SIPCore
Internet-Draft
Updates: 3261 (if approved)
Intended status: Standards Track
Expires: September 9, 2016

R. Shekh-Yusef, Ed.
Avaya
V. Pascual
Oracle
C. Holmberg
Ericsson

March 8, 2016

The Session Initiation Protocol (SIP) OAuth
draft-yusef-sipcore-sip-oauth-03

Abstract

This document defines an authorization framework for SIP that is based on the OAuth 2.0 framework, and adds a simple identity layer on top of that, based on the OpenID Connect Core 1.0, to enable Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User.

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 September 9, 2016.

Table of Contents

1. Introduction

The SIP protocol [RFC3261] uses the framework used by the HTTP protocol for authenticating users, which is a simple challenge-response authentication mechanism that allows a server to challenge a client request and allows a client to provide authentication information in response to that challenge.

The SIP protocol does not have an authorization framework to allow the system to control access to various services provided by the system.

OAuth 2.0 [RFC6749] defines a token based authorization framework to allow clients to access resources on behalf of their user. It also defines four types of authorization grants, which the client uses to request the access token.

The OpenID Connect 1.0 [OPENID] specifications defines a simple identity layer on top of the OAuth 2.0 protocol, which enables Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User.

This document defines an authorization framework for SIP that is based on the OAuth 2.0 framework, and adds the identity layer on top of that, based on the OpenID Connect Core 1.0 specification

1.1. 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].

1.2. Definitions

Types of SIP services:

* Basic SIP Services: make/receive call, transfer, call forward, etc.

* Advanced SIP Services: services provided by SIP application servers, e.g. Voice Mail, Conference Services, Presence, IM, ...

Single Sign-On (SSO)

SSO is a property that allows the user to be authenticated once and as a result have access to multiple services in the system.

Authentication

The process of verifying the identity of a user trying to get access to some network services.

Authorization

The process of controlling an authenticated user access to network services and the level of service provided to the user.

1.3. Use Cases

1.3.1. Enterprise SSO

An enterprise is interested in providing its users with an SSO capability to the various corporate services. The enterprise has an authorization server for controlling the user access to their network and would like to extend that existing authorization server to control the user access to the various services provided by their SIP network.

The user is expected to provide his corporate credentials to login to the corporate network and get different types of services, regardless of the protocol used to provide the service, and without the need to create different accounts for these different types of services.

1.3.2. 3GPP

The 3GPP network has a requirement to allow a user using a WebRTC IMS Client (WIC) to authenticate to a WebRTC Authorization Function (WAF) and in response be given an access token that allows the user to register and get service from the 3GPP SIP network.

The WIC downloads an IMS webpage from the WebRTC Web Server Function (WWSF) using HTTP. The WIC then requests an access token from the WAF using HTTP, which the WIC then uses to register to the SIP network throught the P-CSCF enhanced for WebRTC (eP-CSCF) element.

1.3.3. Confidential SIP Hardphone

A SIP hardphone with rich UI, that has the capability to maintain the confidentiality of user's crecentials, is used to authenticate to an authorization server, get a token, and use that token to register and get service from the SIP network.

When the phone interacts with the authorization server and gets challenged to provide credentials, the phone will prompt the user to enter his credentials which will be used to authenticate to the authorization server.

1.3.4. Public SIP Hardphone

A SIP hardphone with limited UI capabilities, that is incapable of maintaining confidentiality of user's crecentials, is used to register with the SIP network by providing an access code obtaied from an authorization server.

When the phone interacts with the SIP network without providing any credentials, the phone gets challenged to provide proper credentials.

The user will then use an out of band method, e.g browser, to authenticate to the authorization server and get a short-lived numeric access code.

The user will then use the phone's keypad to provide the numeric access code to the SIP phone. The phone will then use the access code to register and get service from the SIP network. The SIP Proxy will exchange the access code with access token from the authorization server.

1.3.5. SIP SSO

An enterprise is interested in providing its users with an SSO capability to the various corporate SIP services.

The enterprise wants to control the services provided to their SIP users and the level of service provided to the user by their SIP application servers without the need to create different accounts for these services.

The enterprise wants to utilize an existing authentication mechanism provided by SIP, but would like to be able to control who gets access to what service and when.

The user is expected to use his SIP credentials to login to the SIP network and get access to the basic services, and to get access to the services provided by the various SIP application servers without being challenged to provide credentials for each type of service.

1.4. Roles

resource owner

An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

In a typical SIP network, it is the management element in the system that acts as a resource owner.

resource server

The server hosting the protected resources or services, capable of accepting and responding to protected resource and services requests using access tokens.

OAuth 2.0 client

An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

SIP client

An application making requests to access SIP services on behalf of the end-user.

authorization server

The server issuing tokens to the OAuth 2.0 client or SIP Client after successfully authenticating the resource owner and obtaining authorization.

proof-of-possession (pop)

A hash used by one party to prove to another party that it is in possession of some shared credentials, without sending the credentials on the wire.

1.5. ID Token

ID token, as defined in the OpenID document, is a security token that contains claims about the authentication of an end-user by an authorization server.

1.6. Authentication Types

There are two types of user authentications in SIP:

- Proxy-to-User: which allows a server that is providing a service to authenticate the identity of a user before providing the service.

- User-to-User: which allows a user recieving a request to authenticate the identity of the remote user before processing the request.

The mechanism defined in this document addresses the proxy-to-user authentication only. For user-to-user authentication refer to the mechanism defined in [STIR].

2. Benefits

This section describes the benefit of this authorization framework:

2.1. Single Sign-On

With the existing mechanism, the proxy and application servers might need to challenge many of the requests sent by a client, which adds traffic that could be avoided with this authorization mechanism.

Single Sign-On is a property that allows the user to be authenticated once and as a result have access to multiple services in the system.

This authorization mechanism would enable Single Sign-On, as the user will be authenticated once and as a result given a token and a refresh token to allow the user access to various services based on the token scope.

2.2. Service Authorization

This authorization mechanism allows the system to centrally control the services provided to the user, e.g conference services, voice mail, etc. The mechanism also allow control over the level of services provided to the user; for example, if the user is given access to conference services, the system controls whether the user gets access to video conference services or only audio conference services.

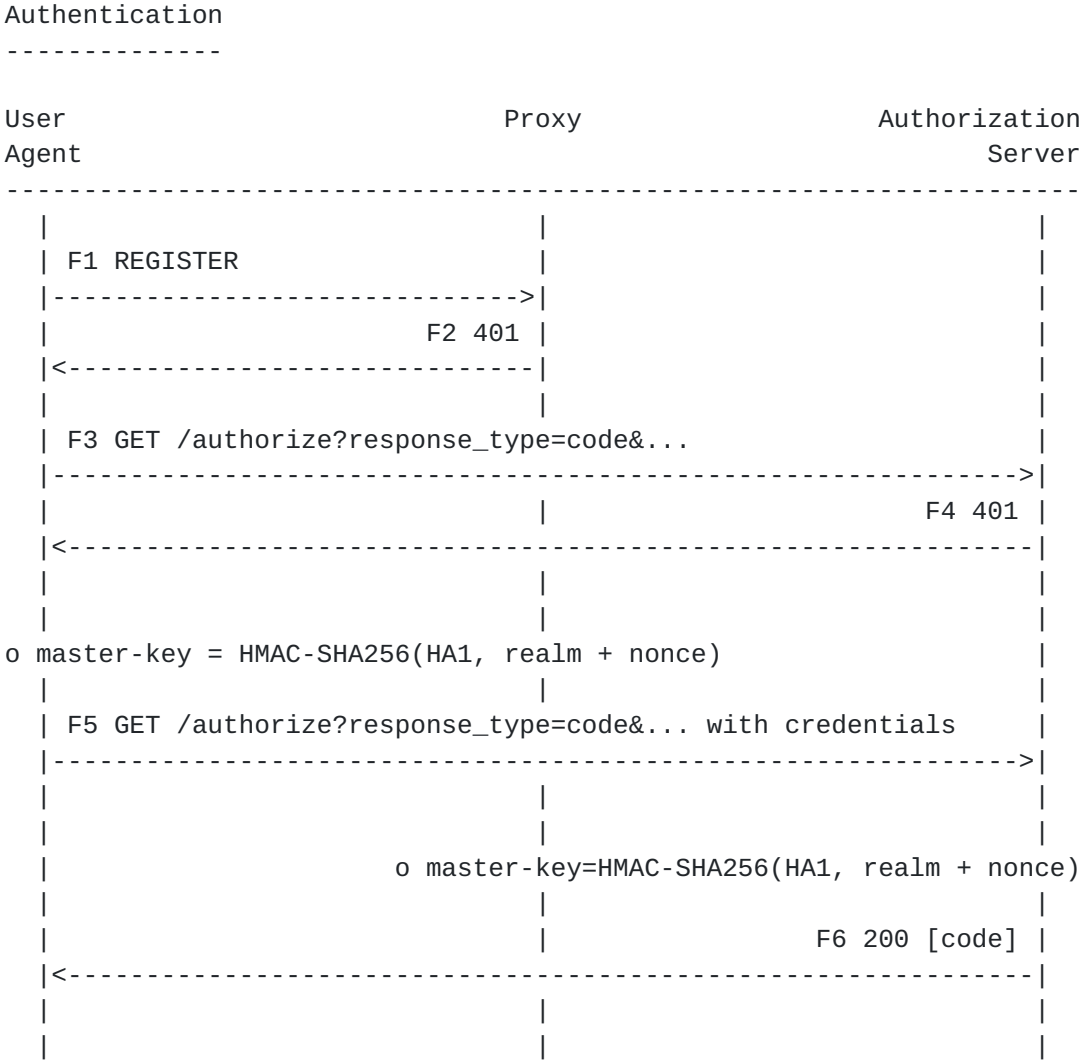
2.3. Third-Party Authentication

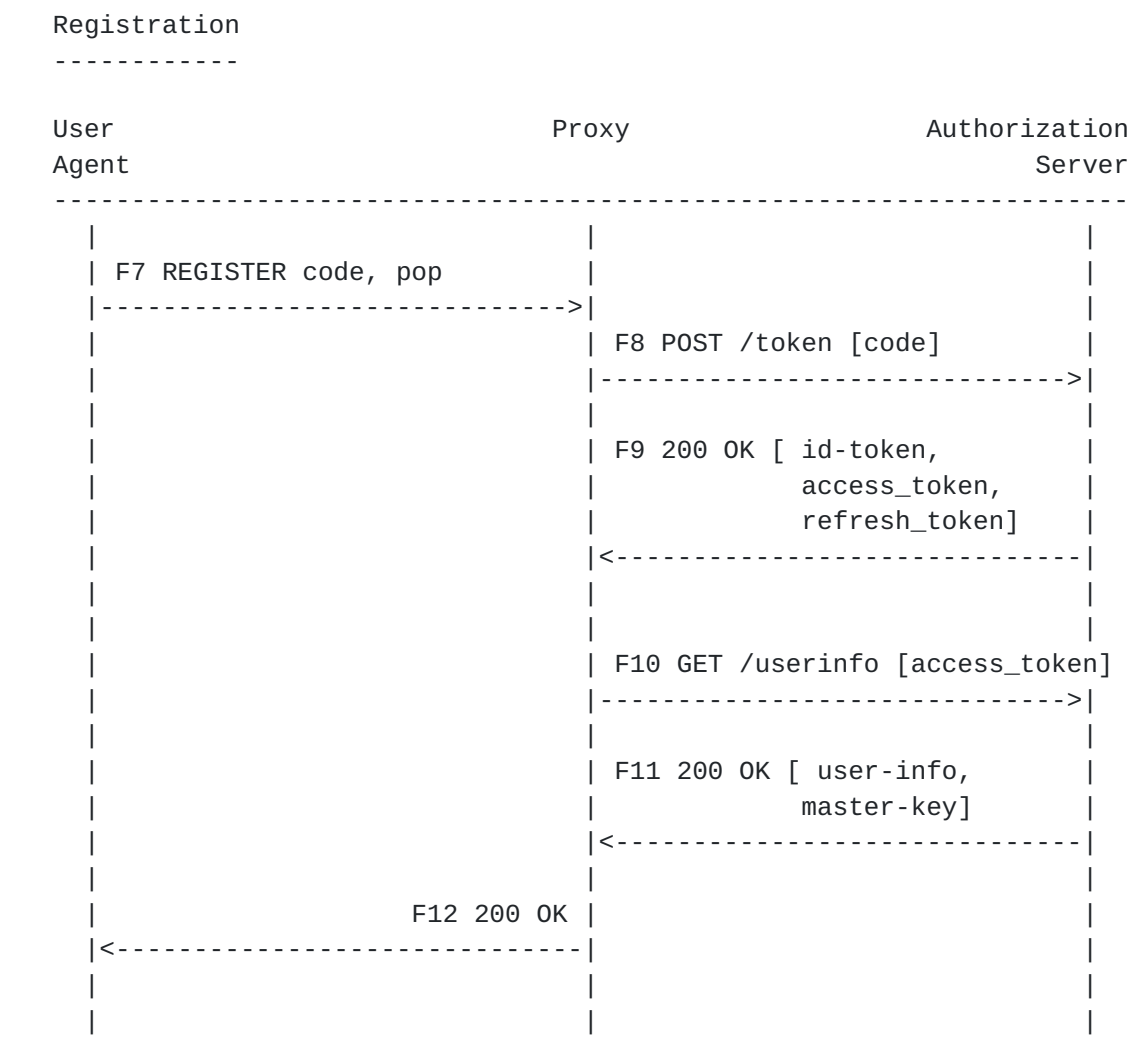
This authorization mechanism allows the user to be authenticated and obtain tokens using some Third-Party Authorization mechanism and still get services from the system.

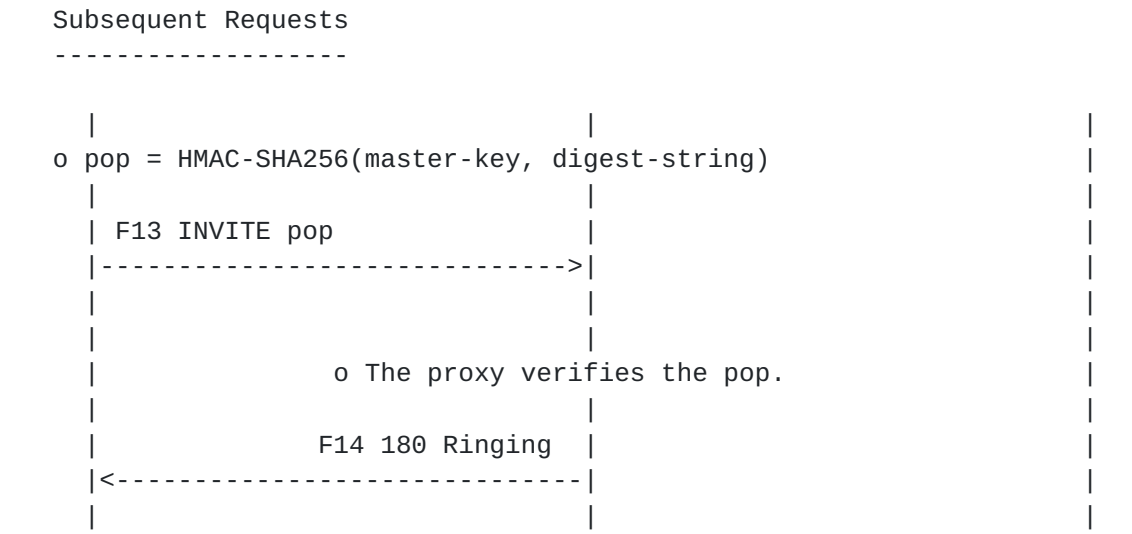
3. Authorization Code Grant type

3.1. Operations Overview

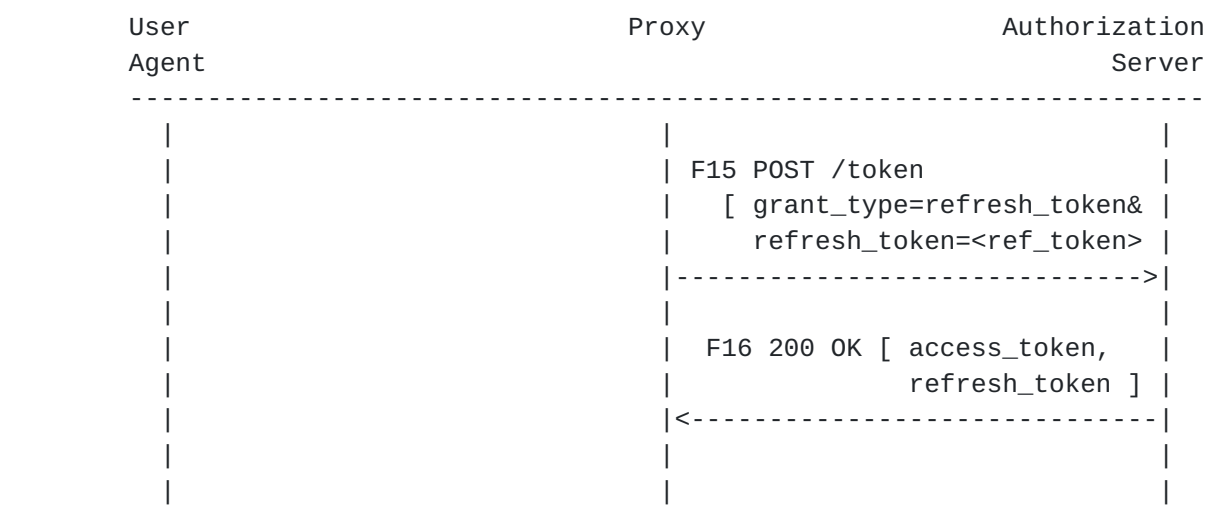
The following figure provides a high level view of flow of messages for the Authorization Code Grant type:




Registration
User Agent
Proxy
Authorization Server
F7 REGISTER code, pop
F8 POST /token [code]
F9 200 OK [id-token, access_token, refresh_token]
F10 GET /userinfo [access_token]
F11 200 OK [user-info, master-key]
F12 200 OK


Subsequent Requests
o pop = HMAC-SHA256(master-key, digest-string)
F13 INVITE pop
o The proxy verifies the pop.
F14 180 Ringing

Token Refresh



The UA initially sends a REGISTER request (F1) without providing any credentials.

The proxy challenges the UA by responding with 401 (F2) that includes the address of the Authorization Server.

[[OPEN ISSUE]] How should the UA be redirected to the Authorization Server: 1. New SIP parameter? 2. Extend the Bearer scheme? 3. Define a new Scheme?

The UA will then contact the Authorization Server without providing any credentials in the first request (F3). The Authorization Server challenges the request using the Digest scheme (F4), and the client retries the request (F5) and provides the user's credentials.

The Authorization Server verifies the request from the client; if the verification is successful, the Authorization Server responds with 200 OK (F6) and includes a code in the body part.

The UA then retries the request (F7) and include the code in the body of the request. The proxy then contacts the Authorization Server and exchanges the code for tokens (F8 and F9), and gets the user information (F10 and F11). The proxy then sends 200 OK to the UA to complete the registration process.

3.2. Authentication

The UA initiates the process by sending a REGISTER request (F1) to the proxy. The proxy will redirect the UA to the Authorization Server by responding with 401 (F2) that includes the address of the Authorization Server in the form of an HTTP URI.

The UA constructs the initial request (F3) to the Authorization Server without providing any user credentials, but with the following URI parameters in the query component:

response_type (REQUIRED)

Value MUST be set to "code".

user_id (REQUIRED)

The user's identification with the Authorization Server.

scope (OPTIONAL)

The scope of the access request

state (RECOMMENDED)

The value of this parameter is a nonce created by the client to prevent replay attack. The nonce is a uniquely generated value for each request. This parameter might not be included with the initial request that does not include credentials (F3).

The Authorization Server uses the user identification specified in the user_id parameter to verify that the user has an account in the system, and then challenges the request by responding with 401 (F4) with Digest scheme.

The UA will generate a master-key that is based on an HMAC-Hash algorithm, e.g. HMAC-SHA256, that takes an input the user's HA1 and the concatenation of realm and nonce received in the challenge from the server.

The UA will then send a new authorization request (F5), but this time include the credentials requested by the server. The UA will use the same parameters values used in the initial authorization request with the exception of the state parameter which will get a new nonce value.

When the server receives the request with the credentials (F5), the server will verify the digest provided by the UA; if that is

successful, the server will respond with 200 OK (F6) and include a code in the body of the response with the following parameters:

grant_type (REQUIRED)

Value MUST be set to "authorization_code".

code (REQUIRED)

The authorization code received from the authorization server.

The server then generates a master-key that is based on an HMAC-Hash algorithm, e.g. HMAC-SHA256, that takes an input the user's HA1, and the concatenation of realm and nonce sent in the challenge (F4) to the client.

3.3. Registration

The UA will send a new REGISTER request (F7) and include the code in the body of the request with the following parameters:

grant_type (REQUIRED)

Value MUST be set to "authorization_code".

code (REQUIRED)

The authorization code received from the authorization server.

The proxy sends a POST request (F8) to the Authorization Server and include the following parameters in the body:

grant_type (REQUIRED)

Value MUST be set to "authorization_code".

code (REQUIRED)

The authorization code received from the authorization server.

If the request is valid and authorized, the authorization server responds with a 200 OK (F9) with id_token, access token, and refresh_token in the body.

The UA sends a GET request (F10) to the Authorization Server to fetch the user information, and includes the access token in the body of the request. In response the Autorization Server will respond with

200 OK and include the user information and the master-key associated with the user in the body part.

The proxy then responds with 200 OK (F12) to the UA to complete the registration process.

3.4. Subsequent Requests

When the UA wants to send any request to the proxy, it MUST include the Authorization header and use the Bearer scheme to carry the proof-of-possession of the master-key.

The pop is calculated using the master-key as follows:

```
pop = HMAC-SHA256(master-key, digest-string)
```

The following is an example of an Authorization header with Bearer scheme:

```
Authorization: Bearer pop=<pop>
```

See rfc4474, section 9, for the SIP headers to hash to create digest-string.

[[OPEN ISSUE]] The Bearer scheme is used to deliver tokens without providing any proof of possession. We probably need to use different scheme later on.

3.5. Token Refresh

The proxy makes a refresh request to the Authorization Server by sending a refresh POST request (F13) that includes a body with the grant_type and the refresh_token.

For example:

```
grant_type=refresh_token&refresh_token=<refresh_token>
```

If the proxy fails to refresh the token, then it MUST challenge the next request from the UA, and as a result the UA MUST go through the authorization process again to obtain new tokens.

3.6. Services

When the UA tries to access a service on behalf of a user, e.g. Voice Mail Service, the proxy forwards the request to the server providing the service and MUST include an Authorization header with the Bearer scheme that carries the token needed to get service, as follows:

```
Authorization: Bearer token=<token>
```

4. Implicit Grant Type

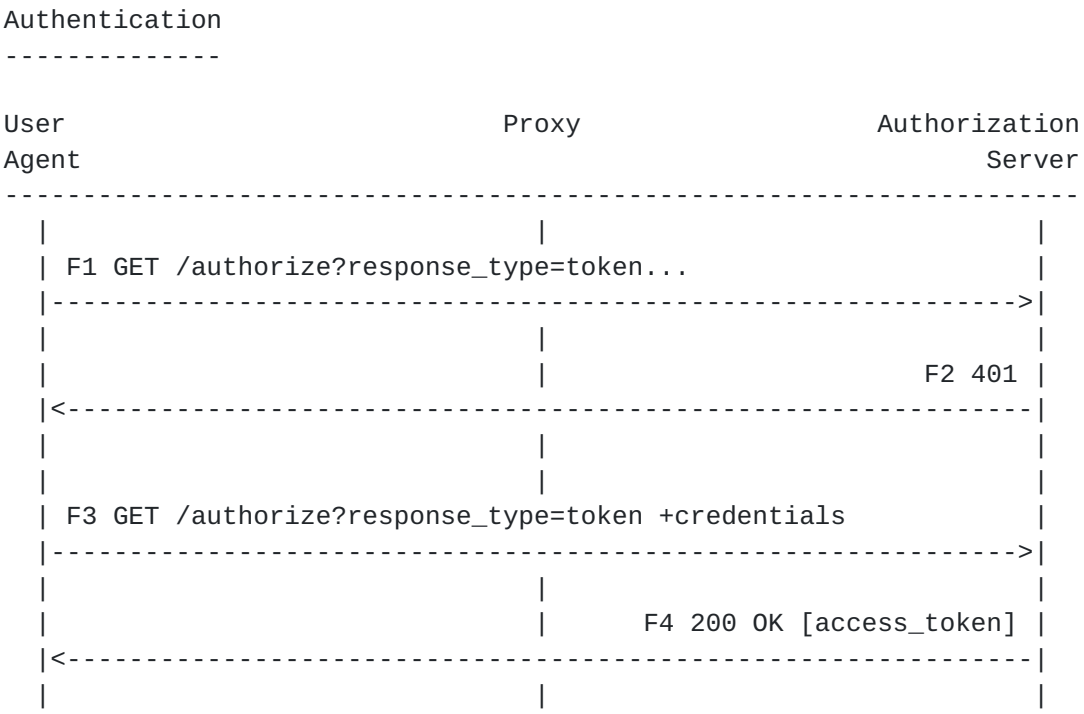
The impicit grant type is used by the SIP UA to directly obtain access tokens from the Authorization Server to be able to register and get service from the SIP network.

This grant type does not support the issuance of refresh tokens, which means that the SIP UA must re-authenticate again to the Authorization Server to get a new token before the current token expires.

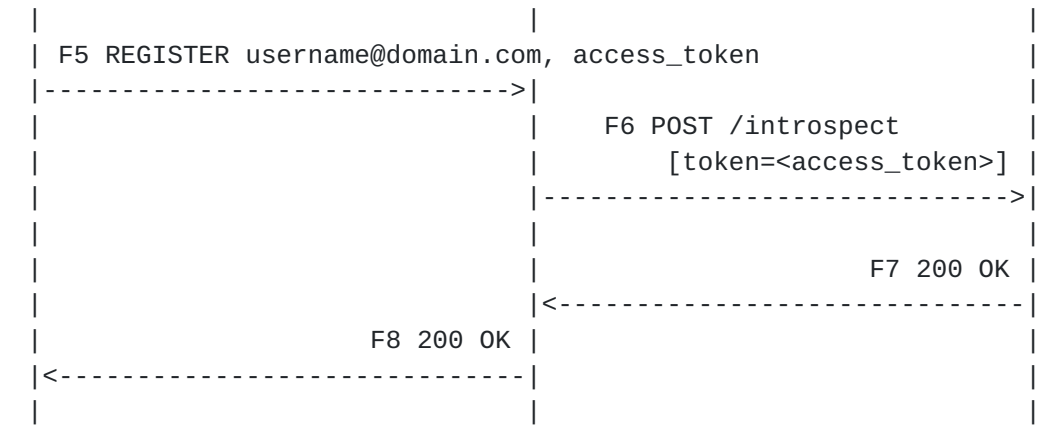
4.1. OAuth Implicit Grant

4.1.1. Overview

The following figure provides a high level view of flow of messages for the OAuth Implicit Grant type:



Registration



4.1.2. Authentication

The UA starts the process by sending an HTTP GET request to the Authorization Server without providing any credentials in the first request (F1).

The UA constructs the initial request (F1) to the Authorization Server with the following URI parameters in the query component:

response_type (REQUIRED)

Value MUST be set to "token".

user_id (REQUIRED)

The user's identification with the Authorization Server.

scope (OPTIONAL)

The scope of the access request.

The Authorization Server challenges the request using the Digest scheme (F2). The client retries the request (F3) and provides the user's credentials. In response the Authorization Server responds with 200 OK (F4) with the Access Token in the body.

4.1.3. Registration

The UA starts the registration process with the SIP proxy by sending a REGISTER request (F5) with the access token it obtained in the previous steps (F1-F4).

The UA adds the following parameters to the body of the REGISTER request:

access_token (REQUIRED)

The access token issued by the authorization server.

token_type (REQUIRED)

The type of the token issued by the authorization server. Value is case insensitive.

expires (RECOMMENDED)

The lifetime in seconds of the access token.

scope (OPTIONAL)

The scope of the access request.

If introspection is used [RFC7662], then the proxy validates the access token by sending an HTTP POST request (F6), with the parameters sent as "application/x-www-form-urlencoded" data, to the Authorization Server and include the following parameters:

token (REQUIRED)

The string value of the token.

token_type_hint (OPTIONAL)

A hint about the type of the token submitted for introspection.

Authorization Server then validates the request and responds with 200 OK (F7), with a JSON object in the body with the following parameters:

active (REQUIRED)

Boolean indicator of whether or not the presented token is currently active.

scope (OPTIONAL)

The scope of the access request.

Other parameters

TBD.

4.1.4. Subsequent Requests

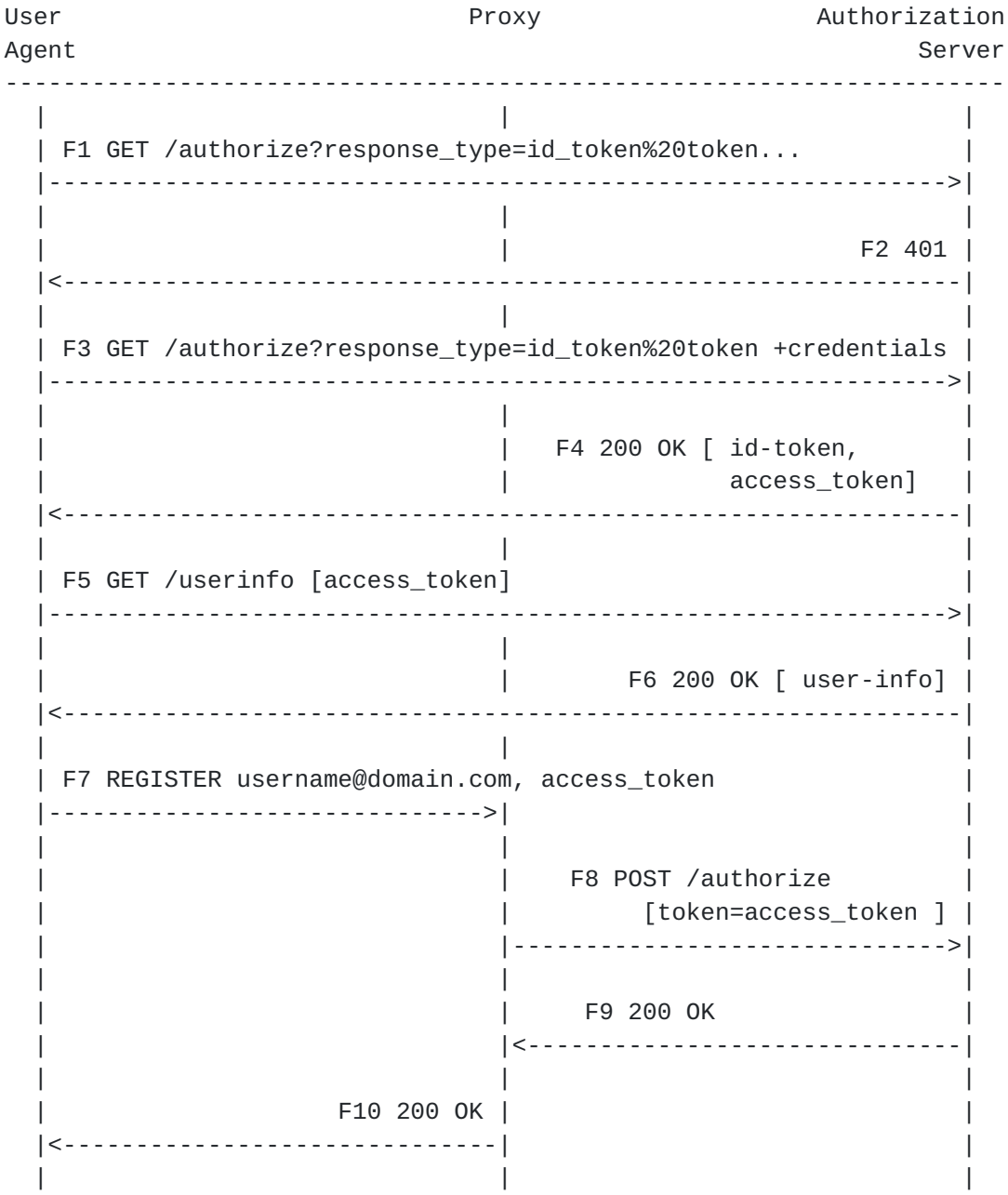
All subsequent requests from the UA MUST include a valid access token. The UA MUST obtain a new access token before the access token expiry period to continue to get service from the system.

4.1.5. Services

When the proxy forwards a request from a UA to an application server, it makes sure to keep the access token and scope in the message to allow the application server to provide the proper service to the user.

4.2. OpenID Implicit Grant

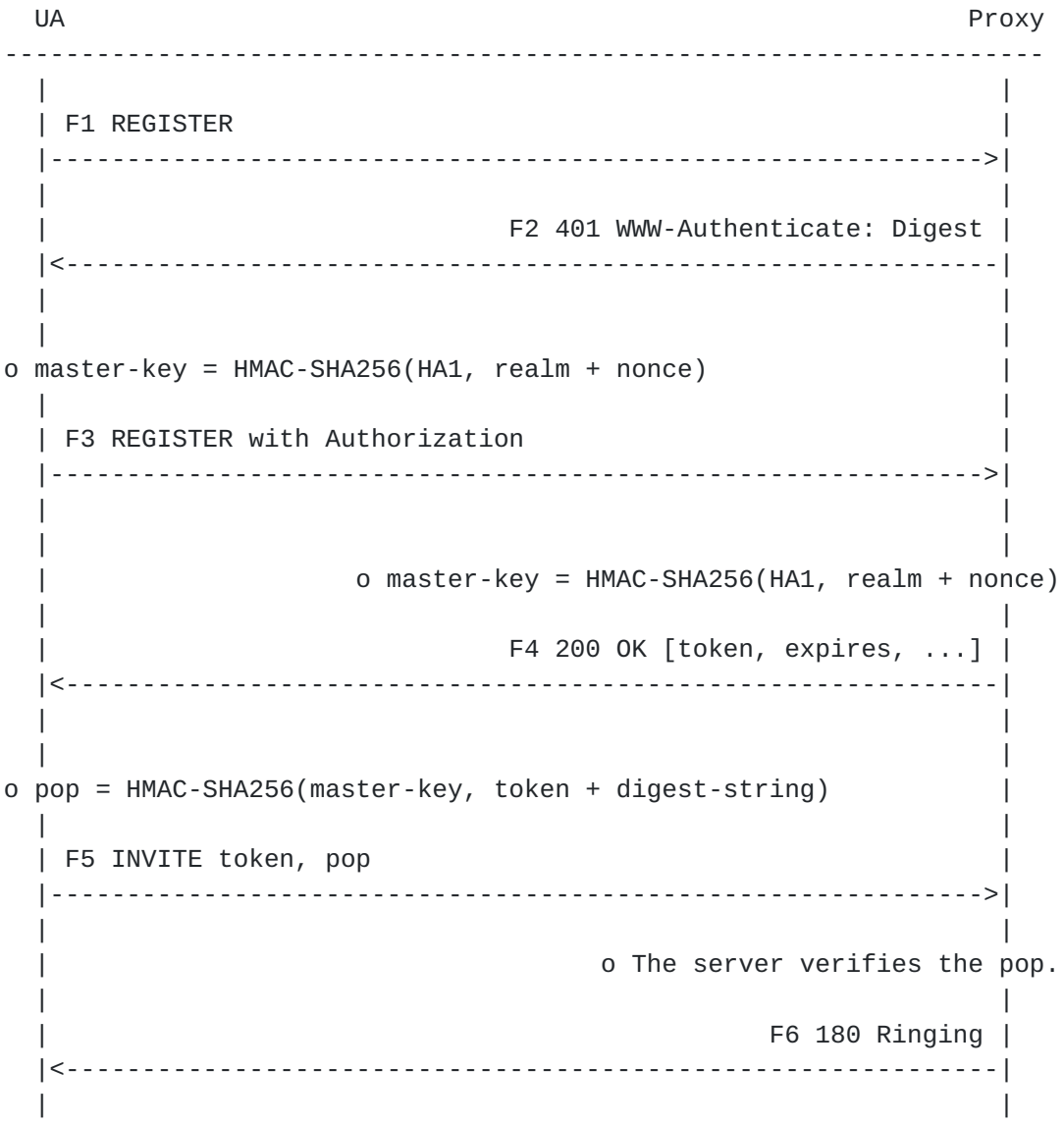
The following figure provides a high level view of flow of messages for the OpenID Implicit Grant type:



5. Resource Owner Password Credentials Grant type

5.1. Operations Overview

The following figure provides a high level view of flow of messages for the Resource Owner Password Credentials Grant type:



During registration the UA initially sends a REGISTER request (F1) without providing any credentials.

The proxy then challenges the UA by responding with 401 (F2) that includes the Digest scheme in the www-authenticate header.

The UA will generate a master-key that is based on an HMAC-Hash algorithm, e.g. HMAC-SHA256, that takes an input the user's HA1 and the concatenation of realm and nonce received in the challenge from the server. The UA will continue to use the existing operation of handling the Digest challenge and then sends a new REGISTER request (F3) with the credentials to the server.

When the server receives the request with the credentials (F3), the server will verify the digest provided by the UA; if that is successful, the server will accept the registration (F4) and include the details of the token in the response.

The server then generates a master-key that is based on an HMAC-Hash algorithm, e.g. HMAC-SHA256, that takes an input the user's HA1, and the concatenation of realm and nonce sent in the challenge to the client.

At the end of the above process the UA would have registered with the proxy and both the UA and the proxy would have created the same master-key without sending the master-key on the wire.

Later when the UA wants to send a request to the proxy it MUST always include the token and SHOULD include the pop as defined in section 4.6.

5.2. Registration and Acquiring Tokens

The UA MUST request the access token during the registration process with the proxy, by including a body with the grant_type as "password". Initially, the UA sends a REGISTER request without providing any credentials.

The proxy MUST then challenge the UA by responding with 401 with the Digest scheme in the WWW-Authenticate header.

When the UA gets challenged by the proxy to provide its credentials, the UA MUST include its credentials in the new REGISTER request in the authorization header as it is done with the existing mechanism, and MUST include a body with the grant_type as "password".

In addition, the UA MUST generate a master-key as follows:

```
master-key = HMAC-SHA256(HA1, realm + nonce)
```

Where

o HA1 - this is the user's H(A1) as defined in [DIGEST].

- realm - this is the realm that is returned by the server in the response to the initial request from the UA.

- nonce - this is the nonce that is returned by the server in the response to the initial request from the UA.

When the server receives the request with the credentials, the server will verify the digest provided by the UA; if that is successful, the server will accept the registration and include the details of the token in the response.

[[OPEN ISSUE]] How should the tokens be transported to the UA? in the body of the 200 OK? or a SIP header?

The server then generates a master-key following the same procedure followed by the client.

As a result of this procedure both the UA and the server would have created the same master-key without sending the master-key on the wire.

5.3. Discarding Credentials

After successfully receiving the access and refresh tokens from the proxy, the UA SHOULD discard the user credentials.

5.4. Token Refresh

The UA makes a refresh request to the token by sending a refresh REGISTER request that includes the authorization header and a body with the grant_type, the refresh_token, and the proof-of-possession of the master-key.

For example:

```
grant_type=refresh_token&refresh_token=<refresh_token>&pop=<pop>
```

5.5. Authenticated Requests

When the UA wants to send any request to the proxy, it MUST include the Authorization header and use the Bearer scheme to carry the access token, and the proof-of-possession of the master-key.

For example:

```
Authorization: Bearer token=<token>, pop=<pop>
```

See rfc4474, section 9, for the SIP headers to hash to create the value for the proof.

[[OPEN ISSUE]] The Bearer scheme is used to deliver tokens without providing any proof of possession. We probably need to use different scheme later on.

5.6. Examples

```
REGISTER sip:registrar.biloxi.com SIP/2.0
Via: SIP/2.0/TCP bobspc.biloxi.com:5060;branch=z9hG4bKnashds7
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Bob <sip:bob@biloxi.com>;tag=456248
Call-ID: 843817637684230@998sdasdh09
CSeq: 1826 REGISTER
Contact: <sip:bob@192.0.2.4>
Expires: 7200
Content-Length: 19

grant_type=password&pop=<pop>
```

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP bobspc.biloxi.com:5060;branch=z9hG4bKnashds7
     ;received=192.0.2.4
To: Bob <sip:bob@biloxi.com>;tag=2493k59kd
From: Bob <sip:bob@biloxi.com>;tag=456248
Call-ID: 843817637684230@998sdasdh09
CSeq: 1826 REGISTER
Contact: <sip:bob@192.0.2.4>
Expires: 7200
Content-Length: 0

{
   "access_token":"2YotnFZFEjr1zCsicMWpAA",
   "token_type":"example",
   "expires_in":3600,
   "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
   "example_parameter":"example_value"
}
```

6. Outbound

RFC5626 defines a mechanism that allows a UA to simultaneously connect and establish registration with multiple outbound proxies to get service.

This section describes that impact of outbound on this authorization mechanism.

6.1. Authorization Code Grant type

During initial registration with the primary proxy, the UA is able to get an authorization code that it will use to register with the primary proxy. Assuming the authorization server is shared between the various outbound proxies, the UA will be able to use the same authorization code to register with the secondary proxies and as a result each one of the secondary proxies will get the master-key associated with the user to be used for the calculation of the proof-of-possession.

6.2. Resource Owner Password Credentials Grant type

During registration the proxy challenges the UA, and both the proxy and the UA create a master-key based on HA1, realm, and nonce. Since the nonce is not shared between the various proxies, it is not possible for the outbound proxies to use the same master-key; as a result, the UA is expected to maintain a master-key and token per outbound proxy.

7. Security Considerations

<Security considerations text>

8. IANA Considerations

<IANA considerations text>

9. Acknowledgments

<Acknowledgments text>

10. Normative References

[OPENID] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", February 2014.

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

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, H., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

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

[RFC7662] Richer, J., "OAuth 2.0 Token Introspection", RFC 7662, October 2015.

Authors' Addresses

Rifaat Shekh-Yusef (editor)
Avaya
250 Sidney Street
Belleville, Ontario
Canada

Phone: +1-613-967-5267
EMail: rifaat.ietf@gmail.com

Victor Pascual
Oracle
Spain

EMail: victor.pascual.avila@oracle.com

Christer Holmberg
Ericsson
Hirsalantie 11
Jorvas 02420
Finland

EMail: christer.holmberg@ericsson.com