    A GSS-API Mechanism for the Extensible Authentication Protocol

Abstract

   This document defines protocols, procedures, and conventions to be
   employed by peers implementing the Generic Security Service
   Application Program Interface (GSS-API) when using the Extensible
   Authentication Protocol mechanism. Through the GS2 family of
   mechanisms defined in RFC 5801, these protocols also define how
   Simple Authentication and Security Layer (SASL) applications use the
   Extensible Authentication Protocol.

Status of This Memo

   This is an Internet Standards Track document.

   This document is a product of the Internet Engineering Task Force
   (IETF). It represents the consensus of the IETF community. It has
   received public review and has been approved for publication by the
   Internet Engineering Steering Group (IESG). Further information on
   Internet Standards is available in Section 2 of RFC 5741.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc7055.

Table of Contents

1. Introduction

   The Application Bridging for Federated Access Beyond Web (ABFAB)
   document [ABFAB-ARCH] describes an architecture for providing
   federated access management to applications using the Generic
   Security Service Application Programming Interface (GSS-API)
   [RFC2743] and Simple Authentication and Security Layer (SASL)
   [RFC4422]. This specification provides the core mechanism for
   bringing federated authentication to these applications.

   The Extensible Authentication Protocol (EAP) [RFC3748] defines a
   framework for authenticating a network access client and server in
   order to gain access to a network. A variety of different EAP
   methods are in wide use; one of EAP's strengths is that for most
   types of credentials in common use, there is an EAP method that
   permits the credential to be used.

   EAP is often used in conjunction with a backend Authentication,
   Authorization and Accounting (AAA) server via RADIUS [RFC3579] or
   Diameter [RFC4072]. In this mode, the Network Access Server (NAS)
   simply tunnels EAP packets over the backend authentication protocol
   to a home EAP/AAA server for the client. After EAP succeeds, the
   backend authentication protocol is used to communicate key material
   to the NAS. In this mode, the NAS need not be aware of or have any
   specific support for the EAP method used between the client and the
   home EAP server. The client and EAP server share a credential that
   depends on the EAP method; the NAS and AAA server share a credential
   based on the backend authentication protocol in use. The backend
   authentication server acts as a trusted third party, enabling network
   access even though the client and NAS may not actually share any
   common authentication methods. As described in the architecture
   document [ABFAB-ARCH], using AAA proxies, this mode can be extended
   beyond one organization to provide federated authentication for
   network access.

   The GSS-API provides a generic framework for applications to use
   security services including authentication and per-message data
   security. Between protocols that support GSS-API directly or
   protocols that support SASL [RFC4422], many application protocols can
   use GSS-API for security services. However, with the exception of
   Kerberos [RFC4121], few GSS-API mechanisms are in wide use on the
   Internet. While GSS-API permits an application to be written
   independent of the specific GSS-API mechanism in use, there is no
   facility to separate the server from the implementation of the
   mechanism as there is with EAP and backend authentication servers.

The goal of this specification is to combine GSS-API's support for
application protocols with EAP/AAA's support for common credential
types and for authenticating to a server without requiring that
server to specifically support the authentication method in use. In
addition, this specification supports the architectural goal of
transporting attributes about subjects to relying parties. Together
this combination will provide federated authentication and
authorization for GSS-API applications. This specification meets the
applicability requirements for EAP to application authentication
[RFC7057].

This mechanism is a GSS-API mechanism that encapsulates an EAP
conversation. From the perspective of RFC 3748, this specification
defines a new lower-layer protocol for EAP. From the perspective of
the application, this specification defines a new GSS-API mechanism.

Section 1.3 of [RFC5247] outlines the typical conversation between
EAP peers where an EAP key is derived:

Phase 0: Discovery
Phase 1: Authentication
        1a: EAP authentication
        1b: AAA Key Transport (optional)
Phase 2: Secure Association Protocol
        2a: Unicast Secure Association
        2b: Multicast Secure Association (optional)

1.1. Discovery

   GSS-API peers discover each other and discover support for GSS-API in
   an application-dependent mechanism. SASL [RFC4422] describes how
   discovery of a particular SASL mechanism such as a GSS-API EAP
   mechanism is conducted. The Simple and Protected Negotiation
   mechanism (SPNEGO) [RFC4178] provides another approach for
   discovering what GSS-API mechanisms are available. The specific
   approach used for discovery is out of scope for this mechanism.

1.2. Authentication

   GSS-API authenticates a party called the "GSS-API initiator" to the
   GSS-API acceptor, optionally providing authentication of the acceptor
   to the initiator. Authentication starts with a mechanism-specific
   message called a "context token" sent from the initiator to the
   acceptor. The acceptor responds, followed by the initiator, and so
   on until authentication succeeds or fails. GSS-API context tokens
   are reliably delivered by the application using GSS-API. The
   application is responsible for in-order delivery and retransmission.

EAP authenticates a party called a "peer" to a party called the "EAP
server". A third party called an "EAP pass-through authenticator"
may decapsulate EAP messages from a lower layer and re-encapsulate
them into a AAA protocol. The term EAP authenticator refers to
whichever of the pass-through authenticator or EAP server receives
the lower-layer EAP packets. The first EAP message travels from the
authenticator to the peer; a GSS-API message is sent from the
initiator to acceptor to prompt the authenticator to send the first
EAP message. The EAP peer maps onto the GSS-API initiator. The role
of the GSS-API acceptor is split between the EAP authenticator and
the EAP server. When these two entities are combined, the division
resembles GSS-API acceptors in other mechanisms. When a more typical
deployment is used and there is a pass-through authenticator, most
context establishment takes place on the EAP server and per-message
operations take place on the authenticator. EAP messages from the
peer to the authenticator are called responses; messages from the
authenticator to the peer are called requests.

Because GSS-API applications provide guaranteed delivery of context
tokens, the EAP retransmission timeout MUST be infinite and the EAP
layer MUST NOT retransmit a message.

This specification permits a GSS-API acceptor to hand off the
processing of the EAP packets to a remote EAP server by using AAA
protocols such as RADIUS, Transport Layer Security (TLS) Encryption
thereof [RFC6929], or Diameter. In this case, the GSS-API acceptor
acts as an EAP pass-through authenticator. The pass-through
authenticator is responsible for retransmitting AAA messages if a
response is not received from the AAA server. If a response cannot
be received, then the authenticator generates an error at the GSS-API
level. If EAP authentication is successful, and where the chosen EAP
method supports key derivation, EAP keying material may also be
derived. If a AAA protocol is used, this can also be used to
replicate the EAP Key from the EAP server to the EAP authenticator.

See Section 5 for details of the authentication exchange.

1.3. Secure Association Protocol

   After authentication succeeds, GSS-API provides a number of per-
   message security services that can be used:

      GSS_Wrap() provides integrity and optional confidentiality for a
      message.

      GSS_GetMIC() provides integrity protection for data sent
      independently of the GSS-API

      GSS_Pseudo_random [RFC4401] provides key derivation functionality.

   These services perform a function similar to secure association
   protocols in network access. Like secure association protocols,
   these services need to be performed near the authenticator/acceptor
   even when a AAA protocol is used to separate the authenticator from
   the EAP server. The key used for these per-message services is
   derived from the EAP key; the EAP peer and authenticator derive this
   key as a result of a successful EAP authentication. In the case that
   the EAP authenticator is acting as a pass-through, it obtains it via
   the AAA protocol. See Section 6 for details.

2. Requirements Notation

   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 [RFC2119].

3. EAP Channel Binding and Naming

   EAP authenticates a user to a realm. The peer knows that it has
   exchanged authentication with an EAP server in a given realm. Today,
   the peer does not typically know which NAS it is talking to securely.
   That is often fine for network access. However, privileges to
   delegate to a chat server seem very different than privileges for a
   file server or trading site. Also, an EAP peer knows the identity of
   the home realm, but perhaps not even the visited realm.

   In contrast, GSS-API takes a name for both the initiator and acceptor
   as inputs to the authentication process. When mutual authentication
   is used, both parties are authenticated. The granularity of these
   names is somewhat mechanism dependent. In the case of the Kerberos
   mechanism, the acceptor name typically identifies both the protocol
   in use (such as IMAP) and the specific instance of the service being
   connected to. The acceptor name almost always identifies the
   administrative domain providing service.

A GSS-API EAP mechanism needs to provide GSS-API naming semantics in
order to work with existing GSS-API applications. EAP channel
binding [RFC6677] is used to provide GSS-API naming semantics.
Channel binding sends a set of attributes from the peer to the EAP
server either as part of the EAP conversation or as part of a secure
association protocol. In addition, attributes are sent in the
backend authentication protocol from the authenticator to the EAP
server. The EAP server confirms the consistency of these attributes.
Confirming attribute consistency also involves checking consistency
against a local policy database as discussed in Section 3.5. In
particular, the peer sends the name of the acceptor it is
authenticating to as part of channel binding. The acceptor sends its
full name as part of the backend authentication protocol. The EAP
server confirms consistency of the names.

EAP channel binding is easily confused with a facility in GSS-API
also called "channel binding". GSS-API channel binding provides
protection against man-in-the-middle attacks when GSS-API is used as
authentication inside some tunnel; it is similar to a facility called
"cryptographic binding" in EAP. See [RFC5056] for a discussion of
the differences between these two facilities and Section 6.1 for how
GSS-API channel binding is handled in this mechanism.

3.1. Mechanism Name Format

Before discussing how the initiator and acceptor names are validated
in the AAA infrastructure, it is necessary to discuss what composes a
name for an EAP GSS-API mechanism. GSS-API permits several types of
generic names to be imported using GSS_Import_name(). Once a
mechanism is chosen, these names are converted into a mechanism-
specific name called a "Mechanism Name". Note that a Mechanism Name
is the name of an initiator or acceptor, not of a GSS-API mechanism.
This section first discusses the mechanism name form and then
discusses what name forms are supported.

The string representation of the GSS-EAP mechanism name has the
following ABNF [RFC5234] representation:

```
char-normal = %x00-2E/%x30-3F/%x41-5B/%x5D-FF
char-escaped = "\" %x2F / "\" %x40 / "\" %x5C
name-char = char-normal / char-escaped
name-string = 1*name-char
user-or-service = name-string
host = [name-string]
realm = name-string
service-specific = name-string
service-specifics = service-specific 0*("/" service-specifics)
name = user-or-service ["/" host [ "/" service-specifics]] [ "@"
         realm ]
```

Special characters appearing in a name can be backslash escaped to
avoid their special meanings. For example, "\\" represents a literal
backslash. This escaping mechanism is a property of the string
representation; if the components of a name are transported in some
mechanism that will keep them separate without backslash escaping,
then backslash SHOULD have no special meaning.

The user-or-service component is similar to the portion of a network
access identifier (NAI) before the '@' symbol for initiator names and
the service name from the registry of GSS-API host-based services in
the case of acceptor names [GSS-IANA]. The NAI specification
provides rules for encoding and string preparation in order to
support internationalization of NAIs; implementations of this
mechanism MUST NOT prepare the user-or-service according to these
rules; see Section 3.2 for internationalization of this mechanism.
The host portion is empty for initiators and typically contains the
domain name of the system on which an acceptor service is running.
Some services MAY require additional parameters to distinguish the
entity being authenticated against. Such parameters are encoded in
the service-specifics portion of the name. The EAP server MUST
reject authentication of any acceptor name that has a non-empty
service-specifics component unless the EAP server understands the
service-specifics and authenticates them. The interpretation of the
service-specifics is scoped by the user-or-service portion. The
realm is similar to the realm portion of a NAI for initiator names;
again the NAI specification's internationalization rules MUST NOT be
applied to the realm. The realm is the administrative realm of a
service for an acceptor name.

The string representation of this name form is designed to be
generally compatible with the string representation of Kerberos names
defined in [RFC1964].

The GSS_C_NT_USER_NAME form represents the name of an individual
user. From the standpoint of this mechanism, it may take the form of
either an undecorated user name or a name semantically similar to a
network access identifier (NAI) [RFC4282]. The name is split at the
first at-sign ('@') into the part preceding the realm, which is the
user-or-service portion of the mechanism name, and the realm portion,
which is the realm portion of the mechanism name.

The GSS_C_NT_HOSTBASED_SERVICE name form represents a service running
on a host; it is textually represented as "service@host". This name
form is required by most SASL profiles and is used by many existing
applications that use the Kerberos GSS-API mechanism. While support
for this name form is critical, it presents an interesting challenge
in terms of EAP channel binding. Consider a case where the server
communicates with a "server proxy," or a AAA server near the server.
That server proxy communicates with the EAP server. The EAP server
and server proxy are in different administrative realms. The server
proxy is in a position to verify that the request comes from the
indicated host. However, the EAP server cannot make this
determination directly. So, the EAP server needs to determine
whether to trust the server proxy to verify the host portion of the
acceptor name. This trust decision depends both on the host name and
the realm of the server proxy. In effect, the EAP server decides
whether to trust that the realm of the server proxy is the right
realm for the given hostname and then makes a trust decision about
the server proxy itself. The same problem appears in Kerberos:
there, clients decide what Kerberos realm to trust for a given
hostname. The service portion of this name is imported into the
user-or-service portion of the mechanism name; the host portion is
imported into the host portion of the mechanism name. The realm
portion is empty. However, authentication will typically fail unless
some AAA component indicates the realm to the EAP server. If the
application server knows its realm, then it should be indicated in
the outgoing AAA request. Otherwise, a proxy SHOULD add the realm.
An alternate form of this name type MAY be used on acceptors; in this
case, the name form is "service" with no host component. This is
imported with the service as user-or-service and an empty host and
realm portion. This form is useful when a service is unsure which
name an initiator knows it by.

If the null name type or the GSS_EAP_NT_EAP_NAME (OID
1.3.6.1.5.5.15.2.1) (see Section 7.1) is imported, then the string
representation above should be directly imported. Mechanisms MAY
support the GSS_KRB5_NT_KRB5_PRINCIPAL_NAME name form with the OID
{iso(1) member-body(2) United States(840) mit(113554) infosys(1)
gssapi(2) krb5(2) krb5_name(1)}. In many circumstances, Kerberos
GSS-API mechanism names will behave as expected when used with the
GSS-API EAP mechanism, but there are some differences that may cause

some confusion. If an implementation does support importing Kerberos
names it SHOULD fail the import if the Kerberos name is not
syntactically a valid GSS-API EAP mechanism name as defined in this
section.

3.2. Internationalization of Names

For the most part, GSS-EAP names are transported in other protocols;
those protocols define the internationalization semantics. For
example, if a AAA server wishes to communicate the user-or-service
portion of the initiator name to an acceptor, it does so using
existing mechanisms in the AAA protocol. Existing
internationalization rules are applied. Similarly, within an
application, existing specifications such as [RFC5178] define the
encoding of names that are imported and displayed with the GSS-API.

This mechanism does introduce a few cases where name components are
sent. In these cases, the encoding of the string is UTF-8. Senders
SHOULD NOT normalize or map strings before sending. These strings
include RADIUS attributes introduced in Section 3.4.

When comparing the host portion of a GSS-EAP acceptor name supplied
in EAP channel binding by a peer to that supplied by an acceptor, EAP
servers SHOULD prepare the host portion according to [RFC5891] prior
to comparison. Applications MAY prepare domain names prior to
importing them into this mechanism.

3.3. Exported Mechanism Names

GSS-API provides the GSS_Export_name call. This call can be used to
export the binary representation of a name. This name form can be
stored on access control lists for binary comparison.

The exported name token MUST use the format described in Section 3.2
of RFC 2743. The mechanism specific portion of this name token is
the string format of the mechanism name described in Section 3.1.

RFC 2744 [RFC2744] places the requirement that the result of
importing a name, canonicalizing it to a Mechanism Name and then
exporting it needs to be the same as importing that name, obtaining
credentials for that principal, initiating a context with those
credentials and exporting the name on the acceptor. In practice, GSS
mechanisms often, but not always, meet this requirement. For names
expected to be used as initiator names, this requirement is met.
However, permitting empty host and realm components when importing
host-based services may make it possible for an imported name to

differ from the exported name actually used. Other mechanisms such
as Kerberos have similar situations where imported and exported names
may differ.

3.4. Acceptor Name RADIUS AVP

See Section 7.4 for registrations of RADIUS attribute types to carry
the acceptor service name. All the attribute types registered in
that section are strings. See Section 3.1 for details of the values
in a name.

If RADIUS is used as a AAA transport, the acceptor MUST send the
acceptor name in these attribute types. That is, the acceptor
decomposes its name and sends any non-empty portion as a RADIUS
attribute. With the exception of the service-specifics portion of
the name, the backslash escaping mechanism is not used in RADIUS
attributes; backslash has no special meaning. In the service-
specifics portion, a literal "/" separates components. In this one
attribute, "\/" indicates a slash character that does not separate
components and "\\" indicates a literal backslash character.

The initiator MUST require that the EAP method in use support channel
binding and MUST send the acceptor name as part of the channel
binding data. The client MUST NOT indicate mutual authentication in
the result of GSS_Init_sec_context unless all name elements that the
client supplied are in a successful channel binding response. For
example, if the client supplied a hostname in channel binding data,
the hostname MUST be in a successful channel binding response.

If an empty target name is supplied to GSS_Init_sec_context, the
initiator MUST fail context establishment unless the acceptor
supplies the acceptor name response (Section 5.4.3). If a null
target name is supplied, the initiator MUST use this response to
populate EAP channel bindings.

3.5. Proxy Verification of Acceptor Name

Proxies may play a role in verification of the acceptor identity.
For example, a AAA proxy near the acceptor may be in a position to
verify the acceptor hostname, while the EAP server is likely to be
too distant to reliably verify this on its own.

The EAP server or some proxy trusted by the EAP server is likely to
be in a position to verify the acceptor realm. In effect, this proxy
is confirming that the right AAA credential is used for the claimed
realm and thus that the acceptor is in the organization it claims to

be part of. This proxy is also typically trusted by the EAP server
to make sure that the hostname claimed by the acceptor is a
reasonable hostname for the realm of the acceptor.

A proxy close to the EAP server is unlikely to be in a position to
confirm that the acceptor is claiming the correct hostname. Instead,
this is typically delegated to a proxy near the acceptor. That proxy
is typically expected to verify the acceptor hostname and to verify
the appropriate AAA credential for that host is used. Such a proxy
may insert the acceptor realm if it is absent, permitting realm
configuration to be at the proxy boundary rather than on acceptors.

Ultimately, specific proxy behavior is a matter for deployment. The
EAP server MUST assure that the appropriate validation has been done
before including acceptor name attributes in a successful channel
binding response. If the acceptor service is included, the EAP
server asserts that the service is plausible for the acceptor. If
the acceptor hostname is included, the EAP server asserts that the
acceptor hostname is verified. If the realm is included the EAP
server asserts that the realm has been verified, and if the hostname
was also included, that the realm and hostname are consistent. Part
of this verification MAY be delegated to proxies, but the EAP server
configuration MUST guarantee that the combination of proxies meets
these requirements. Typically, such delegation will involve business
or operational measures such as cross-organizational agreements as
well as technical measures.

It is likely that future technical work will be needed to communicate
what verification has been done by proxies along the path. Such
technical measures will not release the EAP server from its
responsibility to decide whether proxies on the path should be
trusted to perform checks delegated to them. However, technical
measures could prevent misconfigurations and help to support diverse
environments.

4. Selection of EAP Method

EAP does not provide a facility for an EAP server to advertise what
methods are available to a peer. Instead, a server starts with its
preferred method selection. If the peer does not accept that method,
the peer sends a NAK response containing the list of methods
supported by the client.

Providing multiple facilities to negotiate which security mechanism
to use is undesirable. Section 7.3 of [RFC4462]describes the problem
referencing the Secure Shell (SSH) Protocol key exchange negotiation
and the SPNEGO GSS-API mechanism. If a client preferred an EAP
method A, a non-EAP authentication mechanism B, and then an EAP

method C, then the client would have to commit to using EAP before
learning whether A is actually supported. Such a client might end up
using C when B is available.

The standard solution to this problem is to perform all the
negotiation at one layer. In this case, rather than defining a
single GSS-API mechanism, a family of mechanisms should be defined.
Each mechanism corresponds to an EAP method. The EAP method type
should be part of the GSS-API OID. Then, a GSS-API rather than EAP
facility can be used for negotiation.

Unfortunately, using a family of mechanisms has a number of problems.
First, GSS-API assumes that both the initiator and acceptor know the
entire set of mechanisms that are available. Some negotiation
mechanisms are driven by the client; others are driven by the server.
With EAP GSS-API, the acceptor does not know what methods the EAP
server implements. The EAP server that is used depends on the
identity of the client. The best solution so far is to accept the
disadvantages of multi-layer negotiation and commit to using EAP GSS-
API before a specific EAP method. This has two main disadvantages.
First, authentication may fail when other methods might allow
authentication to succeed. Second, a non-optimal security mechanism
may be chosen.

5. Context Tokens

All context establishment tokens emitted by the EAP mechanism SHALL
have the framing described in Section 3.1 of [RFC2743], as
illustrated by the following pseudo-ASN.1 structures:

```
GSS-API DEFINITIONS ::=
        BEGIN

        MechType ::= OBJECT IDENTIFIER
        -- representing EAP mechanism
        GSSAPI-Token ::=
        -- option indication (delegation, etc.) indicated within
        -- mechanism-specific token
        [APPLICATION 0] IMPLICIT SEQUENCE {
                thisMech MechType,
                innerToken ANY DEFINED BY thisMech
                    -- contents mechanism-specific
                    -- ASN.1 structure not required
                }
        END
```

The innerToken field starts with a 16-bit network byte order token
type identifier. The remainder of the innerToken field is a set of
type-length-value subtokens. The following figure describes the
structure of the inner token:

```
+----------------+--------------------------+
| Octet Position | Description              |
+----------------+--------------------------+
| 0..1           | token ID                 |
|                |                          |
| 2..5           | first subtoken type      |
|                |                          |
| 6..9           | length  of first subtoken|
|                |                          |
| 10..10+n-1     | first subtoken body      |
|                |                          |
| 10+n..10+n+3   | second subtoken type     |
+----------------+--------------------------+
```

                    Structure of Inner Token

   The inner token continues with length, second subtoken body, and so
   forth. If a subtoken type is present, its length and body MUST be
   present.

   The length is a four-octet length of the subtoken body in network
   byte order. The length does not include the length of the type field
   or the length field; the length only covers the body.

   Tokens from the initiator to acceptor use an inner token type with ID
   06 01; tokens from acceptor to initiator use an inner token type with
   ID 06 02. These token types are registered in the registry of RFC
   4121 token types; see Section 7.2.

   See Section 5.7 for the encoding of a complete token. The following
   sections discuss how mechanism OIDs are chosen and the state machine
   that defines what subtokens are permitted at each point in the
   context establishment process.

5.1. Mechanisms and Encryption Types

   This mechanism family uses the security services of the Kerberos
   cryptographic framework [RFC3961]. The root of the OID ARC for
   mechanisms described in this document is 1.3.6.1.5.5.15.1.1; a
   Kerberos encryption type number [RFC3961] is appended to that root
   OID to form a mechanism OID. As such, a particular encryption type
   needs to be chosen. By convention, there is a single object
   identifier arc for the EAP family of GSS-API mechanisms. A specific

mechanism is chosen by adding the numeric Kerberos encryption type
number to the root of this arc. However, in order to register the
SASL name, the specific usage with a given encryption type needs to
be registered. This document defines the EAP-AES128 GSS-API
mechanism.

5.2. Processing Received Tokens

Whenever a context token is received, the receiver performs the
following checks. First, the receiver confirms the object identifier
is that of the mechanism being used. The receiver confirms that the
token type corresponds to the role of the peer: acceptors will only
process initiator tokens and initiators will only process acceptor
tokens.

Implementations of this mechanism maintain a state machine for the
context establishment process. Both the initiator and acceptor start
out in the initial state; see Section 5.4 for a description of this
state. Associated with each state are a set of subtoken types that
are processed in that state and rules for processing these subtoken
types. The receiver examines the subtokens in order, processing any
that are appropriate for the current state. Unknown subtokens or
subtokens that are not expected in the current state are ignored if
their critical bit (see below) is clear.

A state may have a set of required subtoken types. If a subtoken
type is required by the current state but no subtoken of that type is
present, then the context establishment MUST fail.

The most significant bit (0x80000000) in a subtoken type is the
critical bit. If a subtoken with this bit set in the type is
received, the receiver MUST fail context establishment unless the
subtoken is understood and processed for the current state.

The subtoken type MUST be unique within a given token.

5.3. Error Subtokens

   The acceptor may always end the exchange by generating an error
   subtoken. The error subtoken has the following format:

```
+--------+-----------------------------------------------------------+
| Pos    | Description                                               |
+--------+-----------------------------------------------------------+
| 0..3   | 0x80 00 00 01                                             |
|        |                                                           |
| 4..7   | length of error token                                     |
|        |                                                           |
| 8..11  | major status from RFC 2744 as 32-bit network byte order   |
|        |                                                           |
| 12..15 | GSS-EAP error code as 32-bit network byte order; see       |
|        | Section 7.6                                               |
+--------+-----------------------------------------------------------+
```

   Initiators MUST ignore octets beyond the GSS-EAP error code for
   future extensibility. As indicated, the error token is always marked
   critical.

5.4. Initial State

   Both the acceptor and initiator start the context establishment
   process in the initial state.

   The initiator sends a token to the acceptor. It MAY be empty; no
   subtokens are required in this state. Alternatively, the initiator
   MAY include a vendor ID subtoken or an acceptor name request
   subtoken.

   The acceptor responds to this message. It MAY include an acceptor
   name response subtoken. It MUST include a first EAP request; this is
   an EAP request/identity message (see Section 5.5.1 for the format of
   this subtoken).

   The initiator and acceptor then transition to authenticate state.

5.4.1. Vendor Subtoken

   The vendor ID subtoken has type 0x0000000B and the following
   structure:

```
             +-------------+------------------------+
             | Pos         | Description            |
             +-------------+------------------------+
             | 0..3        | 0x0000000B             |
             |             |                        |
             | 4..7        | length of vendor token |
             |             |                        |
             | 8..8+length | Vendor ID string       |
             +-------------+------------------------+
```

   The vendor ID string is an UTF-8 string describing the vendor of this
   implementation. This string is unstructured and for debugging
   purposes only.

5.4.2. Acceptor Name Request

   The acceptor name request token is sent from the initiator to the
   acceptor indicating that the initiator wishes a particular acceptor
   name. This is similar to Transport Layer Security (TLS) Server Name
   Indication [RFC6066] that permits a client to indicate which one of a
   number of virtual services to contact. The structure is as follows:

```
             +------+-----------------------------+
             | Pos  | Description                 |
             +------+-----------------------------+
             | 0..3 | 0x00000002                  |
             |      |                             |
             | 4..7 | length of subtoken          |
             |      |                             |
             | 8..n | string form of acceptor name |
             +------+-----------------------------+
```

   It is likely that channel binding and thus authentication will fail
   if the acceptor does not choose a name that is a superset of this
   name. That is, if a hostname is sent, the acceptor needs to be
   willing to accept this hostname.

5.4.3. Acceptor Name Response

   The acceptor name response subtoken indicates what acceptor name is
   used. This is useful, for example, if the initiator supplied no
   target name to the context initialization. This allows the initiator
   to learn the acceptor name. EAP channel bindings will provide
   confirmation that the acceptor is accurately naming itself.

   This token is sent from the acceptor to initiator. In the Initial
   state, this token would typically be sent if the acceptor name
   request is absent, because if the initiator already sent an acceptor
   name, then the initiator knows what acceptor it wishes to contact.
   This subtoken is also sent in Extensions state Section 5.6, so the
   initiator can protect against a man-in-the-middle modifying the
   acceptor name request subtoken.

                  +------+-----------------------------+
                  | Pos | Description |
                  +------+-----------------------------+
                  | 0..3 | 0x00000003 |
                  | | |
                  | 4..7 | length of subtoken |
                  | | |
                  | 8..n | string form of acceptor name |
                  +------+-----------------------------+

5.5. Authenticate State

   In this state, the acceptor sends EAP requests to the initiator and
   the initiator generates EAP responses. The goal of the state is to
   perform a successful EAP authentication. Since the acceptor sends an
   identity request at the end of the initial state, the first half-
   round-trip in this state is a response to that request from the
   initiator.

   The EAP conversation can end in a number of ways:

   o If the EAP state machine generates an EAP Success message, then
      the EAP authenticator believes the authentication is successful.
      The acceptor MUST confirm that a key has been derived
      (Section 7.10 of [RFC3748]). The acceptor MUST confirm that this
      success indication is consistent with any protected result
      indication for combined authenticators and with AAA indication of
      success for pass-through authenticators. If any of these checks
      fail, the acceptor MUST send an error subtoken and fail the
      context establishment. If these checks succeed, the acceptor
      sends the Success message using the EAP Request subtoken type and
      transitions to Extensions state. If the initiator receives an EAP

Success message, it confirms that a key has been derived and that
the EAP Success is consistent with any protected result
indication. If so, it transitions to Extensions state.
Otherwise, it returns an error to the caller of
GSS_Init_sec_context without producing an output token.

o If the acceptor receives an EAP failure, then the acceptor sends
   this in the EAP Request subtoken type. If the initiator receives
   an EAP Failure, it returns GSS failure.

o If there is some other error, the acceptor MAY return an error
   subtoken.

5.5.1. EAP Request Subtoken

The EAP Request subtoken is sent from the acceptor to the initiator.
This subtoken is always critical and is REQUIRED in the
authentication state.

```
        +-------------+----------------------+
        | Pos         | Description          |
        +-------------+----------------------+
        | 0..3        | 0x80000005           |
        |             |                      |
        | 4..7        | length of EAP message|
        |             |                      |
        | 8..8+length | EAP message          |
        +-------------+----------------------+
```

5.5.2. EAP Response Subtoken

This subtoken is REQUIRED in authentication state messages from the
initiator to the acceptor. It is always critical.

```
        +-------------+----------------------+
        | Pos         | Description          |
        +-------------+----------------------+
        | 0..3        | 0x80000004           |
        |             |                      |
        | 4..7        | length of EAP message|
        |             |                      |
        | 8..8+length | EAP message          |
        +-------------+----------------------+
```

5.6. Extensions State

   After EAP Success, the initiator sends a token to the acceptor
   including additional subtokens that negotiate optional features or
   provide GSS-API channel binding (see Section 6.1). The acceptor then
   responds with a token to the initiator. When the acceptor produces
   its final token, it returns GSS_S_COMPLETE; when the initiator
   consumes this token, it returns GSS_S_COMPLETE if no errors are
   detected.

   The acceptor SHOULD send an acceptor name response (Section 5.4.3) so
   that the initiator can get a copy of the acceptor name protected by
   the Message Integrity Check (MIC) subtoken.

   Both the initiator and acceptor MUST include and verify a MIC
   subtoken to protect the extensions exchange.

5.6.1. Flags Subtoken

   This subtoken is sent to convey initiator flags to the acceptor. The
   flags are sent as a 32-bit integer in network byte order. The only
   flag defined so far is GSS_C_MUTUAL_FLAG, indicating that the
   initiator successfully performed mutual authentication of the
   acceptor. This flag is communicated to the acceptor because some
   protocols [RFC4462] require the acceptor to know whether the
   initiator has confirmed its identity. This flag has the value 0x2 to
   be consistent with RFC 2744.

```
            +-------+-----------------------+
            | Pos   | Description           |
            +-------+-----------------------+
            | 0..3  | 0x0000000C            |
            |       |                       |
            | 4..7  | length of flags token |
            |       |                       |
            | 8..11 | flags                 |
            +-------+-----------------------+
```

   Initiators MUST send 4 octets of flags. Acceptors MUST ignore flag
   octets beyond the first 4 and MUST ignore flag bits other than
   GSS_C_MUTUAL_FLAG. Initiators MUST send undefined flag bits as zero.

5.6.2. GSS Channel Bindings Subtoken

   This subtoken is always critical when sent. It is sent from the
   initiator to the acceptor. The contents of this token are an RFC
   3961 get_mic token of the application data from the GSS channel
   bindings structure passed into the context establishment call.

```
+-------------+-------------------------------------------------+
| Pos         | Description                                     |
+-------------+-------------------------------------------------+
| 0..3        | 0x80000006                                      |
|             |                                                 |
| 4..7        | length of token                                 |
|             |                                                 |
| 8..8+length | get_mic  of  channel binding application data   |
+-------------+-------------------------------------------------+
```

Again, only the application data is sent in the channel binding. Any
initiator and acceptor addresses passed by an application into
context establishment calls are ignored and not sent over the wire.
The checksum type of the get_mic token SHOULD be the mandatory-to-
implement checksum type of the Context Root Key (CRK). The key to
use is the CRK and the key usage is 60 (KEY_USAGE_GSSEAP_CHBIND_MIC).
An acceptor MAY accept any MIC in the channel bindings subtoken if
the channel bindings input to GSS_Accept_sec_context is not provided.
If the channel binding input to GSS_Accept_sec_context is provided,
the acceptor MUST return failure if the channel binding MIC in a
received channel binding subtoken fails to verify.

The initiator MUST send this token if channel bindings including
application data are passed into GSS_Init_sec_context and MUST NOT
send this token otherwise.

5.6.3. MIC Subtoken

This subtoken MUST be the last subtoken in the tokens sent in
Extensions state. This subtoken is sent both by the initiator and
acceptor.

```
+-------------+------------------------------------------------------+
| Pos         | Description                                          |
+-------------+------------------------------------------------------+
| 0..3        | 0x8000000D for initiator 0x8000000E for acceptor     |
|             |                                                      |
| 4..7        | length of RFC 3961 MIC token                         |
|             |                                                      |
| 8..8+length | RFC 3961 result of get_mic                           |
+-------------+------------------------------------------------------+
```

As with any call to get_mic, a token is produced as described in RFC
3961 using the CRK (Section 6) as the key and the mandatory checksum
type for the encryption type of the CRK as the checksum type. The
key usage is 61 (KEY_USAGE_GSSEAP_ACCTOKEN_MIC) for the subtoken from

the acceptor to the initiator and 62 (KEY_USAGE_GSSEAP_INITTOKEN_MIC)
for the subtoken from the initiator to the acceptor. The input is as
follows:

1. The DER-encoded object identifier of the mechanism in use; this
    value starts with 0x06 (the tag for object identifier). When
    encoded in an RFC 2743 context token, the object identifier is
    preceded by the tag and length for [Application 0] SEQUENCE.
    This tag and the length of the overall token is not included;
    only the tag, length, and value of the object identifier itself.

2. A 16-bit token type in network byte order of the RFC 4121 token
    identifier (0x0601 for initiator, 0x0602 for acceptor).

3. For each subtoken, other than the MIC subtoken itself, the order
    the subtokens appear in the token is as follows:

4.

    1. A four-octet subtoken type in network byte order

    2. A four-byte length in network byte order

    3. Length octets of value from that subtoken

5.7. Example Token

```
    +----+------+----+------+-----+------------------------+
    | 60 |  23  | 06 |  09  | 2b  | 06 01 05 05 0f 01 01 11 |
    +----+------+----+------+-----+------------------------+
    |App0|Token |OID |OID   | 1 3 | 6  1  5  5 15  1  1 17 |
    |Tag |length|Tag |length|     |    Mechanism object ID    |
    +----+------+----+------+------------------------------+


    +---------+-------------+-------------+
    |  06 01  | 00 00 00 02 | 00 00 00 0e |
    +---------+-------------|-------------|
    |Initiator | Acceptor   | Length      |
    |context   | name       | (14 octets) |
    |token ID  | request    |             |
    +---------+-------------+-------------+


    +-----------------------------------------+
    | 68 6f 73 74 2f 6c 6f 63 61 6c 68 6f 73 74 |
    +-----------------------------------------+
    | String form of acceptor name            |
    | "host/localhost"                        |
    +-----------------------------------------+
```

Example Initiator Token

5.8. Context Options

   GSS-API provides a number of optional per-context services requested
   by flags on the call to GSS_Init_sec_context and indicated as outputs
   from both GSS_Init_sec_context and GSS_Accept_sec_context. This
   section describes how these services are handled. Which services the
   client selects in the call to GSS_Init_sec_context controls what EAP
   methods MAY be used by the client. Section 7.2 of RFC 3748 describes
   a set of security claims for EAP. As described below, the selected
   GSS options place requirements on security claims that MUST be met.

   This GSS mechanism MUST only be used with EAP methods that provide
   dictionary-attack resistance. Typically, dictionary-attack
   resistance is obtained by using an EAP tunnel method to tunnel an
   inner method in TLS.

   The EAP method MUST support key derivation. Integrity,
   confidentiality, sequencing, and replay detection MUST be indicated
   in the output of GSS_Init_sec_context and GSS_Accept_sec_context
   regardless of which services are requested.

   The PROT_READY service defined in Section 1.2.7 of [RFC2743] is never
   available with this mechanism. Implementations MUST NOT offer this
   flag or permit per-message security services to be used before
   context establishment.

   The EAP method MUST support mutual authentication and channel
   binding. See Section 3.4 for details on what is required for
   successful mutual authentication. Regardless of whether mutual
   authentication is requested, the implementation MUST include channel
   bindings in the EAP authentication. If mutual authentication is
   requested and successful mutual authentication takes place as defined
   in Section 3.4, the initiator MUST send a flags subtoken
   Section 5.6.1 in Extensions state.

6. Acceptor Services

   The context establishment process may be passed through to an EAP
   server via a backend authentication protocol. However, after the EAP
   authentication succeeds, security services are provided directly by
   the acceptor.

   This mechanism uses an RFC 3961 cryptographic key called the Context
   Root Key (CRK). The CRK is derived from the GMSK (GSS-API Master
   Session Key). The GMSK is the result of the random-to-key [RFC3961]
   operation of the encryption type of this mechanism consuming the

appropriate number of bits from the EAP MSK. For example, for
aes128-cts-hmac-sha1-96, the random-to-key operation consumes 16
octets of key material; thus, the first 16 bytes of the MSK are input
to random-to-key to form the GMSK. If the MSK is too short,
authentication MUST fail.

In the following, pseudorandom is the RFC 3961 pseudorandom operation
for the encryption type of the GMSK and random-to-key is the RFC 3961
random-to-key operation for the enctype of the mechanism. The
truncate function takes the initial l bits of its input. The goal in
constructing a CRK is to call the pseudorandom function enough times
to produce the right number of bits of output and discard any excess
bits of output.

The CRK is derived from the GMSK using the following procedure:

Tn = pseudorandom(GMSK, n || "rfc4121-gss-eap")
CRK = random-to-key(truncate(L, T0 || T1 || .. || Tn))
L = random-to-key input size

Where n is a 32-bit integer in network byte order starting at 0 and
incremented to each call to the pseudo_random operation.

6.1. GSS-API Channel Binding

GSS-API channel binding [RFC5554] is a protected facility for
exchanging a cryptographic name for an enclosing channel between the
initiator and acceptor. The initiator sends channel binding data and
the acceptor confirms that channel binding data has been checked.

The acceptor SHOULD accept any channel binding provided by the
initiator if null channel bindings are passed into
gss_accept_sec_context. Protocols such as HTTP Negotiate [RFC4559]
depend on this behavior of some Kerberos implementations.

As discussed, the GSS channel bindings subtoken is sent in the
Extensions state.

6.2. Per-Message Security

The per-message tokens of Section 4 of RFC 4121 are used. The CRK
SHALL be treated as the initiator sub-session key, the acceptor sub-
session key and the ticket session key.

6.3. Pseudorandom Function

The pseudorandom function defined in [RFC4402] is used to provide
GSS_Pseudo_Random functionality to applications.

7. IANA Considerations

   This specification creates a number of IANA registries.

7.1. OID Registry

   IANA has created a registry of ABFAB object identifiers titled
   "Object Identifiers for Application Bridging for Federated Access".
   The initial contents of the registry are specified below. The
   registration policy is IETF Review or IESG Approval [RFC5226]. Early
   allocation is permitted. IANA has updated the reference for the root
   of this OID delegation to point to the newly created registry.

Decimal	Name	Description	References
0	Reserved	Reserved	RFC 7055
1	mechanisms	A sub-arc containing ABFAB mechanisms	RFC 7055
2	nametypes	A sub-arc containing ABFAB GSS-API Name Types	RFC 7055

   Prefix:
   iso.org.dod.internet.security.mechanisms.abfab
        (1.3.6.1.5.5.15)

   NOTE: the following mechanisms registry is the root of the OID for
   the mechanism in question. As discussed in Section 5.1, a Kerberos
   encryption type number [RFC3961] is appended to the mechanism version
   OID below to form the OID of a specific mechanism.

   Prefix:
   iso.org.dod.internet.security.mechanisms.abfab.mechanisms
        (1.3.6.1.5.5.15.1)

Decimal	Name	Description	References
0	Reserved	Reserved	RFC 7055
1	gss-eap-v1	The GSS-EAP mechanism	RFC 7055

   Prefix:
   iso.org.dod.internet.security.mechanisms.abfab.nametypes
        (1.3.6.1.5.5.15.2)

Decimal	Name	Description	References
0	Reserved	Reserved	RFC 7055
1	GSS_EAP_NT_EAP_NAME		RFC 7055, Section 3.1

7.2. RFC 4121 Token Identifiers

   In the top-level registry titled "Kerberos V GSS-API Mechanism
   Parameters", a subregistry called "Kerberos GSS-API Token Type
   Identifiers" was created; the references for this subregistry are RFC
   4121 and this document. The allocation procedure is Expert Review
   [RFC5226]. The Expert's primary job is to make sure that token type
   identifiers are requested by an appropriate requester for the RFC
   4121 mechanism in which they will be used and that multiple values
   are not allocated for the same purpose. For RFC 4121 and this
   mechanism, the Expert is currently expected to make allocations for
   token identifiers from documents in the IETF stream; effectively, for
   these mechanisms, the Expert currently confirms the allocation meets
   the requirements of the IETF Review process.

   The ID field is a hexadecimal token identifier specified in network
   byte order.

   The initial registrations are as follows:

   +-------+----------------------------+--------------------------+
   | ID | Description | Reference |
   +-------+----------------------------+--------------------------+
01 00	KRB_AP_REQ	RFC 4121, Section 4.1
02 00	KRB_AP_REP	RFC 4121, Section 4.1
03 00	KRB_ERROR	RFC 4121, Section 4.1
04 04	MIC tokens	RFC 4121, Section 4.2.6.1
05 04	wrap tokens	RFC 4121, Section 4.2.6.2
06 01	GSS-EAP initiator context	RFC 7055, Section 5
	token	
06 02	GSS EAP acceptor context	RFC 7055, Section 5
	token	
   +-------+----------------------------+--------------------------+

7.3. GSS-EAP Subtoken Types

   This document creates a top-level registry called "The Extensible
   Authentication Protocol Mechanism for the Generic Security Service
   Application Programming Interface (GSS-EAP) Parameters". In any
   short form of that name, including any URI for this registry, it is
   important that the string GSS come before the string EAP; this will

help to distinguish registries if EAP methods for performing GSS-API
authentication are ever defined.

In this registry is a subregistry of subtoken types. Identifiers are
32-bit integers; the upper bit (0x80000000) is reserved as a critical
flag and should not be indicated in the registration. Assignments of
GSS-EAP subtoken types are made by Expert Review [RFC5226]. The
Expert is expected to require a public specification of the subtoken
similar in detail to registrations given in this document. The
security of GSS-EAP depends on making sure that subtoken information
has adequate protection and that the overall mechanism continues to
be secure. Examining the security and architectural consistency of
the proposed registration is the primary responsibility of the
Expert.

```
+------------+-------------------------+---------------+
| Type       | Description             | Reference     |
+------------+-------------------------+---------------+
| 0x00000001 | Error                   | Section 5.3   |
|            |                         |               |
| 0x0000000B | Vendor                  | Section 5.4.1 |
|            |                         |               |
| 0x00000002 | Acceptor name request   | Section 5.4.2 |
|            |                         |               |
| 0x00000003 | Acceptor name response  | Section 5.4.3 |
|            |                         |               |
| 0x00000005 | EAP request             | Section 5.5.1 |
|            |                         |               |
| 0x00000004 | EAP response            | Section 5.5.2 |
|            |                         |               |
| 0x0000000C | Flags                   | Section 5.6.1 |
|            |                         |               |
| 0x00000006 | GSS-API channel bindings| Section 5.6.2 |
|            |                         |               |
| 0x0000000D | Initiator MIC           | Section 5.6.3 |
|            |                         |               |
| 0x0000000E | Acceptor MIC            | Section 5.6.3 |
+------------+-------------------------+---------------+
```

7.4. RADIUS Attribute Assignments

The following RADIUS attribute type values [RFC3575] are assigned.
The allocation instructions in Section 10.3 of [RFC6929] have been
followed.

```
+--------------------------------+-------+--------------------------+
| Description                    | Value | More Information         |
+--------------------------------+-------+--------------------------+
| GSS-Acceptor-Service-Name      | 164   | user-or-service portion  |
|                                |       | of name                  |
|                                |       |                          |
| GSS-Acceptor-Host-Name         | 165   | host portion of name     |
|                                |       |                          |
| GSS-Acceptor-Service-Specifics | 166   | service-specifics        |
|                                |       | portion of name          |
|                                |       |                          |
| GSS-Acceptor-Realm-Name        | 167   | Realm portion of name    |
+--------------------------------+-------+--------------------------+
```

7.5. Registration of the EAP-AES128 SASL Mechanisms

   Subject: Registration of SASL mechanisms EAP-AES128 and
      EAP-AES128-PLUS

   SASL mechanism names: EAP-AES128 and EAP-AES128-PLUS

   Security considerations: See RFC 5801 and RFC 7055

   Published specification (recommended): RFC 7055

   Person & email address to contact for further information:
      Abfab Working Group, abfab@ietf.org

   Intended usage: common

   Owner/Change controller: iesg@ietf.org

   Note: This mechanism describes the GSS-EAP mechanism used with the
      aes128-cts-hmac-sha1-96 enctype. The GSS-API OID for this
      mechanism is 1.3.6.1.5.5.15.1.1.17.

      As described in RFC 5801, a PLUS variant of this mechanism is also
      required.

7.6. GSS-EAP Errors

   A new subregistry is created in the GSS-EAP parameters registry
   titled "GSS-EAP Error Codes". The error codes in this registry are
   unsigned 32-bit numbers. Values less than or equal to 127 are
   assigned by Standards Action [RFC5226]. Values 128 through 255 are
   assigned with the Specification Required assignment policy [RFC5226].

Values greater than 255 are reserved; updates to registration policy
may make these values available for assignment and implementations
MUST be prepared to receive them.

This table provides the initial contents of the registry.

```
+-------+------------------------------------------------+
| Value | Description                                    |
+-------+------------------------------------------------+
| 0     | Reserved                                       |
|       |                                                |
| 1     | Buffer is incorrect size                       |
|       |                                                |
| 2     | Incorrect mechanism OID                        |
|       |                                                |
| 3     | Token is corrupted                             |
|       |                                                |
| 4     | Token is truncated                             |
|       |                                                |
| 5     | Packet received by direction that sent it      |
|       |                                                |
| 6     | Incorrect token type identifier                |
|       |                                                |
| 7     | Unhandled critical subtoken received           |
|       |                                                |
| 8     | Missing required subtoken                      |
|       |                                                |
| 9     | Duplicate subtoken type                        |
|       |                                                |
| 10    | Received unexpected subtoken for current state |
|       |                                                |
| 11    | EAP did not produce a key                      |
|       |                                                |
| 12    | EAP key too short                              |
|       |                                                |
| 13    | Authentication rejected                        |
|       |                                                |
| 14    | AAA returned an unexpected message type        |
|       |                                                |
| 15    | AAA response did not include EAP request       |
|       |                                                |
| 16    | Generic AAA failure                            |
+-------+------------------------------------------------+
```

7.7. GSS-EAP Context Flags

   A new subregistry is created in the GSS-EAP parameters registry.
   This registry holds registrations of flag bits sent in the flags
   subtoken (Section 5.6.1). There are 32 flag bits available for
   registration represented as hexadecimal numbers from the most
   significant bit 0x80000000 to the least significant bit 0x1. The
   registration policy for this registry is IETF Review or, in
   exceptional cases, IESG Approval. The following table indicates
   initial registrations; all other values are available for assignment.

```
              +------+-------------------+---------------+
              | Flag | Name              | Reference     |
              +------+-------------------+---------------+
              | 0x2  | GSS_C_MUTUAL_FLAG | Section 5.6.1 |
              +------+-------------------+---------------+
```

8. Security Considerations

   RFC 3748 discusses security issues surrounding EAP. RFC 5247
   discusses the security and requirements surrounding key management
   that leverages the AAA infrastructure. These documents are critical
   to the security analysis of this mechanism.

   RFC 2743 discusses generic security considerations for the GSS-API.
   RFC 4121 discusses security issues surrounding the specific per-
   message services used in this mechanism.

   As discussed in Section 4, this mechanism may introduce multiple
   layers of security negotiation into application protocols. Multiple
   layer negotiations are vulnerable to a bid-down attack when a
   mechanism negotiated at the outer layer is preferred to some but not
   all mechanisms negotiated at the inner layer; see Section 7.3 of
   [RFC4462] for an example. One possible approach to mitigate this
   attack is to construct security policy such that the preference for
   all mechanisms negotiated in the inner layer falls between
   preferences for two outer-layer mechanisms or falls at one end of the
   overall ranked preferences including both the inner and outer layer.
   Another approach is to only use this mechanism when it has
   specifically been selected for a given service. The second approach
   is likely to be common in practice because one common deployment will
   involve an EAP supplicant interacting with a user to select a given
   identity. Only when an identity is successfully chosen by the user
   will this mechanism be attempted.

   EAP channel binding is used to give the GSS-API initiator confidence
   in the identity of the GSS-API acceptor. Thus, the security of this
   mechanism depends on the use and verification of EAP channel binding.

Today, EAP channel binding is in very limited deployment. If EAP
channel binding is not used, then the system may be vulnerable to
phishing attacks where a user is diverted from one service to
another. If the EAP method in question supports mutual
authentication then users can only be diverted between servers that
are part of the same AAA infrastructure. For deployments where
membership in the AAA infrastructure is limited, this may serve as a
significant limitation on the value of phishing as an attack. For
other deployments, use of EAP channel binding is critical to avoid
phishing. These attacks are possible with EAP today although not
typically with common GSS-API mechanisms. For this reason,
implementations are required to implement and use EAP channel
binding; see Section 3 for details.

The security considerations of EAP channel binding [RFC6677] describe
the security properties of channel binding. Two attacks are worth
calling out here. First, when a tunneled EAP method is used, it is
critical that the channel binding be performed with an EAP server
trusted by the peer. With existing EAP methods, this typically
requires validating the certificate of the server tunnel endpoint
back to a trust anchor and confirming the name of the entity who is a
subject of that certificate. EAP methods may suffer from bid-down
attacks where an attacker can cause a peer to think that a particular
EAP server does not support channel binding. This does not directly
cause a problem because mutual authentication is only offered at the
GSS-API level when channel binding to the server's identity is
successful. However, when an EAP method is not vulnerable to these
bid-down attacks, additional protection is available. This mechanism
will benefit significantly from new strong EAP methods such as
[TEAP].

Every proxy in the AAA chain from the authenticator to the EAP server
needs to be trusted to help verify channel bindings and to protect
the integrity of key material. GSS-API applications may be built to
assume a trust model where the acceptor is directly responsible for
authentication. However, GSS-API is definitely used with trusted-
third-party mechanisms such as Kerberos.

RADIUS does provide a weak form of hop-by-hop confidentiality of key
material based on using MD5 as a stream cipher. Diameter can use TLS
or IPsec but has no mandatory-to-implement confidentiality mechanism.
Operationally, protecting key material as it is transported between
the Identity Provider (IdP) and Relying Party (RP) is critical to
per-message security and verification of GSS-API channel binding
[RFC5056]. Mechanisms such as RADIUS over TLS [RFC6614] provide
significantly better protection of key material than the base RADIUS
specification.

9. Acknowledgements

   Luke Howard, Jim Schaad, Alejandro Perez Mendez, Alexey Melnikov, and
   Sujing Zhou provided valuable reviews of this document.

   Rhys Smith provided the text for the OID registry section. Sam
   Hartman's work on this document has been funded by JANET.

10. References

10.1. Normative References

   [GSS-IANA] IANA, "GSS-API Service Name Registry",
              <http://www.iana.org/assignments/gssapi-service-names>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC2743] Linn, J., "Generic Security Service Application Program
              Interface Version 2, Update 1", RFC 2743, January 2000.

   [RFC2744] Wray, J., "Generic Security Service API Version 2 :
              C-bindings", RFC 2744, January 2000.

   [RFC3575] Aboba, B., "IANA Considerations for RADIUS (Remote
              Authentication Dial In User Service)", RFC 3575, July
              2003.

   [RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H.
              Levkowetz, "Extensible Authentication Protocol (EAP)", RFC
              3748, June 2004.

   [RFC3961] Raeburn, K., "Encryption and Checksum Specifications for
              Kerberos 5", RFC 3961, February 2005.

   [RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos
              Version 5 Generic Security Service Application Program
              Interface (GSS-API) Mechanism: Version 2", RFC 4121, July
              2005.

   [RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The
              Network Access Identifier", RFC 4282, December 2005.

   [RFC4401] Williams, N., "A Pseudo-Random Function (PRF) API
              Extension for the Generic Security Service Application
              Program Interface (GSS-API)", RFC 4401, February 2006.

   [RFC4402] Williams, N., "A Pseudo-Random Function (PRF) for the
              Kerberos V Generic Security Service Application Program
              Interface (GSS-API) Mechanism", RFC 4402, February 2006.

   [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure
              Channels", RFC 5056, November 2007.

   [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
              IANA Considerations Section in RFCs", BCP 26, RFC 5226,
              May 2008.

   [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax
              Specifications: ABNF", STD 68, RFC 5234, January 2008.

   [RFC5554] Williams, N., "Clarifications and Extensions to the
              Generic Security Service Application Program Interface
              (GSS-API) for the Use of Channel Bindings", RFC 5554, May
              2009.

   [RFC5891] Klensin, J., "Internationalized Domain Names in
              Applications (IDNA): Protocol", RFC 5891, August 2010.

   [RFC6677] Hartman, S., Clancy, T., and K. Hoeper, "Channel-Binding
              Support for Extensible Authentication Protocol (EAP)
              Methods", RFC 6677, July 2012.

   [RFC7057] Winter, S. and J. Salowey, "Update to the Extensible
              Authentication Protocol (EAP) Applicability Statement for
              Application Bridging for Federated Access Beyond Web
              (ABFAB)", RFC 7057, December 2013.

10.2. Informative References

   [ABFAB-ARCH]
              Howlett, J., Hartman, S., Tschofenig, H., Lear, E., and J.
              Schaad, "Application Bridging for Federated Access Beyond
              Web (ABFAB) Architecture", Work in Progress, July 2013.

   [RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC
              1964, June 1996.

   [RFC3579] Aboba, B. and P. Calhoun, "RADIUS (Remote Authentication
              Dial In User Service) Support For Extensible
              Authentication Protocol (EAP)", RFC 3579, September 2003.

   [RFC4072] Eronen, P., Hiller, T., and G. Zorn, "Diameter Extensible
              Authentication Protocol (EAP) Application", RFC 4072,
              August 2005.

   [RFC4178] Zhu, L., Leach, P., Jaganathan, K., and W. Ingersoll, "The
              Simple and Protected Generic Security Service Application
              Program Interface (GSS-API) Negotiation Mechanism", RFC
              4178, October 2005.

   [RFC4422] Melnikov, A. and K. Zeilenga, "Simple Authentication and
              Security Layer (SASL)", RFC 4422, June 2006.

   [RFC4462] Hutzelman, J., Salowey, J., Galbraith, J., and V. Welch,
              "Generic Security Service Application Program Interface
              (GSS-API) Authentication and Key Exchange for the Secure
              Shell (SSH) Protocol", RFC 4462, May 2006.

   [RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based
              Kerberos and NTLM HTTP Authentication in Microsoft
              Windows", RFC 4559, June 2006.

   [RFC5178] Williams, N. and A. Melnikov, "Generic Security Service
              Application Program Interface (GSS-API)
              Internationalization and Domain-Based Service Names and
              Name Type", RFC 5178, May 2008.

   [RFC5247] Aboba, B., Simon, D., and P. Eronen, "Extensible
              Authentication Protocol (EAP) Key Management Framework",
              RFC 5247, August 2008.

   [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions:
              Extension Definitions", RFC 6066, January 2011.

   [RFC6614] Winter, S., McCauley, M., Venaas, S., and K. Wierenga,
              "Transport Layer Security (TLS) Encryption for RADIUS",
              RFC 6614, May 2012.

   [RFC6929] DeKok, A. and A. Lior, "Remote Authentication Dial In User
              Service (RADIUS) Protocol Extensions", RFC 6929, April
              2013.

   [TEAP]   Zhou, H., Cam-Winget, N., Salowey, J., and S. Hanna,
              "Tunnel EAP Method (TEAP) Version 1", Work in Progress,
              September 2013.

Appendix A. Pre-publication RADIUS VSA

   As described in Section 3.4, RADIUS attributes are used to carry the
   acceptor name when this family of mechanisms is used with RADIUS.
   Prior to the publication of this specification, a vendor-specific
   RADIUS attribute was used. This non-normative appendix documents
   that attribute as it may be seen from older implementations.

   Prior to IANA assignment, GSS-EAP used a RADIUS vendor-specific
   attribute for carrying the acceptor name. The Vendor-Specific
   Attribute (VSA) with enterprise ID 25622 is formatted as a VSA
   according to the recommendation in the RADIUS specification. The
   following sub-attributes are defined:

   +------------------------------+-----------+---------------------+
   | Name | Attribute | Description |
   +------------------------------+-----------+---------------------+
GSS-Acceptor-Service-Name	128	user-or-service
		portion of name
GSS-Acceptor-Host-Name	129	host portion of name
GSS-Acceptor-Service-Specifics	130	service-specifics
		portion of name
GSS-Acceptor-Realm-Name	131	Realm portion of
		name
   +------------------------------+-----------+---------------------+

Authors' Addresses

   Sam Hartman (editor)
   Painless Security

   EMail: hartmans-ietf@mit.edu

   Josh Howlett
   JANET(UK)

   EMail: josh.howlett@ja.net