TRAM
Internet-Draft
Intended status: Standards Track
Expires: April 5, 2015

T. Reddy
P. Patil
R. Ravindranath
Cisco
J. Uberti
Google

October 2, 2014

Session Traversal Utilities for NAT (STUN) Extension for Third Party Authorization
draft-ietf-tram-turn-third-party-authz-05

Abstract

This document proposes the use of OAuth to obtain and validate ephemeral tokens that can be used for Session Traversal Utilities for NAT (STUN) authentication. The usage of ephemeral tokens ensure that access to a STUN server can be controlled even if the tokens are compromised.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Session Traversal Utilities for NAT (STUN) [RFC5389] provides a mechanism to control access via "long-term" username/ password credentials that are provided as part of the STUN protocol. It is expected that these credentials will be kept secret; if the credentials are discovered, the STUN server could be used by unauthorized users or applications. However, in web applications, ensuring this secrecy is typically impossible.

To address this problem and the ones described in [I-D.ietf-tram-auth-problems], this document proposes the use of third party authorization using OAuth for STUN. Using OAuth, a client obtains an ephemeral token from an authorization server e.g. WebRTC server, and the token is presented to the STUN server instead of the traditional mechanism of presenting username/password

credentials. The STUN server validates the authenticity of the token and provides required services.

2. Terminology

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].

- WebRTC Server: A web server that supports WebRTC [I-D.ietf-rtcweb-overview].

- Access Token: OAuth 2.0 access token.

- mac_key: The session key generated by the authorization server. This session key has a lifetime that corresponds to the lifetime of the access token, is generated by the authorization server and bound to the access token.

- kid: An ephemeral and unique key identifier. The kid also allows the resource server to select the appropriate keying material for decryption.

3. Solution Overview

This specification uses the token type 'Assertion' (aka self-contained token) described in [RFC6819] where all the information necessary to authenticate the validity of the token is contained within the token itself. This approach has the benefit of avoiding a protocol between the STUN server and the authorization server for token validation, thus reducing latency. The exact mechanism used by a client to obtain a token from the OAuth authorization server is outside the scope of this document. For example, a client could make an HTTP request to an authorization server to obtain a token that can be used to avail STUN services. The STUN token is returned in JSON, along with other OAuth Parameters like token type, mac_key, kid, token lifetime etc. The client is oblivious to the content of the token. The token is embedded within a STUN request sent to the STUN server. Once the STUN server has determined the token is valid, it's services are offered for a determined period of time.

```
   +-------------------+                         +--------+  +---------+
   | .........  STUN   |                         |  STUN  |  |  WebRTC |
   | .WebRTC .  Client |                         |        |  |         |
   | .Client .         |                         | Server |  |  Server |
   | .........         |                         |        |  |         |
   +-------------------+                         +--------+  +---------+
     |       |           STUN request                    |         |
     |       |------------------------------------------>|         |
     |       |                                           |         |
     |       |         STUN error response               |         |
     |       |         (401 Unauthorized)                |         |
     |       |<------------------------------------------|         |
     |       |         THIRD-PARTY-AUTHORIZATION         |         |
     |       |                                           |         |
     |       |                                           |         |
     |       |      HTTP Request for token               |         |
     |------------------------------------------------------------>|
     |       |      HTTP Response with token parameters  |         |
     |<------------------------------------------------------------|
     |OAuth  |                                           |         |
      Attributes                                         |         |
     |------>|                                           |         |
     |       |    STUN request with ACCESS-TOKEN         |         |
     |       |------------------------------------------>|         |
     |       |                                           |         |
     |       |         STUN success response             |         |
     |       |<------------------------------------------|         |
     |       |             STUN Messages                 |         |
     |       |      ////// integrity protected //////    |         |
     |       |      ////// integrity protected //////    |         |
     |       |      ////// integrity protected //////    |         |
```

Figure 1: STUN Third Party Authorization

Note : An implementation may choose to contact the WebRTC server to obtain a token even before it makes a STUN request, if it knows the server details before hand. For example, once a client has learnt that a STUN server supports Third Party authorization from a WebRTC server, the client can obtain the token before making subsequent STUN requests.

[I-D.ietf-oauth-pop-key-distribution] describes the interaction between the client and the authorization server. For example, the client learns the STUN server name "stun1@example.com" from THIRD-PARTY-AUTHORIZATION attribute value and makes the following HTTP request for the access token using transport-layer security (with extra line breaks for display purposes only):

```
     POST /o/oauth2/token HTTP/1.1
     Host: server.example.com
     Content-Type: application/x-www-form-urlencoded
     aud=stun1@example.com
     timestamp=1361471629
     grant_type=implicit
     token_type=pop
     alg=HMAC-SHA-1 HMAC-SHA-256-128
```

Figure 2: Request

In the future STUNbis [I-D.salgueiro-tram-stunbis] will support hash agility and accomplish this agility by conveying the HMAC algorithms supported by the STUN server along with a STUN error message to the client. The client then signals the intersection-set of algorithms supported by it and the STUN server to the authorization server in the 'alg' parameter defined in [I-D.ietf-oauth-pop-key-distribution]. Authorization server selects an HMAC algorithm from the list of algorithms client had provided and determines length of the mac_key based on the selected HMAC algorithm. Note that until STUN supports hash agility HMAC-SHA1 is the only valid hash algorithm that client can signal to the authorization server and vice-versa.

If the client is authorized then the authorization server issues an access token. An example of successful response:

```
     HTTP/1.1 200 OK
     Content-Type: application/json
     Cache-Control: no-store

     {
       "access_token":
"U2FsdGVkX18qJK/kkWmRcnfHglrVTJSpS6yU32kmHmOrfGyI3m1gQj1jRPsr0uBb
HctuycAgsfRX7nJW2BdukGyKMXSiNGNnBzigkAofP6+Z3vkJ1Q5pWbfSRroOkWBn",
       "token_type":"pop",
       "expires_in":1800,
       "kid":"22BIjxU93h/IgwEb",
       "mac_key":"v51N62OM65kyMvfTI08O"
       "alg":HMAC-SHA-256-128
     }
```

Figure 3: Response

Access token and other attributes issued by the authorization server are explained in Section 6.2. OAuth in [RFC6749] defines four grant types. This specification uses the OAuth grant type "Implicit" explained in section 1.3.2 of [RFC6749] where the WebRTC client is

issued an access token directly. The value of the scope parameter explained in section 3.3 of [RFC6749] MUST be 'stun' string.

4. Obtaining a Token Using OAuth

A STUN client should know the authentication capability of the STUN server before deciding to use third party authorization. A STUN client initially makes a request without any authorization. If the STUN server supports or mandates third party authorization, it will return an error message indicating support for third party authorization. The STUN server includes an ERROR-CODE attribute with a value of 401 (Unauthorized), a nonce value in a NONCE attribute and a SOFTWARE attribute that gives information about the STUN server's software. The STUN servers also includes additional STUN attribute THIRD-PARTY-AUTHORIZATION signaling the STUN client that the STUN server supports third party authorization.

Consider the following example that illustrates the use of OAuth to achieve third party authorization for TURN. In this example, a resource owner i.e. WebRTC server, authorizes a TURN client to access resources on a TURN server.

OAuth	WebRTC
Client	WebRTC client
Resource owner	WebRTC server
Authorization server	Authorization server
Resource server	TURN Server

Figure 4: OAuth terminology mapped to WebRTC terminology

Using the OAuth 2.0 authorization framework, a WebRTC client (third-party application) obtains limited access to a TURN (resource server) on behalf of the WebRTC server (resource owner or authorization server). The WebRTC client requests access to resources controlled by the resource owner (WebRTC server) and hosted by the resource server (TURN server). The WebRTC client obtains access token, lifetime, session key (in the mac_key parameter) and key id (kid). The TURN client conveys the access token and other OAuth parameters learnt from the authorization server to the resource server (TURN server). The TURN server obtains the session key from the access token. The TURN server validates the token, computes the message integrity of the request and takes appropriate action i.e permits the

TURN client to create allocations. This is shown in an abstract way in Figure 5.

```
                           +---------------+
                           |               +<******+
            +------------->| Authorization |       *
            |              | Server        |       *
            |   +----------|(WebRTC Server)|       *  AS-RS,
            |   |          |               |       *  AUTH keys
   (2)      |   |           +---------------+      *   (1)
   Access   |   |  (3)                             *
   Token    |   | Access Token                     *
   Request  |   |    +                             *
            |   | Session Key                      *
            |   |                                  *
            |   V                                  V
        +-------+---+                       +-+----=-----+
        |           |         (4)           |            |
        |           | TURN Request + Access |            |
        | WebRTC    | Token                 | TURN       |
        | Client    |---------------------->| Server     |
        | (Alice)   | Allocate Response (5) |            |
        |           |<----------------------|            |
        +-----------+                       +------------+

   User : Alice
   ****: Out-of-Band Long-Term Key Establishment
```

Figure 5: Interactions

4.1. Key Establishment

The authorization server shares a long-term secret (like asymmetric credentials) with the resource server for mutual authentication. The STUN server and authorization server MUST establish a symmetric key (K), using an out of band mechanism. Symmetric key MUST be chosen to ensure that the size of encrypted token is not large because usage of asymmetric keys will result in large encrypted tokens which may not fit into a single STUN message. The AS-RS, AUTH keys will be derived from K. AS-RS key is used for encrypting the self-contained token and message integrity of the encrypted token is calculated using the AUTH key. The STUN and authorization servers MUST establish the symmetric key over an authenticated secure channel. The establishment of symmetric key is outside the scope of this specification. For example, implementations could use one of the following mechanisms in to establish a symmetric key.

4.1.1. DSKPP

The two servers could choose to use Dynamic Symmetric Key Provisioning Protocol (DSKPP) [RFC6063] to establish a symmetric key (K). The encryption and MAC algorithms will be negotiated using the KeyProvClientHello, KeyProvServerHello messages. A unique key identifier (referred to as KeyID) for the symmetric key is generated by the DSKPP server (i.e. Authorization server) and signalled to the DSKPP client (i.e STUN server) which is equivalent to the kid defined in this specification. The AS-RS, AUTH keys would be derived from the symmetric key using (HMAC)-based key derivation function (HKDF) [RFC5869] and the default hash function is SHA-256. For example if the input symmetric key (K) is 32 octets length, encryption algorithm is AES_256_CBC and HMAC algorithm is HMAC-SHA-256-128 then the secondary keys AS-RS, AUTH are generated from the input key K as follows

1. HKDF-Extract(zero, K) -> PRK

2. HKDF-Expand(PRK, zero, 32) -> AS-RS key

3. HKDF-Expand(PRK, zero, 32) -> AUTH key

If Authenticated Encryption with Associated Data (AEAD) algorithm defined in [RFC5116] is used then there is no need to generate the AUTH key.

4.1.2. HTTP interactions

The two servers could choose to use REST API to establish a symmetric key. To retrieve a new symmetric key, the STUN server makes an HTTP GET request to the authorization server, specifying STUN as the service to allocate the symmetric keys for, and specifying the name of the STUN server. The response is returned with content-type "application/json", and consists of a JSON object containing the symmetric key.

```
Request
-------

service - specifies the desired service (turn)
name    -  STUN server name be associated with the key

example: GET /?service=stun&name=turn1@example.com


Response
--------

key - Long-term key (K)
ttl - the duration for which the key is valid, in seconds.

example:
{
   "key" :
"ESIzRFVmd4iZABEiM0RVZgKn6WjLaTC1FXAghRMVTzkBGNaaN496523WIISKerLi",
   "ttl" : 86400,
   "kid" :"22BIjxU93h/IgwEb"
  }
```

The AS-RS, AUTH keys are derived from K using HKDF as discussed in Section 4.1.1. Authorization server must also signal a unique key identifier (kid) to the STUN server which will be used to select the appropriate keying material for decryption. The default encryption algorithm to encrypt the self-contained token could be Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode (AES_256_CBC). The default HMAC algorithm to calculate the integrity of the token could be HMAC-SHA-256-128. In this case AS-RS key length must be 256-bit, AUTH key length must be 256-bit (section 2.6 of [RFC4868]).

4.1.3. Manual provisioning

STUN and authorization servers could be manually configured with a symmetric key (K) and kid. The default encryption and HMAC algorithms could be AES_256_CBC, HMAC-SHA-256-128.

Note : The mechanisms specified in Section 4.1.2 Section 4.1.3 are easy to implement and deploy compared to DSKPP but lack encryption and HMAC algorithm agility.

5. Forming a Request

When a STUN server responds that third party authorization is required, a STUN client re-attempts the request, this time including access token and kid values in ACCESS-TOKEN and USERNAME STUN attributes. The STUN client includes a MESSAGE-INTEGRITY attribute as the last attribute in the message over the contents of the STUN message. The HMAC for the MESSAGE-INTEGRITY attribute is computed as described in section 15.4 of [RFC5389] where the mac_key is used as the input key for the HMAC computation. The STUN client and server will use the mac_key to compute the message integrity and doesn't have to perform MD5 hash on the credentials.

6. STUN Attributes

The following new STUN attributes are introduced by this specification to accomplish third party authorization.

6.1. THIRD-PARTY-AUTHORIZATION

This attribute is used by the STUN server to inform the client that it supports third party authorization. This attribute value contains the STUN server name. The STUN server may have tie-up with multiple authorization servers and vice versa, so the client MUST provide the STUN server name to the authorization server so that it can select the appropriate keying material to generate the self-contained token. The THIRD-PARTY-AUTHORIZATION attribute is a comprehension-optional attribute (see Section 15 from [RFC5389]).

6.2. ACCESS-TOKEN

The access token is issued by the authorization server. OAuth does not impose any limitation on the length of the access token but if path MTU is unknown then STUN messages over IPv4 would need to be less than 548 bytes (Section 7.1 of [RFC5389]), access token length needs to be restricted to fit within the maximum STUN message size. Note that the self-contained token is opaque to the client and it MUST NOT examine the ticket. The ACCESS-TOKEN attribute is a comprehension-required attribute (see Section 15 from [RFC5389]).

The token is structured as follows:

```
      struct {
          opaque {
              uint16_t key_length;
              opaque mac_key[key_length];
              uint64_t timestamp;
              uint32_t lifetime;
          } encrypted_block;
          opaque mac[mac_length];
      } token;
```

Figure 6: Self-contained token format

Note: uintN_t means an unsigned integer of exactly N bits. Single-byte entities containing uninterpreted data are of type opaque. All values in the token are stored in network byte order.

The fields are described below:

key_length: Length of the session key in octets. Key length of 160-bits MUST be supported (i.e only 160-bit key is used by HMAC-SHA-1 for message integrity of STUN message). The key length facilitates the hash agility plan discussed in section 16.3 of [RFC5389].

mac_key: The session key generated by the authorization server.

timestamp: 64-bit unsigned integer field containing a timestamp. The value indicates the time since January 1, 1970, 00:00 UTC, by using a fixed point format. In this format, the integer number of seconds is contained in the first 48 bits of the field, and the remaining 16 bits indicate the number of 1/64K fractions of a second (Native format - Unix).

lifetime: The lifetime of the access token, in seconds. For example, the value 3600 indicates one hour. The Lifetime value SHOULD be equal to the "expires_in" parameter defined in section 4.2.2 of [RFC6749].

encrypted_block: The encrypted_block is encrypted using the symmetric long-term key established between the resource server and the authorization server. Shown in Figure 5 as AS-RS key.

mac: The Hashed Message Authentication Code (HMAC) is calculated with the AUTH key over the 'encrypted_block' and the STUN server name (N) conveyed in the THIRD-PARTY-AUTHORIZATION response. This ensures that the client does not use the same token to gain illegal access to other STUN servers provided by the same administrative domain i.e., when multiple STUN servers in a single

administrative domain share the same symmetric key with an authorization server. The length of the mac field is known to the STUN and authorization server based on the negotiated MAC algorithm.

An example encryption process is illustrated below. Here C, N denote Ciphertext and STUN server name respectively.

o C = AES_256_CBC(AS-RS, encrypted_block)

o mac = HMAC-SHA-256-128(AUTH, C | | N)

Encryption is applied before message authentication on the sender side and conversely on the receiver side. The entire token i.e., the 'encrypted_block' and 'mac' is base64 encoded (see section 4 of [RFC4648]) and the resulting access token is signaled to the client. If AEAD algorithm is used then there is no need to explicitly compute HMAC, the associated data MUST be the STUN server name (N) and the mac field MUST carry the nonce. The length of nonce MUST be 12 octets.

7. Receiving a request with ACCESS-TOKEN attribute

The STUN server, on receiving a request with ACCESS-TOKEN attribute, performs checks listed in section 10.2.2 of [RFC5389] in addition to the following steps to verify that the access token is valid:

o STUN server selects the keying material based on kid signalled in the USERNAME attribute.

o It performs the verification of the token message integrity by calculating HMAC over the encrypted portion in the self-contained token and STUN server name using AUTH key and if the resulting value does not match the mac field in the self-contained token then it rejects the request with an error response 401 (Unauthorized). If AEAD algorithm is used then it has only a single output, either a plaintext or a special symbol FAIL that indicates that the inputs are not authentic.

o STUN server obtains the mac_key by retrieving the content of the access token (which requires decryption of the self-contained token using the AS-RS key).

o The STUN server verifies that no replay took place by performing the following check:

* The access token is accepted if the timestamp field (TS) in the self-contained token is recent enough to the reception time of

the STUN request (RDnew) using the following formula: Lifetime + Delta > abs(RDnew - TS). The RECOMMENDED value for the allowed Delta is 5 seconds. If the timestamp is NOT within the boundaries then the STUN server discards the request with error response 401 (Unauthorized).

- The STUN server uses the mac_key to compute the message integrity over the request and if the resulting value does not match the contents of the MESSAGE-INTEGRITY attribute then it rejects the request with an error response 401 (Unauthorized).

- If all the checks pass, the STUN server continues to process the request. Any response generated by the server MUST include the MESSAGE-INTEGRITY attribute, computed using the mac_key.

8. Changes to STUN Client

- A STUN response is discarded by the client if the value computed for message integrity using mac_key does not match the contents of the MESSAGE-INTEGRITY attribute.

- If the access token expires then the client MUST obtain a new token from the authorization server and use it for new STUN requests.

9. Usage with TURN

Traversal Using Relay NAT (TURN) [RFC5766] an extension to the STUN protocol is often used to improve the connectivity of P2P applications. By providing a cloud-based relay service, TURN ensures that a connection can be established even when one or both sides is incapable of a direct P2P connection. However, as a relay service, it imposes a nontrivial cost on the service provider. Therefore, access to a TURN service is almost always access-controlled. In order to achieve third party authorization, a resource owner e.g. WebRTC server, authorizes a TURN client to access resources on the TURN server.

```
   +-------------------+                         +--------+  +---------+
   | .........  TURN   |                         |  TURN  |  |  WebRTC |
   | .WebRTC .  Client |                         |        |  |         |
   | .Client .         |                         | Server |  |  Server |
   | .........         |                         |        |  |         |
   +-------------------+                         +--------+  +---------+
     |       |           Allocate request                |         |
     |       |------------------------------------------>|         |
     |       |                                           |         |
     |       |         Allocate error response           |         |
     |       |         (401 Unauthorized)                |         |
     |       |<------------------------------------------|         |
     |       |         THIRD-PARTY-AUTHORIZATION         |         |
     |       |                                           |         |
     |       |                                           |         |
     |       |      HTTP Request for token               |         |
     |------------------------------------------------------------>|
     |       |      HTTP Response with token parameters  |         |
     |<------------------------------------------------------------|
     |OAuth  |                                           |         |
      Attributes                                         |         |
     |------>|                                           |         |
     |       |    Allocate request ACCESS-TOKEN          |         |
     |       |------------------------------------------>|         |
     |       |                                           |         |
     |       |         Allocate success response         |         |
     |       |<------------------------------------------|         |
     |       |             TURN Messages                 |         |
     |       |      ////// integrity protected //////    |         |
     |       |      ////// integrity protected //////    |         |
     |       |      ////// integrity protected //////    |         |
```

Figure 7: TURN Third Party Authorization

In the above figure, the client sends an Allocate request to the server without credentials. Since the server requires that all requests be authenticated using OAuth, the server rejects the request with a 401 (Unauthorized) error code and STUN attribute THIRD-PARTY-AUTHORIZATION. The WebRTC client obtains access token from the WebRTC server and then tries again, this time including access token. This time, the server validates the token, accepts the Allocate request and returns an Allocate success response containing (amongst other things) the relayed transport address assigned to the allocation.

Changes specific to TURN are listed below:

- The access token can be reused for multiple Allocate requests to the same TURN server. The TURN client MUST include the ACCESS-TOKEN attribute only in Allocate and Refresh requests. Since the access token is only valid for a specific period of time, the TURN server MUST cache it so that it need not to be provided in every request within an existing allocation.

- The lifetime provided by the TURN server in the Allocate and Refresh responses MUST be less than or equal to the lifetime of the token. It is RECOMMENDED that the TURN server calculate the maximum allowed lifetime value using the formula:

```
lifetime + Delta - abs(RDnew - TS)
```

- If the access token expires then the client MUST obtain a new token from the authorization server and use it for new allocations. The client MUST use the new token to refresh existing allocations. This way client has to maintain only one token per TURN server.

10. Security Considerations

When OAuth is used the interaction between the client and the authorization server requires Transport Layer Security (TLS) with a ciphersuite offering confidentiality protection. The session key MUST NOT be transmitted in clear since this would completely destroy the security benefits of the proposed scheme. If an attacker tries to replay message with ACCESS-TOKEN attribute then the server can detect that the transaction ID as used for an old request and thus prevent the replay attack. The client may know some (but not all) of the token fields encrypted with a unknown secret key and the token can be subjected to known-plaintext attack, but AES is secure against this attack.

Threat mitigation discussed in section 5 of [I-D.ietf-oauth-pop-architecture] and security considerations in [RFC5389] are to be taken into account.

11. IANA Considerations

IANA is requested to add the following attributes to the STUN attribute registry [iana-stun],

- THIRD-PARTY-AUTHORIZATION

- ACCESS-TOKEN

12. Acknowledgements

Authors would like to thank Dan Wing, Pal Martinsen, Oleg Moskalenko, Charles Eckel and Hannes Tschofenig for comments and review. The authors would like to give special thanks to Brandon Williams for his help.

Thanks to Oleg Moskalenko for providing ticket samples in the Appendix section.

13. References

13.1. Normative References

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

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, May 2007.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, October 2008.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[iana-stun]
IANA, , "IANA: STUN Attributes", April 2011, <http://www.iana.org/assignments/stun-parameters/stun-parameters.xml>.

13.2. Informative References

[I-D.ietf-oauth-pop-architecture]
Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-00 (work in progress), July 2014.

[I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-00 (work in progress), July 2014.

[I-D.ietf-rtcweb-overview]
Alvestrand, H., "Overview: Real Time Protocols for Browser-based Applications", draft-ietf-rtcweb-overview-11 (work in progress), August 2014.

[I-D.ietf-tram-auth-problems]
Reddy, T., R, R., Perumal, M., and A. Yegin, "Problems with STUN long-term Authentication for TURN", draft-ietf-tram-auth-problems-05 (work in progress), August 2014.

[I-D.salgueiro-tram-stunbis]
Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for (NAT) (STUN)", draft-salgueiro-tram-stunbis-00 (work in progress), July 2014.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, April 2010.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

[RFC6063] Doherty, A., Pei, M., Machani, S., and M. Nystrom, "Dynamic Symmetric Key Provisioning Protocol (DSKPP)", RFC 6063, December 2010.

[RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, January 2013.

Appendix A. Sample tickets

Input data (same for all samples below):

```
//STUN SERVER NAME
server_name = "blackdow.carleon.gov";

//Shared password between AS and RS
long_term_password = "HGkj32KJGiuy098sdfaqbNjOiaz71923";

//MAC key of the session (included in the token)
```

```
   mac_key = "ZksjpweoixXmvn67534m";

   //length of the MAC key
   mac_key_length  =  20;

   //The timestamp field in the token
   token_timestamp = 92470300704768;

   //The lifetime of the token
   token_lifetime = 3600;

   //nonce for AEAD when AEAD is used
   aead_nonce = "h4j3k2l2n4b5";

   Samples:

   1)
   hkdf hash function = SHA-256,
   token encryption algorithm = AES-256-CBC
   token auth algorithm = HMAC-SHA-256

   Result:
   AS_RS key (32 bytes) = \xd\x7e\x54\x5b\x7e\x15\xc9\x81\x8c\x81\x4b\x83
                          \xdc\x4e\xce\x24\x55\xde\x73\xe\xab\x8\x8a\x94
                          \xc4\x29\xab\x45\xfd\x61\xa\xb5

   AUTH key (32 bytes) = \xd\x7e\x54\x5b\x7e\x15\xc9\x81\x8c\x81\x4b\x83
                         \xdc\x4e\xce\x24\x55\xde\x73\xe\xab\x8\x8a\x94
                         \xc4\x29\xab\x45\xfd\x61\xa\xb5

   Encrypted token (80 bytes = 48+32) =

   \x1b\xb6\x4b\x4f\xbf\x99\x6d\x60\x55\xda\xf3\x9f\xa1\xed\x3\x73\x4e
   \x1c\x95\x64\x84\xc1\xeb\xc3\x63\x9b\x70\xe6\xb8\x21\x45\xe6\x45\xa0
   \x23\xaf\xc1\xee\x87\x91\x7b\xea\xb8\x4a\x7f\x80\xb2\x0\xa5\xad\x14
   \x97\x17\xf9\xbc\xfa\xa1\xc6\x2f\x4d\xfc\xaf\xc1\xc5\x11\xc5\x55\x7d
   \xb0\x35\x58\xcf\xc6\xce\x6e\x10\x7\xd1\x98\xbd

   2)

   hkdf hash function = SHA-256,
   token encryption algorithm = AEAD_AES_256_GCM
   token auth algorithm = N/A

   Result:
   AS_RS key (32 bytes) = \xd\x7e\x54\x5b\x7e\x15\xc9\x81\x8c\x81\x4b\x83
                          \xdc\x4e\xce\x24\x55\xde\x73\xe\xab\x8\x8a\x94
                          \xc4\x29\xab\x45\xfd\x61\xa\xb5
```

```
   AUTH key = N/A

   Encrypted token (62 bytes = 34 + 16 + 12) =

   \xa8\x52\x90\x64\xc7\xd9\x3b\x6c\xe\x9\xe\xcf\x9e\x7d\x0\x70\x47\xe2
   \x99\x8d\xe3\x31\xe1\x39\x20\xed\x88\x90\x4\xd8\xcf\x82\x93\x3f\xc6\
   x4\xd1\xaa\xe6\xf5\x62\xea\x3c\x94\x45\x8\x3d\xfa\xe9\x5f\x68\x34\x6a
   \x33\x6b\x32\x6c\x32\x6e\x34\x62\x35

   3)

   hkdf hash function = SHA-1,
   token encryption algorithm = AES-128-CBC
   token auth algorithm = HMAC-SHA-256-128

   Result:
   AS_RS key (16 bytes) = \x8c\x48\x5f\x1e\x1\x3a\xc6\x50\x36\x70\x84\x37
                          \xa5\x4e\xd7\x70
   AUTH key (32 bytes) = \x8c\x48\x5f\x1e\x1\x3a\xc6\x50\x36\x70\x84\x37
                         \xa5\x4e\xd7\x70\x17\xcc\xcd\xa1\x7c\xd7\x8\x39
                         \xfa\xc8\xee\x14\xf9\x77\xb4\xcf

   Encrypted token (64 bytes = 48+16) =

   \x13\xcd\x17\x4a\xde\x54\xe1\xe6\x65\xe6\xbb\x3a\xb9\x4d\x1c\xf7\x3b
   \x60\x31\x8b\xc4\x7\x4b\x3b\x5f\x1c\xda\xf4\x60\x4\x7\x88\x8e\xc9\xc7
   \xd3\xf4\x71\x94\x87\x85\xd9\xad\xf7\x6a\xda\x77\x4e\x11\x13\x8d\x8e
   \xe8\x93\x9\x76\xa3\x85\x96\x1f\x5e\xd3\xc4\x55
```

Figure 8: Sample tickets

Authors' Addresses

Tirumaleswar Reddy
Cisco Systems, Inc.
Cessna Business Park, Varthur Hobli
Sarjapur Marathalli Outer Ring Road
Bangalore, Karnataka 560103
India

Email: tireddy@cisco.com

Prashanth Patil
Cisco Systems, Inc.
Bangalore
India

Email: praspati@cisco.com

Ram Mohan Ravindranath
Cisco Systems, Inc.
Cessna Business Park,
Kadabeesanahalli Village, Varthur Hobli,
Sarjapur-Marathahalli Outer Ring Road
Bangalore, Karnataka 560103
India

Email: rmohanr@cisco.com

Justin Uberti
Google
747 6th Ave S
Kirkland, WA
98033
USA

Email: justin@uberti.name