one-way authentication of the server to the client; however, in certain cases it may be desired to perform certificate authentication of the client during the initial handshake phase as well as tunneled user authentication in a subsequent application phase.

This section establishes rules for using common authentication mechanisms within TLS/IA. Any new authentication mechanism should in general be covered by these rules if it is defined as an EAP type. Authentication mechanisms whose use within TLS/IA is not covered within this specification may require separate standardization, preferably within the standard that describes the authentication mechanism in question.

4.1 Implicit challenge

Certain authentication protocols that use a challenge/response mechanism rely on challenge material that is not generated by the authentication server, and therefore require special handling.

In PPP protocols such CHAP, MS-CHAP and MS-CHAP-V2, for example, the Network Access Server (NAS) issues a challenge to the client, the client then hashes the challenge with the password and forwards the response to the NAS. The NAS then forwards both challenge and response to a AAA server. But because the AAA server did not itself generate the challenge, such protocols are susceptible to replay attack.

If the client were able to create both challenge and response, anyone able to observe a CHAP or MS-CHAP exchange could pose as that user by replaying that challenge and response into a TLS/IA conversation.
To make these protocols secure in TLS/IA, it is necessary to provide a mechanism to produce a challenge that the client cannot control or predict.

When a challenge-based authentication mechanism is used, both client and server use the TLS PRF function to generate as many octets as are required for the challenge, using the constant string "inner application challenge", based on the then-current master secret and random values established during the initial handshake phase:

\[
\text{IA\_challenge} = \text{PRF(\text{SecurityParameters.master\_secret, }}
\]
\[
\text{\quad "inner application challenge",}
\]
\[
\text{\quad \text{SecurityParameters.server\_random +}}
\]
\[
\text{\quad \text{SecurityParameters.client\_random);}}
\]

4.2 Tunneled Authentication Protocols

This section describes the rules for tunneling specific authentication protocols within TLS/IA.

For each protocol, the RADIUS RFC that defines the relevant attribute formats is cited. Note that these attributes are encapsulated as described in section 3.1; that is, as Diameter attributes, not as RADIUS attributes. In other words, the AVP Code, Length, Flags and optional Vendor-ID are formatted as described in section 3.1, while the Data is formatted as described by the cited RADIUS RFC.

All tunneled authentication protocols except EAP must be initiated by the client in the first ApplicationPayload message of an application phase. EAP may be initiated by the client in the first ApplicationPayload message of an application phase; it may also be initiated by the server in any ApplicationPayload message.

The authentication protocols described below may be performed directly by the TLS/IA server or may be forwarded to a backend AAA server. For authentication protocols that generate session keys, the backend server must return those session keys to the TLS/IA server in order to allow the protocol to succeed within TLS/IA. RADIUS or Diameter servers are suitable backend AAA servers for this purpose. RADIUS servers typically return session keys in MS-MPPE-Recv-Key and MS-MPPE-Send-Key attributes [RFC2548]; Diameter servers return session keys in the EAP-Master-Session-Key AVP [AAA-EAP].

4.2.1 EAP

EAP is described in [RFC3784]; RADIUS attribute formats are described in [RFC3579].
When EAP is the tunneled authentication protocol, each tunneled EAP packet between the client and server is encapsulated in an EAP-Message AVP.

Either client or server may initiate EAP.

The client is the first to transmit within any application phase, and it may include an EAP-Response/Identity AVP in its ApplicationPayload message to begin an EAP conversation. Alternatively, if the client does not initiate EAP the server may, by including an EAP-Request/Identity AVP in its ApplicationPayload message.

The client’s EAP-Response/Identity provides the actual username; the privacy of the user’s identity is now guaranteed by the TLS encryption. This username must be a Network Access Identifier (NAI) [RFC2486]; that is, it must be in the following format:

    username@realm

The @realm portion is optional, and is used to allow the server to forward the EAP message sequence to the appropriate server in the AAA infrastructure when necessary.

The EAP authentication between client and server proceeds normally, as described in [RFC3784]. However, upon completion the server does not send an EAP-Success or EAP-Failure AVP. Instead, the server signals success when it concludes the application phase by issuing a Finished or PhaseFinished message, or it signals failure by issuing an InnerApplicationFailure alert.

Note that the client may also issue an InnerApplicationFailure alert, for example, when authentication of the server fails in a method providing mutual authentication.

4.2.2 CHAP

The CHAP algorithm is described in [RFC1994]; RADIUS attribute formats are described in [RFC2865].

Both client and server generate 17 octets of challenge material, using the constant string "inner application challenge" as described above. These octets are used as follows:

    CHAP-Challenge [16 octets]
    CHAP Identifier  [1 octet]

The client initiates CHAP by including User-Name, CHAP-Challenge and CHAP-Password AVPs in the first ApplicationPayload message in any application phase. The CHAP-Challenge value is taken from the challenge material. The CHAP-Password consists of CHAP Identifier,
taken from the challenge material; and CHAP response, computed according to the CHAP algorithm.

Upon receipt of these AVPs from the client, the server must verify that the value of the CHAP-Challenge AVP and the value of the CHAP Identifier in the CHAP-Password AVP are equal to the values generated as challenge material. If either item does not match exactly, the server must reject the client. Otherwise, it validates the CHAP-Challenge to determine the result of the authentication.

4.2.3 MS-CHAP

The MS-CHAP algorithm is described in [RFC2433]; RADIUS attribute formats are described in [RFC2548].

Both client and server generate 9 octets of challenge material, using the constant string "inner application challenge" as described above. These octets are used as follows:

MS-CHAP-Challenge [8 octets]  
Ident              [1 octet]

The client initiates MS-CHAP by including User-Name, MS-CHAP-Challenge and MS-CHAP-Response AVPs in the first ApplicationPayload message in any application phase. The MS-CHAP-Challenge value is taken from the challenge material. The MS-CHAP-Response consists of Ident, taken from the challenge material; Flags, set according the client preferences; and LM-Response and NT-Response, computed according to the MS-CHAP algorithm.

Upon receipt of these AVPs from the client, the server must verify that the value of the MS-CHAP-Challenge AVP and the value of the Ident in the client’s MS-CHAP-Response AVP are equal to the values generated as challenge material. If either item does not match exactly, the server must reject the client. Otherwise, it validates the MS-CHAP-Challenge to determine the result of the authentication.

4.2.4 MS-CHAP-V2

The MS-CHAP-V2 algorithm is described in [RFC2759]; RADIUS attribute formats are described in [RFC2548].

Both client and server generate 17 octets of challenge material, using the constant string "inner application challenge" as described above. These octets are used as follows:

MS-CHAP-Challenge [16 octets]  
Ident              [1 octet]

The client initiates MS-CHAP-V2 by including User-Name, MS-CHAP-Challenge and MS-CHAP2-Response AVPs in the first ApplicationPayload message in any application phase. The MS-CHAP-Challenge value is taken from the challenge material. The MS-CHAP2-Response consists of Ident, taken from the challenge material; Flags, set according the client preferences; and LM-Response and NT-Response, computed according to the MS-CHAP-V2 algorithm.

Upon receipt of these AVPs from the client, the server must verify that the value of the MS-CHAP-Challenge AVP and the value of the Ident in the client’s MS-CHAP2-Response AVP are equal to the values generated as challenge material. If either item does not match exactly, the server must reject the client. Otherwise, it validates the MS-CHAP-Challenge to determine the result of the authentication.
message in any application phase. The MS-CHAP-Challenge value is taken from the challenge material. The MS-CHAP2-Response consists of Ident, taken from the challenge material; Flags, set to 0; Peer-Challenge, set to a random value; and Response, computed according to the MS-CHAP-V2 algorithm.

Upon receipt of these AVPs from the client, the server must verify that the value of the MS-CHAP-Challenge AVP and the value of the Ident in the client’s MS-CHAP2-Response AVP are equal to the values generated as challenge material. If either item does not match exactly, the server must reject the client. Otherwise, it validates the MS-CHAP2-Challenge.

If the MS-CHAP2-Challenge received from the client is correct, the server tunnels the MS-CHAP2-Success AVP to the client.

Upon receipt of the MS-CHAP2-Success AVP, the client is able to authenticate the server. In its next InnerApplicationPayload message to the server, the client does not include any MS-CHAP-V2 AVPs. (This may result in an empty InnerApplicationPayload if no other AVPs need to be sent.)

If the MS-CHAP2-Challenge received from the client is not correct, the server tunnels an MS-CHAP2-Error AVP to the client. This AVP contains a new Ident and a string with additional information such as error reason and whether a retry is allowed. If the error reason is an expired password and a retry is allowed, the client may proceed to change the user’s password. If the error reason is not an expired password or if the client does not wish to change the user’s password, it issues an InnerApplicationFailure alert.

If the client does wish to change the password, it tunnels MS-CHAP-NT-Enc-PW, MS-CHAP2-CPW, and MS-CHAP-Challenge AVPs to the server. The MS-CHAP2-CPW AVP is derived from the new Ident and Challenge received in the MS-CHAP2-Error AVP. The MS-CHAP-Challenge AVP simply echoes the new Challenge.

Upon receipt of these AVPs from the client, the server must verify that the value of the MS-CHAP-Challenge AVP and the value of the Ident in the client’s MS-CHAP2-CPW AVP match the values it sent in the MS-CHAP2-Error AVP. If either item does not match exactly, the server must reject the client. Otherwise, it validates the MS-CHAP2-CPW AVP.

If the MS-CHAP2-CPW AVP received from the client is correct, and the server is able to change the user’s password, the server tunnels the MS-CHAP2-Success AVP to the client and the negotiation proceeds as described above.
Note that additional AVPs associated with MS-CHAP-V2 may be sent by the server; for example, MS-CHAP-Domain. The server must tunnel such authentication-related AVPs along with the MS-CHAP2-Success.

4.2.5 PAP

PAP RADIUS attribute formats are described in [RFC2865].

The client initiates PAP by including User-Name and User-Password AVPs in the first ApplicationPayload message in any application phase.

In RADIUS, User-Password is padded with nulls to a multiple of 16 octets, then encrypted using a shared secret and other packet information.

A TLS/IA, however, does not RADIUS-encrypt the password since all application phase data is already encrypted. The client SHOULD, however, null-pad the password to a multiple of 16 octets, to obfuscate its length.

Upon receipt of these AVPs from the client, the server may be able to decide whether to authenticate the client immediately, or it may need to challenge the client for more information.

If the server wishes to issue a challenge to the client, it MUST tunnel the Reply-Message AVP to the client; this AVP normally contains a challenge prompt of some kind. It may also tunnel additional AVPs if necessary, such the Prompt AVP. Upon receipt of the Reply-Message AVPs, the client tunnels User-Name and User-Password AVPs again, with the User-Password AVP containing new information in response to the challenge. This process continues until the server determines the authentication has succeeded or failed.

4.3 Performing Multiple Authentications

In some cases, it is desirable to perform multiple user authentications. For example, a AAA/H may want first to authenticate the user by password, then by token card.

The server may perform any number of additional user authentications using EAP, simply by issuing a EAP-Request with a new protocol type once the previous authentication has completed.

For example, a server wishing to perform MD5-Challenge followed by Generic Token Card would first issue an EAP-Request/MD5-Challenge AVP and receive a response. If the response is satisfactory, it would then issue EAP-Request/Generic Token Card AVP and receive a response. If that response were also satisfactory, it would consider the user authenticated.
5 Example Message Sequences

This section presents a variety of possible TLS/IA message sequences. These examples do not attempt to exhaustively depict all possible scenarios.

Parentheses indicate optional TLS messages. Brackets indicate optional message exchanges. Ellipsis (.) indicates optional repetition of preceding messages.

5.1 Full Initial Handshake with Intermediate and Final Application Phases

The diagram below depicts a full initial handshake phase followed by two application phases.

Note that the client concludes the intermediate phase and starts the final phase in an uninterrupted sequence of three messages: ChangeCipherSpec and PhaseFinished belong to the intermediate phase, and ApplicationPayload belongs to the final phase.

Client                                               Server
------                                               ------

*** Initial Phase:              ServerHello
ClientHello                  -------->                        (Certificate)
                             (Certificate)                ServerKeyExchange
                             (CertificateRequest)  <--------      ServerHelloDone

ClientKeyExchange
(CertificateVerify)
ChangeCipherSpec
PhaseFinished             -------->                        ChangeCipherSpec
                            <--------        PhaseFinished

[ ApplicationPayload          -------->
  [
    ApplicationPayload          -------->
      ...
  ]
                             ChangeCipherSpec
                             <--------        PhaseFinished

Client
------
5.2 Resumed Session with Single Application Phase

The diagram below depicts a resumed session followed by a single application phase.

Note that the client concludes the initial phase and starts the final phase in an uninterrupted sequence of three messages: ChangeCipherSpec and PhaseFinished belong to the initial phase, and ApplicationPayload belongs to the final phase.

```plaintext
Client                                               Server
------                                               -----

*** Initial Phase:
ClientHello                  -------->               ServerHello
ChangeCipherSpec
PhaseFinished

*** Final Phase:
ApplicationPayload           -------->

[                              
[                             

----------  ApplicationPayload

----------  ApplicationPayload

...                       ...

]                              ]

----------  ChangeCipherSpec
Finished

----------  ChangeCipherSpec
Finished

```
5.3 Resumed Session with No Application Phase

The diagram below depicts a resumed session without any subsequent application phase. This will occur if the client indicates in its ClientInnerApplication message that no application phase is required and the server concurs.

Note that this message sequence is identical to that of a standard TLS resumed session.

```
Client                                               Server
------                                               ------
*** Initial/Final Phase:                             ServerHello
       ClientHello                   -------->
       ChangeCipherSpec
       Finished
       ChangeCipherSpec
       Finished
```

6 Security Considerations

This document introduces a new TLS extension called "Inner Application". When TLS is used with the Inner Application extension (TLS/IA), additional messages are exchanged during the TLS handshake. Hence a number of security issues need to be taken into consideration. Since the security heavily depends on the information (called "applications") which are exchanged between the TLS client and the TLS server as part of the TLS/IA extension we try to classify them into two categories: The first category considers the case where the exchange results in the generation of keying material. This is, for example, the case with many EAP methods. EAP is one of the envisioned main "applications". The second category focuses on cases where no session key is generated. The security treatment of the latter category is discouraged since it is vulnerability to man-in-the-middle attacks if the two sessions cannot be bound to each other as shown in [MITM].

Subsequently, we investigate a number of security issues:

- Architecture and Trust Model

For many of the use cases in this document we assume that three functional entities participate in the protocol exchange: TLS client, TLS server and a AAA infrastructure (typically consisting of a AAA server and possibly a AAA broker). The protocol exchange described in this document takes place between the TLS client and the TLS server. The interaction between the AAA client (which corresponds to the TLS server) and the AAA server is described in
the respective AAA protocol documents and therefore outside the scope of this document. The trust model behind this architecture with respect to the authentication, authorization, session key establishment and key transport within the AAA infrastructure is discussed in [KEYING].

- Authentication

This document assumes that the TLS server is authenticated to the TLS client as part of the authentication procedure of the initial TLS Handshake. This approach is similar to the one chosen with the EAP support in IKEv2 (see [IKEv2]). Typically, public key based server authentication is used for this purpose. More interesting is the client authentication property whereby information exchanged as part of the Inner Application is used to authenticate (or authorize) the client. For example, if EAP is used as an inner application then EAP methods are used to perform authentication and key agreement between the EAP peer (most likely the TLS client) and the EAP server (i.e., AAA server).

- Authorization

Throughout this document it is assumed that the TLS server can be authorized by the TLS client as a legitimate server as part of the authentication procedure of the initial TLS Handshake. The entity acting as TLS client can be authorized either by the TLS server or by the AAA server (if the authorization decision is offloaded). Typically, the authenticated identity is used to compute the authorization decision but credential-based authorization mechanisms may be used as well.

- Man-in-the-Middle Attack

Man-in-the-middle attacks have become a concern with tunneled authentication protocols because of the discovered vulnerabilities (see [MITM]) of a missing cryptographic binding between the independent protocol sessions. This document also proposes a tunneling protocol, namely individual inner application sessions are tunneled within a previously executed session. The first protocol session in this exchange is the initial TLS Handshake. To avoid man-in-the-middle attacks a number of sections address how to establish such a cryptographic binding (see Section 2.3 and 2.6).

- User Identity Confidentiality

The TLS/IA extension allows splitting the authentication of the TLS server from the TLS client into two separate sessions. As one of the advantages, this provides active user identity confidentiality since the TLS client is able to authenticate the TLS server and to establish a unilateral authenticated and
confidentiality-protected channel prior to starting the client-side authentication.

- Session Key Establishment

TLS [RFC2246] defines how session key material produced during the TLS Handshake is generated with the help of a pseudo-random function to expand it to keying material of the desired length for later usage in the TLS Record Layer. Section 2.3 gives some guidelines with regard to the master key generation. Since the TLS/IA extension supports multiple exchanges whereby each phase concludes with a generated keying material. In addition to the keying material established as part of TLS itself, most inner applications will produce their keying material. For example, keying material established as part of an EAP method must be carried from the AAA server to the AAA client. Details are subject to the specific AAA protocol (for example, EAP usage in Diameter [AAA-EAP]).

- Denial of Service Attacks

This document does not modify the initial TLS Handshake and as such, does not introduce new vulnerabilities with regard to DoS attacks. Since the TLS/IA extension allows to postpone the client-side authentication to a later stage in the protocol phase. As such, it allows malicious TLS clients to initiate a number of exchanges while remaining anonymous. As a consequence, state at the server is allocated and computational efforts are required at the server side. Since the TLS client cannot be stateless this is not strictly a DoS attack.

- Confidentiality Protection and Dictionary Attack Resistance

Similar to the user identity confidentiality property the usage of the TLS/IA extension allows to establish a unilateral authenticated tunnel which is confidentiality protected. This tunnel protects the inner application information elements to be protected against active adversaries and therefore provides resistance against dictionary attacks when password-based authentication protocols are used inside the tunnel. In general, information exchanged inside the tunnel experiences confidentiality protection.

- Downgrading Attacks

This document defines a new extension. The TLS client and the TLS server indicate the capability to support the TLS/IA extension as part of the client_hello_extension_list and the server_hello_extension_list payload. More details can be found in Section 2.8. To avoid downgrading attacks whereby an adversary
removes a capability from the list is avoided by the usage of the Finish or PhaseFinished message as described in Section 2.6.

7 References

7.1 Normative References


7.2 Informative References


8 Authors’ Addresses

Questions about this memo can be directed to:

Paul Funk
Funk Software, Inc.
222 Third Street
Cambridge, MA 02142
9 Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.
Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The Internet Society (2001 - 2004). This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.