

TRAM  
Internet-Draft  
Intended status: Standards Track  
Expires: August 29, 2015

T. Reddy  
P. Patil  
R. Ravindranath  
Cisco  
J. Uberti  
Google  
February 25, 2015

**Session Traversal Utilities for NAT (STUN) Extension for Third Party  
Authorization  
draft-ietf-tram-turn-third-party-authz-13**

**Abstract**

This document proposes the use of OAuth 2.0 to obtain and validate ephemeral tokens that can be used for Session Traversal Utilities for NAT (STUN) authentication. The usage of ephemeral tokens ensures that access to a STUN server can be controlled even if the tokens are compromised.

**Status of This Memo**

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

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

This Internet-Draft will expire on August 29, 2015.

**Copyright Notice**

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

<a href="#">1.</a>	Introduction . . . . .	<a href="#">2</a>
<a href="#">2.</a>	Terminology . . . . .	<a href="#">3</a>
<a href="#">3.</a>	Solution Overview . . . . .	<a href="#">3</a>
<a href="#">4.</a>	Obtaining a Token Using OAuth . . . . .	<a href="#">4</a>
<a href="#">4.1.</a>	Key Establishment . . . . .	<a href="#">4</a>
<a href="#">4.1.1.</a>	DSKPP . . . . .	<a href="#">5</a>
<a href="#">4.1.2.</a>	HTTP interactions . . . . .	<a href="#">5</a>
<a href="#">4.1.3.</a>	Manual provisioning . . . . .	<a href="#">6</a>
<a href="#">5.</a>	Forming a Request . . . . .	<a href="#">7</a>
<a href="#">6.</a>	STUN Attributes . . . . .	<a href="#">7</a>
<a href="#">6.1.</a>	THIRD-PARTY-AUTHORIZATION . . . . .	<a href="#">7</a>
<a href="#">6.2.</a>	ACCESS-TOKEN . . . . .	<a href="#">7</a>
<a href="#">7.</a>	STUN Server Behaviour . . . . .	<a href="#">9</a>
<a href="#">8.</a>	STUN Client Behaviour . . . . .	<a href="#">10</a>
<a href="#">9.</a>	Usage with TURN . . . . .	<a href="#">11</a>
<a href="#">10.</a>	Operational Considerations . . . . .	<a href="#">14</a>
<a href="#">11.</a>	Security Considerations . . . . .	<a href="#">14</a>
<a href="#">12.</a>	IANA Considerations . . . . .	<a href="#">15</a>
<a href="#">13.</a>	Acknowledgements . . . . .	<a href="#">15</a>
<a href="#">14.</a>	References . . . . .	<a href="#">15</a>
<a href="#">14.1.</a>	Normative References . . . . .	<a href="#">16</a>
<a href="#">14.2.</a>	Informative References . . . . .	<a href="#">16</a>
<a href="#">Appendix A.</a>	Sample tickets . . . . .	<a href="#">18</a>
<a href="#">Appendix B.</a>	Interaction between client and authorization server	20
	Authors' Addresses . . . . .	<a href="#">22</a>

## [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 like WebRTC [[I-D.ietf-rtcweb-overview](#)] where JavaScript uses the browser functionality to make real-time audio and/or video calls, Web conferencing, and direct data transfer, ensuring this secrecy is typically not possible.

To address this problem and the ones described in [[RFC7376](#)], this document proposes the use of third party authorization using OAuth



2.0 [RFC6749] for STUN. Using OAuth 2.0, 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. Third party authorization using OAuth 2.0 for STUN explained in this specification can also be used with Traversal Using Relays around NAT (TURN) [RFC5766].

## 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].

- o WebRTC Server: A web server that supports WebRTC [I-D.ietf-rtcweb-overview].
- o Access Token: OAuth 2.0 access token.
- o 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.
- o kid: An ephemeral and unique key identifier. The kid also allows the resource server to select the appropriate keying material for decryption.

Some sections in this specification show WebRTC server as the authorization server and client as the WebRTC client, however WebRTC is intended to be used for illustrative purpose only.

## 3. Solution Overview

STUN client knows that it can use OAuth 2.0 with the target STUN server either through configuration or when it receives the new STUN attribute THIRD-PARTY-AUTHORIZATION in the error response with an error code of 401(Unauthorized).

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 content of the token is opaque to the client. The client embeds the token within a STUN request sent to the STUN server. Once the STUN server has determined



the token is valid, its services are offered for a determined period of time. Access token issued by the authorization server is explained in [Section 6.2](#). OAuth 2.0 in [\[RFC6749\]](#) defines four grant types. This specification uses the OAuth 2.0 grant type "Implicit" explained in [section 1.3.2 of \[RFC6749\]](#) where the client is issued an access token directly. The string 'stun' is defined by this specification for use as the OAuth scope parameter (see [section 3.3 of \[RFC6749\]](#)) for the OAuth token.

The exact mechanism used by a client to obtain a token from the OAuth 2.0 authorization server is outside the scope of this document. [Appendix B](#) provides an example deployment scenario of interaction between the client and authorization server to obtain a token.

#### **4. Obtaining a Token Using OAuth**

A STUN client needs to 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 third party authorization, it will return an error message indicating that the client can authorize to the STUN server using an OAuth 2.0 access token. 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 server also includes the additional STUN attribute THIRD-PARTY-AUTHORIZATION signaling the STUN client that the STUN server supports third party authorization.

Note: An implementation may choose to contact the authorization 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 authorization server, the client can obtain the token before making subsequent STUN requests.

##### **4.1. Key Establishment**

The authorization server shares a long-term secret (like asymmetric credentials) with the STUN server for mutual authentication. Symmetric-key algorithm with Hash based Message Authentication Codes (HMACs) 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 STUN server and authorization server can establish a symmetric key (K), using an out of band mechanism. If symmetric key is used then the AS-RS and AUTH keys will be derived from K. The AS-RS key



is used for encrypting the self-contained token and the 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 procedure for establishment of the symmetric key is outside the scope of this specification, and this specification does not mandate support of any given mechanism. [Section 4.1.1](#), [Section 4.1.2](#) and [Section 4.1.3](#) show examples of mechanisms that can be used.

#### **[4.1.1](#). DSKPP**

The STUN and AS 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 (HKDF)-based key derivation function (HKDF) [[RFC5869](#)] and the default hash function MUST be 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-SHA1 [[RFC2104](#)] 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, "AUTH key", 20) -> AUTH key
3. HKDF-Expand(PRK, "AS-RS key", 32) -> AS-RS key

If Authenticated Encryption with Associated Data (AEAD) algorithm defined in [[RFC5116](#)] is used then there is no need to generate the AUTH key. AEAD simplifies implementation, since it involves only one key, the user of AEAD need not consider relative order of encrypting and authenticating the token and other benefits of AEAD are explained in [Section 1.3 of \[RFC5116\]](#).

#### **[4.1.2](#). HTTP interactions**

The STUN and AS servers could choose to use REST API over HTTPS to establish a symmetric key. HTTPS MUST be used for mutual authentication and confidentiality. 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 JavaScript Object Notation (JSON) [[RFC7159](#)] 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

-----

k - Long-term key (K)  
exp - identifies the time after which the key expires.

example:

```
{
  "k" :
  "ESIZRFVmd4iZABEiM0RVZgKn6WjLaTC1FXAghRMVTzkBGNaan496523WIISkerLi",
  "exp" : 1300819380,
  "kid" : "22BIjxU93h/IgwEb"
  "enc" : A256CBC-HS512
}
```

The AS-RS, AUTH keys are derived from K using HKDF as discussed in [Section 4.1.1](#). The authorization server must also signal kid to the STUN server which will be used to select the appropriate keying material for decryption. The parameter "k" is defined in Section 6.4.1 of [[I-D.ietf-jose-json-web-algorithms](#)], "enc" is defined in Section 4.1.2 of [[I-D.ietf-jose-json-web-encryption](#)], "kid" is defined in Section 4.1.4 of [[I-D.ietf-jose-json-web-signature](#)] and "exp" is defined in Section 4.1.4 of [[I-D.ietf-oauth-json-web-token](#)]. A256CBC-HS512 and other authenticated encryption algorithms are defined in [[I-D.ietf-jose-json-web-algorithms](#)]. 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

The STUN and AS servers could be manually configured with a symmetric key (K) and kid. If manual provisioning is supported, support MUST also be provided for AES\_256\_CBC\_HMAC\_SHA\_512 (explained in



[[I-D.ietf-jose-json-web-algorithms](#)]) as the authenticated encryption algorithm.

Note : The mechanism specified in [Section 4.1.3](#) compared to DSKPP and REST lacks 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 do not 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-ups 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 [\[RFC5389\]](#)). If the client is able to comprehend THIRD-PARTY-AUTHORIZATION it MUST ensure that third party authorization takes precedence over first party authentication (explained in [section 10 of \[RFC5389\]](#)). If the client does not support or is not capable of doing third party authorization then it defaults to first party authentication.

### **6.2. ACCESS-TOKEN**

The access token is issued by the authorization server. OAuth 2.0 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 \[RFC5389\]](#)). The 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



the client MUST NOT examine the token. 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;  
        uint8_t padding_length;  
        uint8_t padding[padding_length];  
    } encrypted_block;  
    opaque mac[mac_length];  
    uint8_t mac_length;  
} token;
```

Figure 1: 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 MUST be greater than or equal to the "expires\_in" parameter defined in [section 4.2.2 of \[RFC6749\]](#), otherwise resource server could revoke the token but the client would assume that the token has not expired and would not refresh the token.



`padding_length`: The padding length MUST be such that the total size of the `encrypted_block` structure is a multiple of the cipher's block length.

`padding`: Padding that is added to force the length of the plaintext to be an integral multiple of the block cipher's block length.

`encrypted_block`: The `encrypted_block` is encrypted using the symmetric long-term key established between the STUN server and the authorization server. Shown in Figure 3 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.

`mac_length`: Length of the mac field.

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)`

- \* Initialization vector can be set to zero because the `encrypted_block` in each access token will not be identical and hence will not result in generation of identical ciphertext.

- 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 for AEAD algorithms is explained in [\[RFC5116\]](#).

## 7. STUN Server Behaviour

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:  $\text{Lifetime} + \text{Delta} > \text{abs}(\text{RDnew} - \text{TS})$ . 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).
- o 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).
- o 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.

If a STUN server receives an ACCESS-TOKEN attribute unexpectedly (because it had not previously sent out a THIRD-PARTY-AUTHORIZATION), it will respond with an error code of 420 (Unknown Attribute) as specified in [Section 7.3.1 of \[RFC5389\]](#).

## **8. STUN Client Behaviour**

- o The client looks for the MESSAGE-INTEGRITY attribute in the response. If MESSAGE-INTEGRITY is absent or the value computed for message integrity using mac\_key does not match the contents of the MESSAGE-INTEGRITY attribute then the response MUST be discarded.



- o 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. 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.

Consider the following example that illustrates the use of OAuth 2.0 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.

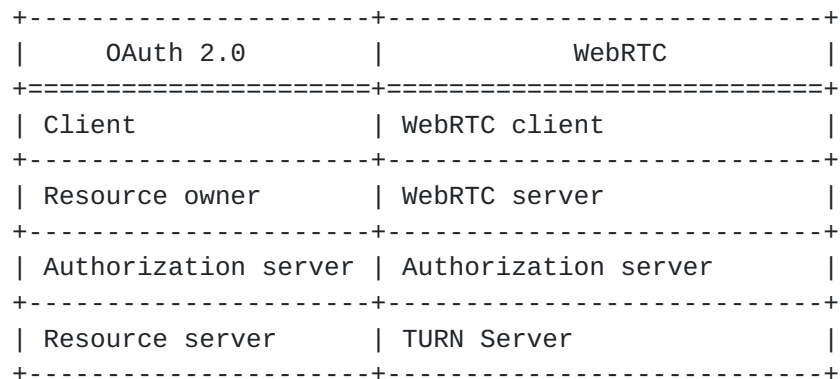
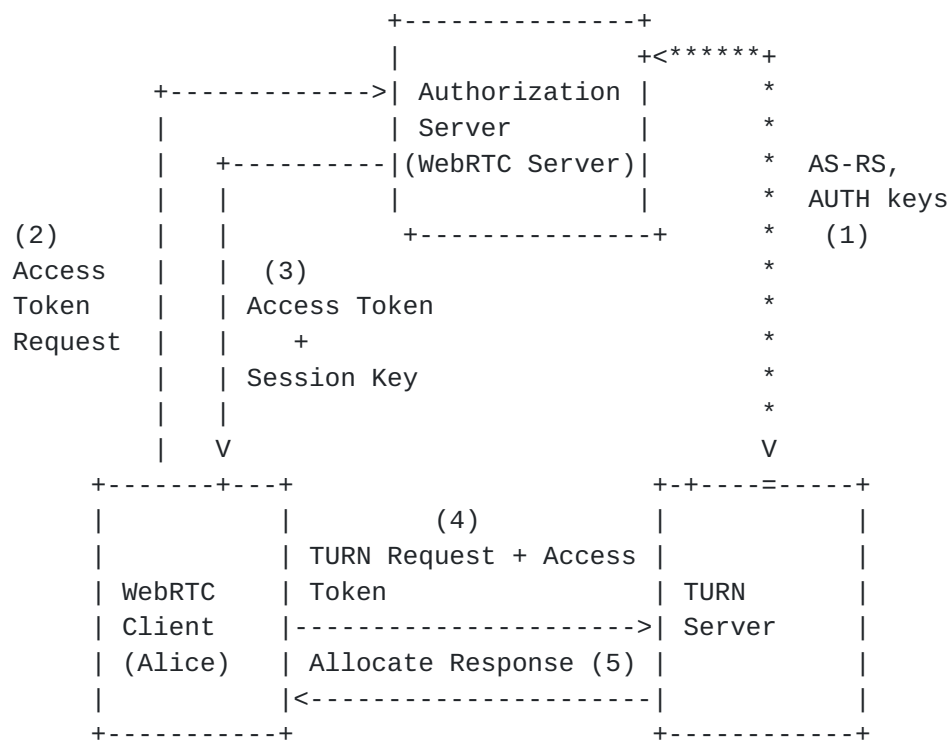


Figure 2: OAuth terminology mapped to WebRTC terminology

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 and kid. The TURN client conveys the access token and other OAuth 2.0 parameters learnt from the authorization server to the 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 3.





User : Alice

\*\*\*\*: Out-of-Band Long-Term Key Establishment

Figure 3: Interactions

In the below figure, the client sends an Allocate request to the server without credentials. Since the server requires that all requests be authenticated using OAuth 2.0, 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.



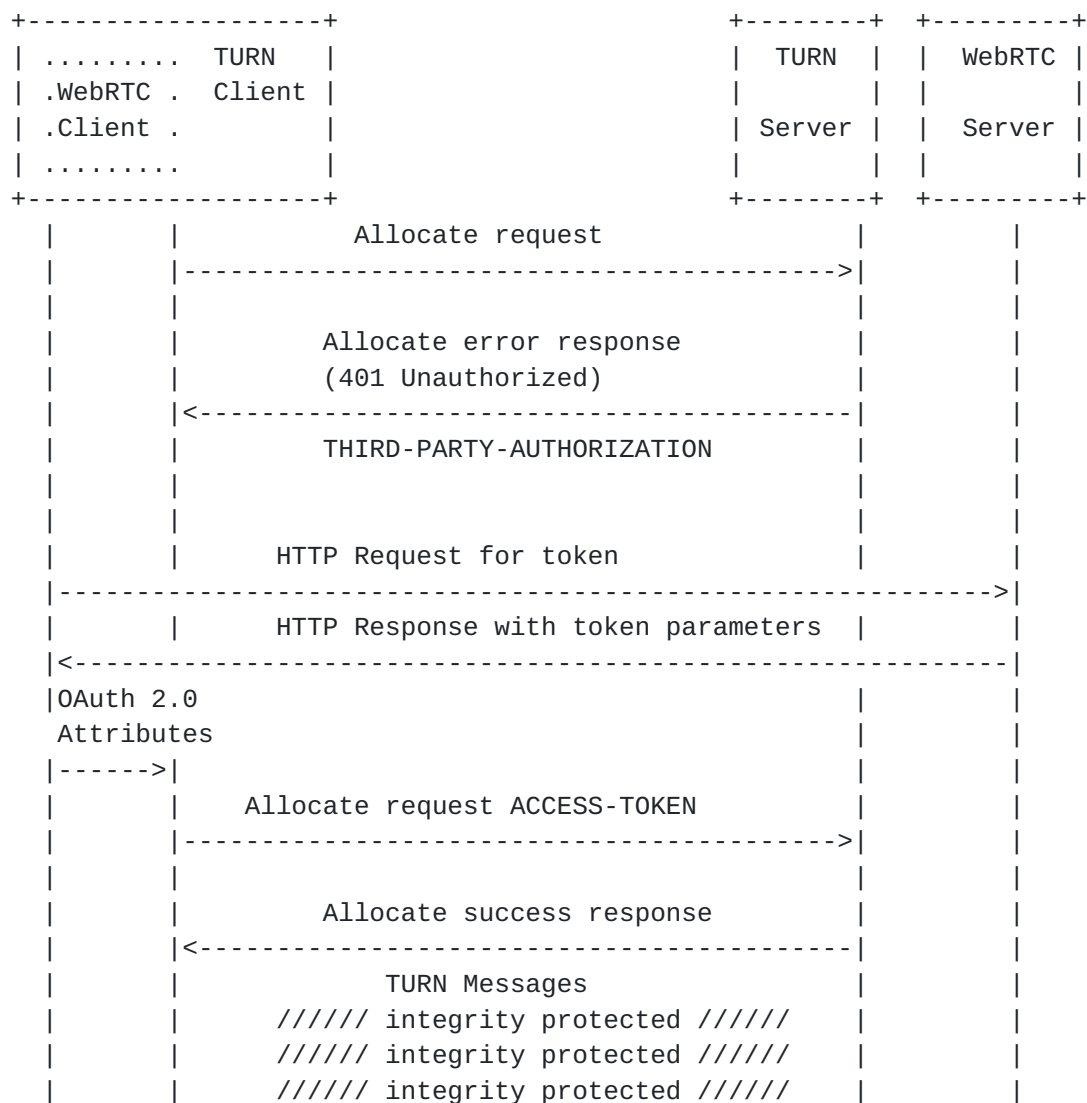


Figure 4: TURN Third Party Authorization

Changes specific to TURN are listed below:

- o 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 valid for a specific period of time, the TURN server can cache it so that it can check if the access token in a new allocation request matches one of the cached tokens and avoids the need to decrypt the token.
- o 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:





$$\text{lifetime} + \text{Delta} - \text{abs}(\text{RDnew} - \text{TS})$$

The RECOMMENDED value for the allowed Delta is 5 seconds.

- o 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. Operational Considerations**

The following operational considerations should be taken into account:

- o Each authorization server should maintain the list of STUN servers for which it will grant tokens, and the long-term secret shared with each of those STUN servers.
- o If manual configuration ([Section 4.1.3](#)) is used to establish symmetric keys, the necessary information which includes long-term key (K), encryption and HMAC algorithms have to be configured on each authorization server and STUN server for each kid. The client obtains the session key and HMAC algorithm from the authorization server in company with the token.
- o When a STUN client sends a request to get access to a particular STUN server (S) the authorization server must ensure that it selects the appropriate kid, access-token depending on the server S.

## **11. Security Considerations**

When OAuth 2.0 is used the interaction between the client and the authorization server requires Transport Layer Security (TLS) with a ciphersuite offering confidentiality protection and the guidance given in [[I-D.ietf-uta-tls-bcp](#)] must be followed to avoid attacks on TLS. The session key MUST NOT be transmitted in clear since this would completely destroy the security benefits of the proposed scheme. An attacker trying to replay message with ACCESS-TOKEN attribute can be mitigated by frequent changes of nonce value as discussed in [section 10.2 of \[RFC5389\]](#). 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.

An attacker may remove the THIRD-PARTY-AUTHORIZATION STUN attribute from the error message forcing the client to pick first party



authentication, this attack may be mitigated by opting for Transport Layer Security (TLS) [[RFC5246](#)] or Datagram Transport Layer Security (DTLS) [[RFC6347](#)] as a transport protocol for Session Traversal Utilities for NAT (STUN), as defined in [[RFC5389](#)] and [[RFC7350](#)].

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

## **12. IANA Considerations**

[Paragraphs below in braces should be removed by the RFC Editor upon publication]

[IANA is requested to add the following attributes to the STUN attribute registry [[iana-stun](#)], The THIRD-PARTY-AUTHORIZATION attribute requires that IANA allocate a value in the "STUN attributes Registry" from the comprehension-optional range (0x8000-0xBFFF)]

This document defines the THIRD-PARTY-AUTHORIZATION STUN attribute, described in [Section 6](#). IANA has allocated the comprehension-optional codepoint TBD for this attribute.

[The ACCESS-TOKEN attribute requires that IANA allocate a value in the "STUN attributes Registry" from the comprehension-required range (0x0000-0x3FFF)]

This document defines the ACCESS-TOKEN STUN attribute, described in [Section 6](#). IANA has allocated the comprehension-required codepoint TBD for this attribute.

## **13. Acknowledgements**

Authors would like to thank Dan Wing, Pal Martinsen, Oleg Moskalenko, Charles Eckel, Spencer Dawkins, Hannes Tschofenig, Yaron Sheffer, Tom Taylor, Christer Holmberg, Pete Resnick, Kathleen Moriarty, Richard Barnes, Stephen Farrell and Alissa Cooper for comments and review. The authors would like to give special thanks to Brandon Williams for his help.

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

## **14. References**



### **14.1. 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>>.

### **14.2. Informative References**

- [I-D.ietf-jose-json-web-algorithms]  
Jones, M., "JSON Web Algorithms (JWA)", [draft-ietf-jose-json-web-algorithms-40](#) (work in progress), January 2015.
- [I-D.ietf-jose-json-web-encryption]  
Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", [draft-ietf-jose-json-web-encryption-40](#) (work in progress), January 2015.
- [I-D.ietf-jose-json-web-signature]  
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", [draft-ietf-jose-json-web-signature-41](#) (work in progress), January 2015.
- [I-D.ietf-oauth-json-web-token]  
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", [draft-ietf-oauth-json-web-token-32](#) (work in progress), December 2014.



[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-13](#) (work in progress), November 2014.

[I-D.ietf-tram-stunbis]

Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", [draft-ietf-tram-stunbis-01](#) (work in progress), February 2015.

[I-D.ietf-uta-tls-bcp]

Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of TLS and DTLS", [draft-ietf-uta-tls-bcp-11](#) (work in progress), February 2015.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), February 1997.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.

[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.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", [RFC 6347](#), January 2012.





- [RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", [RFC 6819](#), January 2013.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", [RFC 7159](#), March 2014.
- [RFC7350] Petit-Huguenin, M. and G. Salgueiro, "Datagram Transport Layer Security (DTLS) as Transport for Session Traversal Utilities for NAT (STUN)", [RFC 7350](#), August 2014.
- [RFC7376] Reddy, T., Ravindranath, R., Perumal, M., and A. Yegin, "Problems with Session Traversal Utilities for NAT (STUN) Long-Term Authentication for Traversal Using Relays around NAT (TURN)", [RFC 7376](#), September 2014.

## [Appendix A](#). Sample tickets

Input data (same for all samples below):

```
//STUN SERVER NAME
```

```
server_name = "blackdow.carleon.gov";
```

```
//Shared key between AS and RS
```

```
long_term_key = \x48\x47\x6b\x6a\x33\x32\x4b\x4a\x47\x69\x75\x79  
                \x30\x39\x38\x73\x64\x66\x61\x71\x62\x4e\x6a\x4f  
                \x69\x61\x7a\x37\x31\x39\x32\x33;
```

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

```
mac_key = \x5a\x6b\x73\x6a\x70\x77\x65\x6f\x69\x78\x58\x6d\x76\x6e  
          \x36\x37\x35\x33\x34\x6d;
```

```
//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 = \x68\x34\x6a\x33\x6b\x32\x6c\x32\x6e\x34\x62\x35;
```

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) = \xd8\xa4\x54\x1\x68\xb\xb8\x7c\x6c\x86  
 \xc5\x29\x3e\x35\x33\x91\x9b\x51\xfc\x9  
 \xde\x25\x19\xa9\xa3\x56\x2b\x51\x7b\xe2\x76\x4f

AUTH key (32 bytes) = \x72\xe1\xf9\xc9\xfe\xb9\xe7\xd8\x2a\xcc\x2d  
 \xc3\x33\xd4\x2f\xc1\x7f\x3e\x36\x29\xfa\xf8\x50  
 \x16\x5c\xcf\x60\xd1\x67\xb9\x8f\x76

Encrypted token (80 bytes = 48+32) =

\x26\x6\x38\x82\xfa\x74\xa6\x2\x6a\xa6\xa4\x37\x30\xac\x83\xc6\x45\x51  
 \x5c\x9e\x3e\xea\x86\x11\xb9\x4e\x8c\x27\x38\x8a\x51\xa7\xd\xc\x74\x72  
 \xdf\xb8\x12\x6a\x8e\x17\xb3\x50\x16\x10\xee\x7e\x53\xe1\x90\x6e\x3b\x37  
 \xda\x17\x57\xb1\xa2\x6\x64\x5c\x63\x79\x33\x34\x7d\xab\x71\x56\x98\x2b  
 \x1\xa6\x66\x29\xce\xb1\xa5\xb4

2)

hkdf hash function = SHA-256,  
 token encryption algorithm = AEAD\_AES\_256\_GCM  
 token auth algorithm = N/A

Result:

AS\_RS key (32 bytes) = \xd8\xa4\x54\x1\x68\xb\xb8\x7c\x6c\x86\xc5\x29\x3e  
 \x35\x33\x91\x9b\x51\xfc\x9\xde\x25\x19\xa9\xa3\x56  
 \x2b\x51\x7b\xe2\x76\x4f

AUTH key = N/A

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

\xd4\x86\x5c\x5d\x59\xfb\x3f\xe3\xf6\xf1\xd8\xc3\x22\xc2\x22\x26\x8d\x2e\xf0  
 \xbe\x2\x5b\xbd\x13\x49\x89\x6e\xa5\xc5\x51\xee\xee\x7f\xd9\xe4\x41\xd7\xcb  
 \x51\x20\x40\xcc\xc5\x53\x90\x2f\xdc\xbb\x8d\x53\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) =
\x22\x93\xd1\x4\xae\x29\x73\x3e\xcd\x1a\x18\xbd\x56\x45\x63\x67
AUTH key (32 bytes) =
\x00\xd2\x70\x78\x60\xc7\x54\xbb\x34\x3e\x4c\xd0\xcc\x03\x66\x9c

\x2e\x1a\x89\xe0\xee\x97\xb7\x8e\x36\x4b\x76\xb0\x39\x46\x19\x4a

Encrypted token (64 bytes = 48+16) =

\xf8\xef\x95\xdc\x6b\x35\x50\x8a\x6a\x36\x8f\xd9\xdc\x51\x92\x0c\x39\x2f\xcb\xfb\x1e
\x2f\x66\x8d\xbc\x50\xb8\x30\x7c\xd9\x04\xf6\x27\xb0\x9e\x73\xdc\x15\xc8\x96\xfb\x4e
\x3b\x5b\xe3\xc7\x54\x20\xc0\xd2\x90\x88\x08\x3d\x72\x9c\xd2\x17\x65\x43\x8f\xaa
\x41\x48

```

Figure 5: Sample tickets

## **Appendix B. Interaction between client and authorization server**

Client makes an HTTP request to an authorization server to obtain a token that can be used to avail itself of STUN services. The STUN token is returned in JSON syntax [\[RFC7159\]](#), along with other OAuth 2.0 parameters like token type, key, token lifetime and kid defined in [\[I-D.ietf-oauth-pop-key-distribution\]](#).



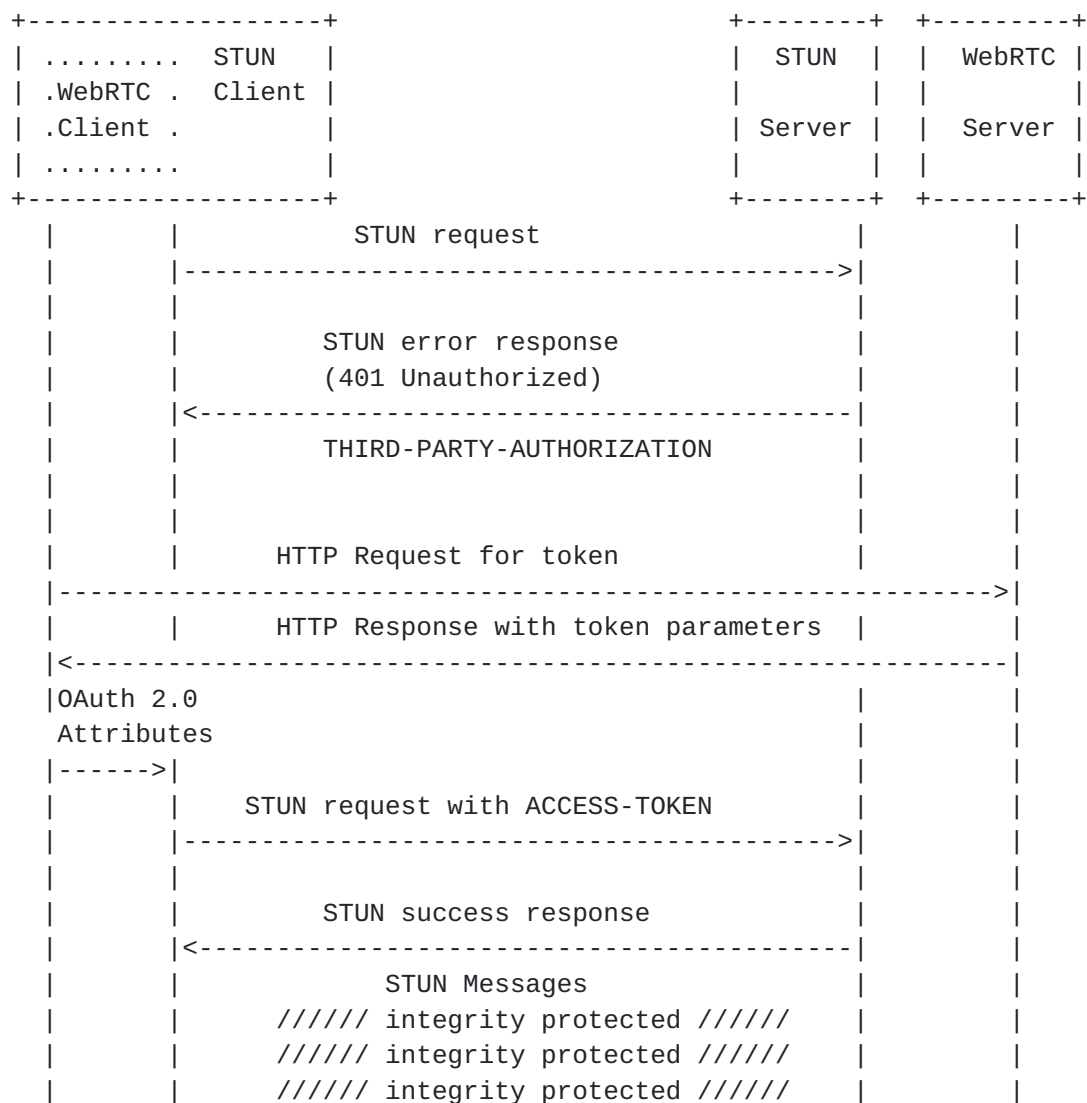


Figure 6: STUN Third Party Authorization

[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):





```
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 7: Request

[I-D.ietf-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 [\[I-D.ietf-oauth-pop-key-distribution\]](#). The authorization server selects an HMAC algorithm from the list of algorithms the client 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 the 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+Z3vkJ1Q5pWbfSRro0kWbn",
  "token_type": "pop",
  "expires_in": 1800,
  "kid": "22BIjxU93h/IgwEb",
  "key": "v51N620M65kyMvfTI080"
  "alg": "HMAC-SHA-256-128"
}
```

Figure 8: Response



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

