The OAuth Security Model for Delegated Authorization
draft-barnes-oauth-model-01

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the
provisions of BCP 78 and BCP 79.

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

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

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

The list of Internet-Draft Shadow Directories can be accessed at

This Internet-Draft will expire on January 9, 2010.

Copyright Notice

Copyright (c) 2009 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 in effect on the date of
publication of this document (http://trustee.ietf.org/license-info).
Please review these documents carefully, as they describe your rights
and restrictions with respect to this document.

Abstract

This document describes the security model for the OAuth
authorization system, which allows a party that holds some
authorization to delegate a subset of that authorization to another
party, without requiring either party to disclose its credentials to
the other. In this document, we describe a set of design
constraints, a high-level work flow for establishing authorizations
subject to those constraints, and set of security requirements for
protocols that implement this model.

Table of Contents

1. Introduction ................................................. 3
2. Definitions .................................................. 4
3. Requirements and design goals ................................. 5
  3.1. Trust assumptions ......................................... 6
  3.2. Threat model ............................................... 6
4. Authorization flow ............................................ 7
  4.1. General Requirements and Assumptions ..................... 8
  4.2. Request token issuance .................................... 9
  4.3. Authorization of the request ............................... 10
    4.3.1. Request for authorization ............................ 10
    4.3.2. Verification token issuance ........................... 11
    4.3.3. Notification of authorization .......................... 12
  4.4. Access token issuance ..................................... 13
  4.5. Accessing protected resources ............................. 14
  4.6. Summary ................................................ 14
5. Security analysis .............................................. 17
  5.1. Protection from outside interference ....................... 17
  5.2. Protection from malicious actors within the model ........ 18
  5.3. Protection from colluding actors .......................... 19
  5.4. Protection from denial of service ........................ 20
6. Acknowledgements .............................................. 20
7. IANA Considerations .......................................... 20
8. Security Considerations ....................................... 21
9. Normative References ......................................... 21
Authors’ Addresses ............................................. 21
1. Introduction

It is an almost universal situation in the Internet today that users make use of many Internet applications, some of which store data for the user and some of which consume user data. There is increasing interest in bringing these two types of application together, i.e., in allowing certain applications that consume user data to access data that is stored by other applications.

However, in order to maintain the security of user data, applications that store data typically require an entity requesting private resources to authenticate that they own those resources (i.e., to authenticate as the owner of the resources). Enabling a second application to access private resources would thus require the user who owns the resources to enable that application to authenticate as the user. This situation is clearly undesirable, since it allows the consuming application to impersonate the user: To access any and all data the user owns, or to take actions as the user.

This situation has created a need for a mechanism for delegating authorization. In such a system, the owner of a set of resources can delegate access to some of those resources to a designated client application, without enabling the client application to impersonate the user.

There are two basic prerequisites of such a delegation mechanism: First, applications that store user data must have separate notions of ownership and access permissions; this, of course, is not subject to standardization. Second, there must be a way for client applications to request access, and for users to instruct servers to grant such access. This second aspect is the subject of this document.

This document describes a security model for the OAuth delegated authorization system. In particular, we provide an abstract message flow that defines how parameters are exchanged between OAuth entities, and what the security requirements are for these exchanges. The definition of protocol mechanisms for conveying parameters in compliance with these requirements is left to future documents.

[[ Editor’s note: This document is essentially an abstraction and security analysis of the OAuth Core protocol, defined in [1]. While the current OAuth protocol should map more or less directly to this model, there may be slight difference, e.g., in where authentication is required. ]]}
2. Definitions

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 RFC 2119 [2].

The focus of this document is the secure delegation of access to resources. In this document, we use the term "resources" broadly, to include actual data (e.g., files, streams) as well as permissions to take particular actions.

There are three roles involved in the OAuth process. These roles are not mutually exclusive; a single party may act in more than one role.

Resource Owner (RO): The entity that has authorization at the beginning of the OAuth process, who delegates authorization to the Client

Client: The entity to which authorization is delegated by the Resource Owner

Server: The entity that enforces the authorizations in question (e.g., the server that stores a protected resource)

[[ Editor’s note: These three roles correspond to the User, the Consumer, and the Service Provider in OAuth 1.0, available at http://oauth.net/ ]]

An OAuth Server issues three types of opaque tokens in the course of the authorization process. These values are random values that are unique within a defined scope.

Request token: A temporary identifier for a specific Client for the duration of a single OAuth authorization transaction. An RO that requests a verification token using that request token indicates that the corresponding Client is the entity to be authorized.

Verification token: A temporary identifier for an authorization, including the authorized Client, the authorizing RO, and the resources to which access is to be granted. A Client that requests an access token using a given verification token indicates that the RO has granted the corresponding authorization.

Access token: A long-lived identifier for an authorization. A Client that requests access to resources using a given access token indicates that it has been authorized via the OAuth process to access those resources.
Many of the security properties of the OAuth process require protocol participants to associate, or "bind", certain other data to tokens, then subsequently verify these associations/bindings. In the below, when an entity is required to use the value "bound to a token" or "associated with a token", it is implied that the value in question is the one associated by the entity at hand.

3. Requirements and design goals

The fundamental goal of the OAuth model is to allow the Resource Owner to grant the Client authorization to access selected resources on the Server, while maintaining strict authentication and access control. The requirement for strict authentication means that no party should be able to authenticate as any other, whether a protocol participant (e.g., the Client authenticating as the RO) or a third party. The requirement for access control means that the delegation process should not grant the Client (or any other party) access to anything other than the resources designated for access by the Resource Owner.

The OAuth model must also ensure that the authorization process is followed: In order for an authorization to be delegated by a Resource Owner to a Client, it must be requested by the Client and authorized by the Resource Owner. In particular, the RO cannot unilaterally grant authorization to a Client without a request (since the Client may not be able to use that authorization), and of course, no authorization can be granted without the approval of the RO. In addition, the security model should prevent the Client from further delegating access without going through the same OAuth process.

It should be clear that these requirements already rule out the simplest models for delegating authorization. For example, a model in which the RO simply provides its authentication credentials to the Client is unacceptable for several reasons: It allows the Client to access all resources (not only any selected subset), and it requires the Client to authenticate as the RO.

Note also that this document defines only the security model for OAuth, i.e., how security-relevant data flow through the system, and what the security requirements are for these flows. In particular, this document does not define protocol mechanisms for carrying data (e.g., tokens) or providing security features (e.g., authentication, confidentiality). Such mechanisms should be defined in protocol-specific documents (e.g., a realization of OAuth in HTTP or XMPP).
3.1. Trust assumptions

In the OAuth architecture, the Server is a trusted entity, in the sense of RFC 4949 [3]: Since the Server is clearly capable of providing anyone access to the protected resources, no further analysis of the risks it presents is necessary; we assume it behaves as specified.

The Resource Owner for a set of protected resources is also trusted in the sense that it can authorize any request from a Client; we assume that genuine ROs will only tell the server to authorize requests from Clients that it genuinely intends to authorize. We do not assume that any entity acting as an RO is the genuine RO for the protected resources. It will be necessary for the OAuth system to provide the Client and Server assurance that the RO in the process is the genuine owner of the resources in question. (The Server can verify this directly, with standard authentication mechanisms; the Client will rely on an authenticated assertion by the Server.)

All parties are assumed to maintain the security of their private authentication keys (the long-lived credentials that enable to authenticate to other protocol entities). Beyond that minimal assumption, we make no security assumptions about the behavior of the Client.

3.2. Threat model

The authorization system here is designed to protect against four classes of threats:

1. Outside intervention in an otherwise honest transaction
2. Malicious actors within the authorization process
3. Collusion among actors within the authorization process
4. Denial of service against the authorization process

The basic goal of the authorization system is to allow an honest Client, Server, and RO to delegate authorization even in the presence of external bad actors. This goal is mainly accomplished by requiring authentication at critical points of the protocol to assure that the parties are always talking among themselves, and not to some outside actor.

OAuth is also intended to prevent the abuse of the authorization process by legitimate participants in it, especially Clients. This goal is addressed by a set of mechanisms that allow the Server and
the Resource Owner to validate that the authorization process is conducted faithfully.

A more advanced class of attack is when two or more participants collude in order to gain false access. The primary example of this sort of attack is when two Clients collude so that the one more trusted by the RO can obtain access on behalf of the other. This class of attack is addressed by assuring that the Client is consistently identified throughout the OAuth process.

Finally, there are several points in the OAuth process where the injection of false information would cause the process to fail, even in the absence of other problems. This threat can be minimized by ensuring that only authentic protocol participants can contribute information to the process.

4. Authorization flow

The OAuth authorization process is the process by which a Resource Owner can grant a Client permission to access resources protected by a Server. This process occurs in four high-level steps, illustrated in Figure 1:

1. The Client obtains a Request Token from the Server
2. The Client requests authorization from the Resource Owner
3. The Resource Owner validates the request and grants authorization
4. The Client exchanges the Request Token for an Access Token

If the process fails at any point, no authorization is granted.
### 4.1. General Requirements and Assumptions

There are a number of authentication relationships that need to be in place before OAuth can be used to delegate authorization. Client authentication is critical for preventing unauthorized parties from gaining access: The Client MUST be able to authenticate to both the Server and the Resource Owner. Moreover, the Client MUST be able to authenticate the same identifier to both parties (e.g., using the same certificate in the context of HTTPS), so that the Server and the Resource Owner can verify that the Client that the RO authorizes is the same one that the Server will enable to access the protected resources.

Likewise the Resource Owner MUST be able to authenticate both to the Server and to the Client, and it MUST be able to authenticate the same identifier to both (it may have other identifiers with each). The Server MUST be able to authenticate both to the Client and the RO, but it is not required to use the same identity for both.

[[ Editor’s note: The requirements that the Client and RO use the same identifiers across multiple relationships can be onerous, but is required to combat the Client and RO collusion attacks discussed in Section 5.3, as are the identity checks by the RO and the Client that]
take advantage of these identifiers (in Section 4.3.3 and Section 4.4). If WG discussions indicate that either or both of these attacks are not to be addressed by this model, then the identity requirements and the corresponding checks can be removed; for now, they are listed as MUST-level requirements. ]]

All messages in this process SHOULD be integrity- and confidentiality-protected; ones that MUST be so protected are noted below. The security of the authorization process relies only on the authentication relationships above, but access to some parameters passed between OAuth entities could allow third parties to disrupt the OAuth process.

4.2. Request token issuance

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Resource Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>req-token-request</td>
<td></td>
</tr>
<tr>
<td>req-token-request</td>
<td>&lt;------------------------&gt;</td>
<td></td>
</tr>
<tr>
<td>req-token-issue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(req-t)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The OAuth process is initiated by a Client that wishes to access a given set of resources on the Server. The authorization flow begins when the Client sends a request for a Request Token to the Server. In response to this request, the Server MUST authenticate the Client. If authentication is unsuccessful, the process fails. If authentication succeeds, the Server MUST generate a random request token and store a binding between the request token and authenticated identity of the Client (denoted Cl-ID below). During this transaction, the Client and the Server MAY negotiate the resources to which access is to be granted.

Upon receiving a request token from the Server, the Client MUST authenticate the Server and store a binding between the Server’s authenticated identity and the request token.

The Request Token identifies a Client’s request for authorization to access resources on the Server. At issuance, the token is bound to a Client identity and a Server identity. To uniquely identify an authorization transaction, the Server MUST ensure that each Request Token is unique within the set of such tokens that it issues. Token values SHOULD be difficult for an outside entity to guess. In order to avoid resource exhaustion at the Server, it is RECOMMENDED that
Request Tokens have a limited lifetime.

4.3. Authorization of the request

Before the Client can access protected resources, it must be authorized to access those resources: The Client must request authorization, the Resource Owner must notify the Server that it intends to grant access, and then the Resource Owner must actually grant access to the Client. In addition, before the Resource Owner grants access, it must assure that the Client’s request is authentic by validating it with the Server.

4.3.1. Request for authorization

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Resource Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>request-authz</td>
<td>(req-t)</td>
</tr>
<tr>
<td>3.</td>
<td>------------------------&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The Client requests authorization to access resources by sending a message to the Resource Owner that contains an unused request token. After sending a request token in an authorization request, the Client MUST mark that request token as used, and MUST NOT use the request token in any subsequent authorization request.

If the Client wishes to verify that the RO that it interacts with on this step is the same entity that it interacts with later, then it MUST authenticate the RO and store a binding between the request token and the RO’s authenticated identity (in addition to the binding to the Server’s identity recorded earlier). It should be noted however, that these steps are purely optional. They provide no additional security guarantees with respect to the authorizations granted; they only allow the Client to track which request tokens have been used. (In the presence of session continuity mechanisms, such as HTTP cookies, this authentication may be used as the basis for the authentication required at the time of notification of authorization, described in Section 4.3.3.)

Note that this transaction does not require the RO to authenticate the Client. The Client will be authenticated in a later step, after the RO is told by the Server which Client identity is bound to the request token.
4.3.2. Verification token issuance

The Resource Owner validates the request for authorization and obtains the ability to grant authorization by requesting a verification token from the Server. This request MUST include the request token provided by the Client. Before sending this request, the RO SHOULD authenticate the Server, since a false Server can disrupt the authorization process (but not cause unintended authorization to be granted).

When the Server receives a request for a verification token, it MUST take the following actions, in order:

1. Authenticate the Resource Owner and verify that the RO owns the resources subject to authorization

2. Verify that the request token has not been used in a prior request. If so, the process fails; if not, the request token is marked as used.

3. Generate a random verification token that is unique in the scope of such tokens

4. Store a binding between the verification token, the Client identity bound to the request token, and the RO’s authenticated identity

5. Establish an integrity- and confidentiality-protected channel to the RO (if one does not already exist, e.g., the channel used the request)

6. Send a message to the RO containing the following information:

   * The verification token
* The Client identity bound to the request token

* The scope of the authorization to be provided (i.e., the set of resources)

The Server and the RO MAY also perform further negotiation of the scope of the authorization at this point.

It should be emphasized that the Server MUST NOT transmit the verification token over any channel that is not authenticated, integrity-protected, and confidentiality-protected, since the verification token is highly sensitive information: Any third party that intercepts the verification token is capable of granting the Client access to the protected resources, and any party that can inject or modify a verification token can cause the RO to grant unintended accesses.

Upon receiving the Server’s message (authenticating the Server again, if necessary), the RO stores a binding between the verification token and the Client identity provided by the Server.

Some might note that there is an asymmetry to the security requirements for the two halves of this transaction: The request for a verification token is only authenticated, while the response also needs to be confidentiality- and integrity-protected. This is because the injection of a false request token can only cause the process to fail (since the Client identity provided by the server will be wrong), while leakage of the verification token can cause used to grant an unintended authorization. For simplicity, it is RECOMMENDED that the same channel (with full protections) be used for the request and the response.

Note that at this point in the process, no authorization has been granted. The Server has only issued the Resource Owner a verification token so that the RO can grant authorization if desired. The Server MUST NOT provide the Client access to protected resources until the RO has provided the verification token to the Client, and Client has exchanged the verification token for an access token, as described below.

### 4.3.3. Notification of authorization

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Resource Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>authz-notify</td>
<td>(ver-t,req-t)</td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td>&lt;-----------------------------------------------</td>
</tr>
</tbody>
</table>

Barnes & Lepinski Expires January 9, 2010 [Page 12]
If the Resource Owner decides to authorize the Client to access protected resources (the set bound to the verification token), then it signals this by sending a message to the Client containing the verification token. Before sending such a message, the RO MUST authenticate the Client and establish an integrity- and confidentiality-protected (i.e., encrypted) channel to the authenticated Client. If the authenticated identity of the Client does not match the Client identity bound to the verification token, or if the authentication fails, then the process fails and the RO MUST NOT send the verification token to the Client. As above, the RO MUST NOT transmit the verification token over any channel that is not authenticated, integrity-protected, and confidentiality-protected.

Upon receipt of an authorization notification message, the Client MUST authenticate the RO and store a binding between the RO’s authenticated identity and the verification token. The Client MUST also store a binding between the verification token and the Server identity bound to the request token. The Client MAY verify that the RO’s authenticated identity matches the one bound earlier to the request token.

4.4. Access token issuance

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Resource Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc-token-request (ver-t)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. --------------&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acc-token-issue (acc-t,ro-id)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. &lt;--------------</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Client obtains access to the designated resources by exchanging a verification token that it has received for an access token. To request an access token, the Client sends the verification token to the Server. Before sending this message, the Client MUST authenticate the Server and verify that the authenticated Server identity is the one bound to the verification token (i.e., the Server identity that the Client originally bound to the request token).

Upon receiving a request for an access token, the Server MUST authenticate the Client and verify (1) that the verification token provided is one that it issued, and (2) that the Client’s authenticated identity matches the identity bound to the verification token. If both criteria are satisfied, then the Server MUST generate
a random access token (unique within the scope of such tokens) and store a binding between the access token, the Client’s authenticated identity, the RO identity bound to the verification token, and the resources bound to the verification token.

The Server completes the authorization process by sending to the Client a message containing (1) the access token and (2) an identifier for the RO to which the verification token was issued. The Client MUST authenticate the Server prior to accepting such a message and verify that the Server’s identity matches that bound to the request token and verification token. The Client MUST also verify that the RO identity returned by the Server matches the RO identity bound to the verification token. If both verifications succeed, then the Client stores a binding between the access token and the protected resources.

Note that because the Client received the verification token over a confidentiality-protected channel, the Server’s acceptance of the verification token provides the Client assurance that the entity that provided the verification token was authorized (at the Server) to grant the specified authorization.

4.5. Accessing protected resources

```
<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Resource Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc-res-request</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(acc-t)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;----------------------------&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;----------------------------&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Finally, once a Client is authorized to access resources on the Server, it accesses them using the corresponding access token. When a Client wishes to access resources, it sends a request to the Server containing the access token. The Server MUST authenticate the Client and verify that the access token presented belongs to the authenticated Client before responding with the requested resources.

4.6. Summary

All of the interactions above are summarized in Figure 4, below. Figure 2 and Figure 3 illustrate the life-cycles of the request token and the verification token, respectively.
1. Issues token +--------+ 2. Uses token
4. Provides info | Server |--------------| Client | request authz
to validate +--------+              +--------+
            ^                      |
            |     +----------+     |
            | +-----| Resource |<----+
            | +------>|   Owner  |
            v +----------+
3. Uses token to request
   verification token

Figure 2: Life cycle of a request token

1. Issues token +--------+ 2. Uses token
4. Provides info | Server |<---------------| Client | request access tok.
to validate +--------------------------------->+--------+
            ^                      |
            |     +----------+     |
            | +---->| Resource |-----+
            | +---->|   Owner  |
            v +----------+
3. Uses token to
   grant authz

Figure 3: Life cycle of a verification token
<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Resource Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>req-token-request</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>req-token-issue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(req-t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>request-authz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(req-t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ver-token-request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(req-t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ver-token-issue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ver-t,cl-id,scope)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>authz-notify</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ver-t,req-t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>acc-token-request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ver-t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>acc-token-issue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(acc-t,ro-id)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>resources</td>
<td></td>
</tr>
<tr>
<td></td>
<td>acc-res-request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(acc-t)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: OAuth message flow
5. Security analysis

As described in the design goals above, the OAuth model aims to protect against four general classes of threat:

1. Outside intervention in an otherwise honest transaction
2. Malicious actors within the authorization process
3. Collusion among actors within the authorization process
4. Denial of service against the authorization process

In this section, we discuss how the model described above addresses each class of threat.

5.1. Protection from outside interference

The authentication requirements of the OAuth process prevent outside entities from interfering with the process. This property means that if each party assumes that the other authenticated parties behave as the process specifies, all three parties are assured that the genuine RO has authorized the Client to access the specified resources on the Server, i.e., that no outside actors have corrupted the process. (The next section discusses how the parties can be sure that misbehaving actors cannot create unintended authorizations.)

The Server’s assurance comes from its knowledge of the token bindings and its confidence in the authenticated identities of the Client and the RO. The Server is assured of the Client’s identity because the Client authenticated that identity when it requested the request token and when it exchanged the verification token for the access token. The Server is assured that the genuine RO authorized the Client because it received the verification token from the Client, and because it knows that it only issued the verification token to the genuine RO (and because it trusts that the RO only gives the verification token to parties it intends to authorize).

The RO is assured that it knows the real identity of the authorized Client because it has verified that the identity that the Server will provide access to (provided by the server with the verification token) is the same as the authenticated identity of the Client that the RO gave the verification token to. The RO is also assured that no other Client has received the verification token because it transmitted that token only over an authenticated, confidentiality-protected channel. (Of course, the RO knows that it is the genuine owner of the subject resources.)
The Client is assured that the Server from which it will receive resources is the same as the Server it requested authorization from by the persistence of the Server’s identity between the requests for the request token and the access token (and subsequent requests for resources). The Client knows that the RO is the genuine RO because it has verified that the authenticated identity of the RO matches the genuine RO identity provided by the Server. The Client knows that it is the intended recipient of the authorization because the Server accepts the verification token in a request authenticated with the Client’s identity (i.e., because the Server only accepts verification tokens submitted with the identity bound to the verification token).

5.2. Protection from malicious actors within the model

The OAuth model ensures that the process fails (i.e., no authorization is granted) if any actor in the process acts improperly. Here we consider the avenues for misbehavior available to each role, and discuss how the model prevents them from having ill effects. Of course, since the Server is trusted, it can subvert the process by granting access to any party, so we do not consider its behavior in any further detail.

The RO performs two actions in the protocol that it might attempt to exploit to create unauthorized accesses: First it sends request tokens to the server to obtain verification tokens, and second, it sends verification tokens to the Client to grant access. Because the Server verifies that the RO owns the subject resources before it issues a verification token, a RO cannot obtain a verification token for resources it does not own. Moreover, once an RO has obtained a verification token for resources, that token can only be used by the Client that requested the request token; the RO cannot use it to grant access to any other Client. So a malicious RO cannot use the OAuth process to grant access beyond its ownership, or beyond the intended Client.

In order to obtain access to protected resources, a Client needs to obtain a verification token bound to its identity and the protected resources. However, verification tokens are only issued to the genuine RO, and only bound to the Client identity that the Server associates with a request token. So the only way for a Client to obtain access without violating the authentication constraints of the model (i.e., forging an authentication or obtaining another Client’s private authentication key) is to obtain a request token with its own Client identity, and convince the genuine RO to authorize this request token -- namely, to go through the proper OAuth procedure. So the only risk from misbehaving Clients is from social engineering attacks against the RO (e.g., phishing).
5.3. Protection from colluding actors

It is possible for multiple actors within the OAuth process to collude in order to subvert the process. There are six possible combinations of roles, including multiple actors in the same role:

1. Server -- Server
2. Server -- Client
3. Server -- RO
4. Client -- RO
5. Client -- Client
6. RO -- RO

Of these, we need only consider the latter three, since collusion with the Server is irrelevant (the Server can simply provide the desired accesses or resources).

Since a legitimate Client and RO can already allow the Client access to the RO’s resources, the only remaining objective for such a collusion would be to convince the Server that access was really being granted to another Client. This attack, however, is prevented by the requirement that the Client authenticate all its interactions with the Server: The RO can only grant access to a Client identity bound to a request token, and that binding is established by a transaction in which the Client’s identity is authenticated.

Two Clients cannot by themselves obtain any access, so two colluding, unauthorized Clients do no damage (neither can present a request token that will result in an authorization). However, if one Client is authorized (say C1), the other (C2) might try to use it to gain access, in the following way: C2 could register a token for itself, pass that token to C1, and have C1 request authorization. If the RO fails to verify that the Client identity bound to the request token (provided by the Server) matches the identity of the requesting Client, then the RO will return the verification token to C1. C1 can then pass the verification token to C2, who can then obtain an access token and access resources. On the other hand, if the Client does perform the specified verification that the two identities match, then the risk that this collusion will result in unintended access is removed.

In a similar vein, the major risk from colluding ROs is that they convince the Client that any given colluding RO is the proper owner
of resources than any other colluding RO owns. (It is a given that they can grant access to their own resources, and there are no attacks available against the Server, since it has authoritative knowledge.) To accomplish this attack, when the Client requests authorization from one RO (say RO1), that RO passes the request token to another RO (RO2). RO2 then acquires a verification token and passes it back to RO1, who uses it to notify the Client. If the Client fails to compare RO1’s authenticated identity to the identity bound to the verification token (provided by the Server), then the Client will think that RO1 is the owner of resources that actually belong to RO2. On the other hand, if the client does perform this check (as required), then this risk is eliminated.

5.4. Protection from denial of service

The successful completion of the OAuth process is dependent on using tokens to communicate bindings between identities and authorizations. Entities that can inject false tokens of any type can therefore cause the process to fail (false in the sense that they are not bound to any identity at the server). The model is secured against this type of denial of service attack because senders of information are authenticated where necessary, and because messages are integrity-protected to prevent tampering en route.

The OAuth process allows the Client and the RO to force the Server to maintain state, in the form of bindings between request and verification tokens and other information. The requirement for the server to maintain this state information makes it vulnerable to resource exhaustion attacks. The Server can mitigate this risk by imposing a limited life-time on request tokens and verification tokens, at the risk that an OAuth process that is not completed within the life-time of the appropriate tokens will fail.

6. Acknowledgements

This document is essentially an abstraction and security analysis of the current OAuth protocol [1]. We are thus deeply in debt to the authors of that protocol, especially Eran Hammer-Lahav, who provided several helpful explanations of the current OAuth protocol specification as this document was being developed.

7. IANA Considerations

This memo makes no request to IANA.
8. Security Considerations

This document describes a security model for delegated authorization, especially in the World Wide Web context. As such, security considerations are discussed in detail throughout.

9. Normative References


Authors’ Addresses

Richard Barnes
BBN Technologies
9861 Broken Land Parkway
Columbia, MD  21046
US

Phone: +1 410 290 6169
Email: rbarnes@bbn.com

Matthew Lepinski
BBN Technologies
10 Moulton St
Cambridge, MA  02138
US

Phone: +1 617 873 5939
Email: mlepinski@bbn.com