OAuthHub — A Service for Consolidating Authentication Services

Xuzong Chen*, Gareth Sime*, Christof Lutteroth†, Gerald Weber†
*Department of Electrical and Computer Engineering, †Department of Computer Science
The University of Auckland
Auckland, New Zealand
{xche985,gsim760}@aucklanduni.ac.nz, {christof,gerald}@cs.auckland.ac.nz

***Abstract*—OAuth has become a widespread authorization protocol to allow inter-enterprise sharing of user preferences and data: a *Consumer* that wants access to a user's protected resources held by a *Service Provider* can use OAuth to ask for the user's authorization for access to these resources. However, it can be tedious for a Consumer to use OAuth as a way to organise user identities, since doing so requires supporting all Service Providers that the Consumer would recognise as users' "identity providers". Each Service Provider added requires extra work; at the very least, registration at that Service Provider. Different Service Providers may differ slightly in the API they offer, their authentication/authorisation process or even their supported version of OAuth. The use of different OAuth Service Providers also creates privacy, security and integration problems. Therefore OAuth is an ideal candidate for Software as a Service, while posing interesting challenges at the same time.**

We use conceptual modelling to derive new high-level models and provide an analysis of the solution space. We address the aforementioned problems by introducing a trusted intermediary – OAuthHub – into this relationship and contrast it with a variant, OAuthProxy. Instead of having to support and control different OAuth providers, Consumers can use OAuthHub as a single trusted intermediary to take care of managing and controlling how authentication is done and what data is shared. OAuthHub eases development and integration issues by providing a consolidated API for a range of services. We describe how a trusted intermediary such as OAuthHub can fit into the overall OAuth architecture and discuss how it can satisfy demands on security, reliability and usability.

I. Introduction

The digital native of today has a variety of online accounts. Each account has resources and data associated with it. Password fatigue is a known phenomenon with users and it extends to fatigue in providing personal data, such as addresses, repeatedly. Therefore many users allow a service (the Consumer) to access their protected resources on another service (the Service Provider). These protected resources are usually artefacts such as personal information, photos or communication mechanisms. In such a scenario, people's private data should be respected and kept safe. For example, if a user (Resource Owner) wanted their music service (Consumer) to be able to post updates on their social media page (Service Provider) about the music they currently listen to, then they may not want the music service to also access their photo albums.

The current industry standard for authorization (and authentication) between Consumers and Service Providers is OAuth [1]. However, although the OAuth protocol successfully standardizes the handshake between a Consumer and different Service Providers, the APIs that these Service Providers offer are inconsistent and also naturally heterogeneous. This means that developers of Consumers need to integrate with new APIs for every new Service Provider they want to support. Vice versa, new Service Providers have a difficult stand in gaining traction. The presence of such different APIs also makes it more difficult to make authentication and sharing of data secure for end users.

Currently there are many Consumers that allow their users to "log in" using their accounts with various Service Providers via OAuth. Examples of such Consumers include news sites such as *The New York Times*, education sites such as *Duolingo*, service/utility sites such as *BitBucket*, and media apps such as *Spotify*. Most of these web services very rarely access protected resources specific to certain OAuth Service Providers, such as "adding a new person into one of your Google+ Circles" or "checking your Facebook messages". They use OAuth mostly as a means to identify users, i.e. as an identity management and authentication mechanism, and most users seem happy to let Consumers make use of OAuth in this way. However, the development cost of interfacing and integrating with multiple OAuth Service Providers and the potential for problems in security (e.g. Consumers accessing data they should not need access to) are high.

In order to interface with any given OAuth server, the Consumer developer needs to study the specification of that server. In the best case in terms of interoperability, the Consumer developer needs to register their Consumer application at the server, and obtain a set of matching *Consumer key and secret* (a.k.a. "client credentials") [1]. In a worse case, the Consumer may need to handle different OAuth 2.0 "Authorization Grant" types, which couples the Consumer to particular servers. In contrast to version 1, OAuth 2.0 is not just simply a protocol but rather a *framework* [2] with a significant potential for server-specific complexity.

This work makes three contributions:

1) We provide a formal analysis of the OAuth protocol (focusing on OAuth 1.0), using techniques from model-driven architecture and conceptual modelling such as interaction diagrams.

2) We define a system, OAuthHub[1], that can provide easy access for Consumers to the benefits of OAuth, and contrast it with a variant called OAuthProxy.

The motivation for OAuthHub is to ease the burden on developers of Consumer applications with regard to OAuth authentication and resource sharing, with a focus on security. OAuthHub offers solutions that are simple and easily adoptable by Consumer developers, without involving protocols other than OAuth, so not to make a system more complex and vulnerable. For Consumer developers with OAuth expertise, the solution does not change its developers' workflow. However, it should help developers create Consumers that can use a variety of OAuth providers without additional security risks.

We cover some of the basics and contemporary issues of OAuth in Section III. Section IV outlines the proposed solution, Section V describes some of its properties and Section VI discusses some of its limitations. Section VII provides a comparison of OAuthHub with a variant, OAuthProxy. Section VIII concludes the paper.

II. Related Work

Managing user identities has long been an area of research. There are different categories of Single-Sign On (SSO) systems [3] that try to handle this task. OpenID, despite it perhaps not receiving as much attention in the developer community as OAuth, is supported by many large websites including Google, Yahoo etc. There were over 1 billion OpenID enabled accounts at the end of 2009 [4]. Bellamy-McIntyre, Lutteroth, and Weber [5] discussed how formal models could be used to identify the risks of using OpenID.

During recent years, as OAuth became a popular authorization protocol in the industry, usages of OAuth as an identity management tool appeared. OAuth.io [6], produced by WebShell [7], is an example of what we call an OAuthProxy. OAuth.io provides, as a whole, both Identity Provider consolidation and service-request (usually REST interfaces) consolidation, via OAuth.io's own servers and SDKs. This solution addresses the same problem that OAuthHub does in a somewhat similar manner. However, there are some significant differences between the two solutions that are discussed in detail in Section VII.

III. OAuth

OAuth is primarily designed for secure delegated access [1], which means Consumers can request that users grant them permission to use the users' protected resources on a Service Provider. Without secure delegated access, users would need to be asked to give away their user name and password to allow a Consumer to access resources on a Service Provider. If this was permitted at all, it would pose a significant security threat and meant surrendering full control over a user's account.

However, often OAuth is used by Consumers just as a Single Sign-On (SSO) protocol. The first motivation for Consumers to use SSO is to avoid managing their own credential system. There are numerous concerns that need to be addressed for a password system to be secure, e.g. allowed password length, password content, expiry time, handling consecutive failed sign-in attempts, password storage methods etc. [8].

Another motivation for Consumers to use SSO is to deal with users' password fatigue [9]. SSO allow users to reuse identities already established at Service Providers. SSO also helps when the user does not trust the Consumer application, e.g. for temporary logins at Internet cafés or public terminals [3].

Each Consumer is known to the Service Provider and can ask for a set of permissions when a user attempts to set up authorization between the two entities. This set of permissions is presented to the user so that they can make educated decisions on whether or not to approve this Consumer. Users can revoke the Consumer's permissions at any time.

The end result of setting up an OAuth connection between Consumer and Service Provider is that the Consumer will have token credentials which the Service Provider recognizes. Every time the Consumer tries to access the Service Provider in the future, the Consumer will provide the token credentials. An interaction diagram describing the OAuth dialogue is given in Figure 5, which will be described in more detail later on.

A. Terminology

In a typical OAuth authorization process, there are three parties involved. In different documents, they are referred to by different terms. In this article, we will mainly use the terms defined in the "community" version, *OAuth Core 1.0 Revision A* [10]. It is worth pointing out the correspondence between those terms and the terms in the RFC version, *RFC 5849: The OAuth 1.0 protocol* [1], because in many articles, the terms are used interchangeably. In studies related to user identity management, such as those regarding OpenID [11], there are also commonly used terms that refer to concepts analogous to the ones that OAuth is concerned with. The correspondences between the different terminologies are shown in Table I. In this article, we chose to use the "community" version terminology to avoid ambiguity due to the terms "client" and "server" being overloaded to mean different things (e.g. HTTP clients vs. OAuth clients).

TABLE I
Correspondences between common terminologies

RFC	Community	OpenID
Server	Service Provider	Identity Provider
Client	Consumer	Relying Party
Resource Owner	User	End User
Client credentials	Consumer key and secret	-
Token credentials	Access token and secret	-

B. OAuth 1.0 and 2.0

There are currently two OAuth standards that are recognized, the OAuth 1.0 protocol and the OAuth 2.0 framework. It

[1]http://github.com/oauthhub/

was intended that 2.0 would replace 1.0 [2] but there has been much controversy surrounding the topic [12]. There have been formal verifications of OAuth 2.0 [13], but in practice there are still serious security issues [14]. As such, there are many providers who do not support OAuth 2.0. Possible solutions would likely have to support both standards, as the market is so divided.

C. Different OAuth APIs

Each Service Provider can have a different API for accessing protected resources. This can make development of new Consumers difficult as code using the new API must be implemented for each Service Provider. For example, Table II shows the different resource locations required to retrieve a user's first name in popular OAuth Service Providers. Their implementations of OAuth are fairly heterogeneous, e.g. Facebook and Google+ use OAuth 2.0, whereas Twitter uses both 1.0 and 2.0.

IV. OAuthHub

To address the aforementioned problems with supporting different OAuth Service Providers, our proposed solution introduces a fourth party – a trusted intermediary – into the traditionally-three-partied OAuth (1.0) setup. The inspiration for this solution boils down to a well-known saying: "all problems in computer science can be solved by another level of indirection" [15]. Such an approach of introducing an intermediary is also seen in the design of other solutions, e.g. systems involving identity federation [16]. In particular, there indeed already exist services that act as OAuth intermediaries that make use of this approach; OAuthProxy, being one such example, is discussed below in Section VII.

In this section, we first give a description of our solution, called "OAuthHub". Then we briefly describe a few points of interest regarding OAuthHub, all of which are discussed in detail in Sections V (Properties) and VI (Limitations).

A. Solution Description

We will call a web service that meets the following description an "OAuthHub". Hence we provide here a specification for a Software as a Service (SaaS) system that may be implemented in different ways.

As seen in Figure 1, OAuthHub is registered beforehand with each Service Provider as an OAuth Consumer; later, each Consumer application is then registered with OAuthHub as an OAuth Consumer. The only party that plays more than one type of role is OAuthHub, which is an OAuth Consumer when interfacing with the Service Provider, and an OAuth Service Provider when interfacing with a Consumer. Figure 2 demonstrates this setup of two sets of 3-parties: one among the user, the Consumer, and the OAuthHub; another among the user, the OAuthHub and the Service Provider. OAuthHub has the elegant property that the two connections that are used between the Consumer and the Service Provider are both OAuth connections.

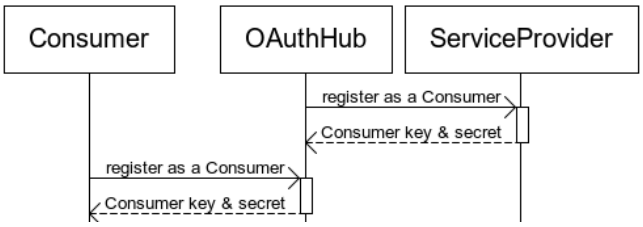

Fig. 1. Registering a new Consumer with OAuthHub

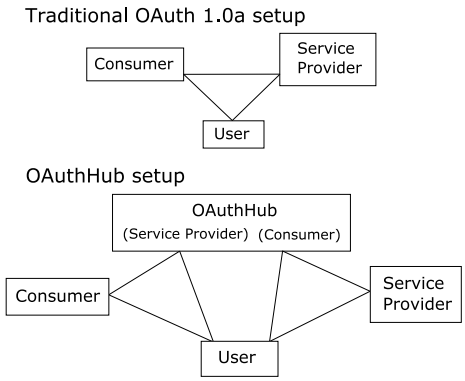

Fig. 2. Overview of the 4-party OAuthHub setup, compared to the traditional 3-party OAuth setup

The purpose of OAuthHub is to reduce development cost of Consumer applications. Instead of having to directly support and control different OAuth Service Providers, Consumers can use OAuthHub as a single trusted intermediary to take care of managing and controlling how authentication is done and what data is shared. Each new Consumer only has to integrate with OAuthHub to gain the benefits of integrating with many different Service Providers.

All requests from the Consumers for access to end-user resources held by OAuth Service Providers go through the OAuthHub. This concept is shown in Figure 3. A Consumer makes a request to OAuthHub, which subsequently makes a request to the Service Provider. This figure demonstrates OAuthHub performing a conversion between the unified API and the Service Provider's API.

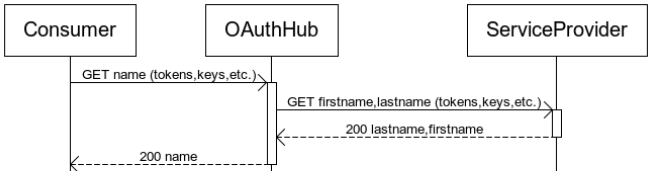

Fig. 3. A Consumer Making a Request Through OAuthHub

This is an interesting case study in service composition. We are investigating here the "recursiveness" of the OAuth protocol, a desirable property of protocol ceteris paribus. The result of our investigation is that this architecture is possible; it has clearly defined advantages and no fatal disadvantages.

TABLE II
RESOURCE LOCATIONS FOR RETRIEVING A USER'S FIRST NAME ON DIFFERENT SERVICE PROVIDERS

OAuth Service Provider	Resource Location	Field Name
Facebook	`https://graph.facebook.com/v2.3/{user-id}`	`first_name`
Twitter	`https://api.twitter.com/1.1/account/verify_credentials.json`	`name`
Google+	`https://www.googleapis.com/plus/v1/people/{user-id}`	`displayName`

B. Characteristics of OAuthHub regarding Consumer Keys

One advantage that will become clearer later is a clean separation of different sets of Consumer keys and secrets. This is in contrast to OAuthProxy, which requires Consumers to *hand over* their Consumer keys and secrets to OAuthProxy. As Figure 1 already demonstrated, each party has their own Consumer token and secret and there is no sharing of any credentials among any two parties. Initially the difference between OAuthHub and OAuthProxy seemed to have a substantial impact on privacy considerations since they affect the way sets of Consumer keys and secrets are handled. An interesting finding of our analysis is, however, that in central questions of privacy the difference does only have a limited effect; the main privacy concern rests with the question of a multitenancy design (Section VI-A).

There is no ambiguity as to which entity is making each request. However, it means that from the perspective of each Service Provider, the only Consumer application that exists (that delegates over all end Consumer applications integrated over OAuthHub) is OAuthHub, and all requests are coming directly from there. This can be expressed as an "$n + 1$" relationship from the perspective of the Service Provider; we say "$n + 1$" because there are n Consumers per OAuthHub, and 1 Consumer, namely OAuthHub, per Service Provider.

OAuthHub definitely offers an advantage for new Service Providers. By just communicating with OAuthHub and inviting OAuthHub to become a Consumer, they can in turn reach all Consumers of OAuthHub.

C. Scalability Considerations

The scalability of the "$n + 1$" option could be a potential issue. Because all of the requests are made through OAuthHub's account with the Service Provider, there would be increased traffic through that account. Details are given in Section VI-E. Some Service Providers enforce rate limits on Consumers [17], [18], hence if many Consumers are added that all access the same Service Provider, the rate limit must be distributed to individual Consumers. This could potentially be solved by having "Consumer Pools"; the OAuthHub could have a large set of Consumer keys and secrets and the load could be balanced between them. In this case, there would be "$n + m$" sets of Consumer keys and secrets, where there are n Consumers per OAuthHub as before (in "$n + 1$") and m Consumers – all of which are controlled by the same OAuthHub – per Service Provider. The "$n+m$" setup is shown in Figure 4.

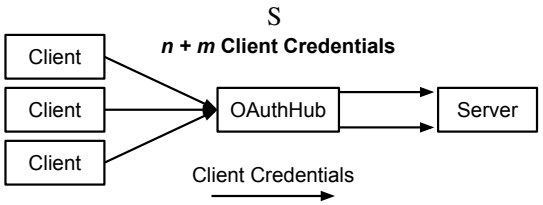

Fig. 4. OAuthHub with n + m Consumer keys and secrets

D. The OAuthHub Log-In Process

The setup of OAuthHub as two sets of three parties motivates a natural protocol for the communication between the end user, which is the Resource Owner, and the other three parties. In that process, all four parties have to communicate in a non-trivial protocol that is specified in Figure 5. When the Consumer wants to log Resource Owners in, the Consumer initiates the standard OAuth protocol with OAuthHub as the Service Provider.

Figure 5 shows the interaction involved in authorizing an end Consumer on OAuthHub. The interaction between the services begins when the user taps "Sign in via OAuthHub". The end Consumer makes a request to OAuthHub using its consumer token and secret. OAuthHub responds with a request token and secret. The browser is given the request token and redirected to the OAuthHub website. If the user is already logged into OAuthHub, then the interaction in the box Alt 1 can be skipped and the user can immediately authorize the end Consumer. The user will then be redirected back to the end Consumer's service. The OAuth authorization is complete when the end Consumer exchanges the request token and the OAuth verifier for an access token and secret.

A user might not have established an identity at the OAuthHub yet. The box Alt 1 in the figure demonstrates this case. The good thing is that even in this case the user does not have to separately register with OAuthHub but instead signs in with their preferred Service Provider. OAuthHub accomplishes this by acting as an OAuth *Consumer*, and interacts with an existing, traditional OAuth Service Provider where the user has already established an identity. Box Alt 2 describes the scenario in which the user has an identity at the Service Provider but is not currently logged into it.

E. Requesting Permission

We must decide which permissions are requested when a user adds a new Service Provider to their OAuthHub account. If OAuthHub requests all of the available permissions, then new Consumers can be added without having to request

more permissions from the end Service Provider. However, this means that OAuthHub may be granted access to more resources than it actually needs. An alternative to this would be to request new permissions incrementally, as required by each new Consumer. The downside to this approach is that there may be cases in which users must grant OAuthHub access to more resources on the Service Provider as well as granting the Consumer access to OAuthHub.

F. Privacy Control

In SaaS, there is the general question how the usage of data by the service can be ring-fenced; in principle, OAuthHub always represents one unit. This raises the question of multitenancy in OAuthHub. Specifically, data should only be available to Consumers with the correct authorization. The fundamental requirement of multitenancy of OAuthHub is independent from the question of which variant ("$n + 1$" or "$n + m$") is used. OAuthHub must guarantee to ring-fence the data obtained on behalf of different Consumers in all variants. However, strictly speaking the requirement on OAuthHub is not different from the requirement on any Service Provider to enforce the end-user's choices on limiting authorization for different Consumers. Multitenancy in software is not a new concept and has been well studied [19] [20].

Apart from the question of multitenancy, OAuthHub offers more interaction options for the user and therefore better features to protect their privacy [21].

G. Usability Considerations

One of the peculiarities of OAuthHub is that end users will – and will need to – become aware of OAuthHub. If users revoke OAuthHub's access to a Service Provider, the implicit, indirect access to that Service Provider held by all Consumers using OAuthHub will effectively be revoked in a cascading manner. Users should do this only if they distrust OAuthHub itself as an intermediary.

To revoke a specific Consumer's (implicit, indirect) access to their resources at end Service Providers, users should go through OAuthHub. As a consequence, OAuthHub has to provide users with an interface to manage the permissions between their (Consumers, Service Providers) pairs. This characteristic of OAuthHub from the point-of-views of users could be beneficial to users trying to manage their accounts, but it does introduce some complexity and could be detrimental to the experience for average users.

H. Interface Consolidation

When needing to access end-user's information or resources, Consumer applications that integrate with OAuthHub would forward the request through OAuthHub, without knowing which Service Provider the request would eventually be served by. This raises the question of "What is included in the scope of the resources that are available via OAuthHub for the Consumer?" If we consider that each Service Provider can provide a *set* of resources, then OAuthHub could offer either the *intersection* or the *union* of all sets of resources across all Service Providers. Providing the intersection gives Consumers a stronger guarantee, whereas providing the union gives more possibilities. Details are discussed in Section VI-C.

By combining these two ideas, OAuthHub can expose an API that offers a guarantee for some resources (the intersection) and offers the possibility of others (the union). Ultimately the choice of authorizing OAuthHub to a Service Provider is up to the user, and the user can decide to revoke any access at any time. The counter to this, from the perspective of the Consumers, is that Consumers can ask users to sign in using a certain service, as a condition of using that Consumer's service. In the considerations on consolidation of service-requests, OAuthHub and OAuthProxy face the same challenges.

I. Knowledge Curation and Maintenance

Service Provider APIs can change. New APIs can break existing apps and there is no guarantee that the new API will have the same capabilities that the old API did. This is a particularly troubling concern when supporting multiple Service Providers.

Currently it is up to the Consumer developers to keep track of API updates and ensure that everything continues to function as intended. Using our solution it would be OAuthHub's responsibility to respond to changes in Service Provider APIs. This is another reason that OAuthHub could be useful to Consumer developers.

J. Implementation Notes

OAuthHub, having an interface consolidation role, needs to keep up to date with the APIs of the whole range of supported Service Providers. Implementers could consider storing the API information either in the form of code (different implementations of the same adapter interface), or stored in a database in some form. If choosing the adapter approach, the implementer may consider existing consolidation studies and efforts, e.g. the Unified Service-Representation Model mentioned in ServiceBase [22]. If choosing the database storage approach, one could store a description of the API of a particular Service Provider for a given consolidated resource that is exposed to the Consumers; however, this may not be straightforward if the consolidation is more than just plain translation. For example, a consolidated API may need to be built from multiple underlying APIs, such as consolidating a "first name" and a "last name" resource into a consolidated resource "name" (reusing the example in Figure 3).

OAuthHub needs to provide a user interface to manage the access to different Service Providers that the user grants to different Consumers. Now that there is no explicit (OAuth) authorization for Consumers to access Protected Resources at Service Providers, OAuthHub is responsible for ensuring that each Consumer is not given more access than the amount the user specifies via the OAuthHub user interface.

V. Properties

In this section we discuss the properties of OAuthHub, namely the above mentioned setup involving "$n + 1$" sets of Consumer keys and secrets.

A. The OAuthHub setup does not violate the OAuth 1.0 protocol specification.

The specification states that, "The way in which the server handles the authorization request [...] is beyond the scope of this specification. However, the server MUST first verify the identity of the Resource Owner" [1, Section 2.2]. Indeed, if the user does not have an active cookie session with an OAuth Service Provider, when the user gets redirected to the authorisation URL (at the OAuth Service Provider), he would need to perform an ordinary username-password authentication. OAuth is capable of replacing this authentication process, by essentially off-loading it to some other Service Provider.

B. If using an OAuthHub, Consumer developers no longer need to interface with multiple OAuth Service Providers individually.

This is the most desirable property of the OAuthHub setup. If an OAuthHub allows users to log on using all the OAuth Service Providers which the Consumer developer would have supported manually, the Consumer developer will only need to interface with this OAuthHub in order to allow users to use existing identities established at various OAuth Service Providers. This would significantly lower the development cost of Consumer applications in the identity management aspect.

C. OAuthHub is non-exclusive to other OAuth Service Providers for the same Consumer.

OAuthHub is just another OAuth Service Provider from the perspective of Consumer applications' implementation. Just as a Consumer application can interface with both Twitter and Google, it can also interface with OAuthHub and Google. Choosing to use an OAuthHub still allows a Consumer application to interact with other OAuth Service Providers.

D. Consumer applications that switch to using an OAuthHub are required to make very little workflow changes.

This is also because "OAuthHub is just another OAuth Service Provider". The amount of work required for a Consumer application to start using an OAuthHub is exactly the same as the amount needed to start providing support for a new OAuth Service Provider, e.g. the amount when a new social network service becomes popular and Consumers starts to support it.

E. The Consumer is not authorized to (directly) access protected resources on the Service Provider.

This is a direct conclusion from the observation that at no point did the user authorize the Consumer to access protected resources at the Service Provider. In practice, of course *some* information flows from the Service Provider to the Consumer via the OAuthHub, e.g. the user's identity. However, since the OAuthHub re-wraps information from the Service Provider into the form of protected resources, the OAuthHub is completely in control of what to expose to the Consumer (see Section IV-F).

F. If a user wants to distrust a Consumer, revocation needs to be lodged to the OAuthHub involved.

What the user can revoke at the (end) OAuth Service Provider is the OAuthHub's access. This is a direct corollary from the previous property (Section V-E). Note that for the same OAuthHub, the user not only may revoke access *from* multiple Consumers to it, but may also revoke its access *to* multiple Service Providers.

G. (Compartmentalisation) The Consumer will not reliably know which OAuth Service Provider is the one where the existing identity that the user chose to use was established.

An OAuthHub, from the perspective of the Consumer and its developers, has just the same interface as any other OAuth Service Provider. There would not be any difference in the build-time interface this OAuthHub exposes, for different choices of OAuth Service Providers made by the user at run-time.

Indeed, an OAuthHub may expose an interface to state "at which OAuth Service Provider did this user establish their identity", but since the OAuthHub is free (or "capable") to provide inaccurate information, even though the Consumer *may* know the user's choice, it will not *reliably* know.

H. Any OAuthHub would be playing the role of an identity provider.

Once the user has authorized an OAuthHub to access protected resources held by a Service Provider, when the user comes across another Consumer that offers sign-in via the same OAuthHub, the user does not need to authorize that OAuthHub to the Service Provider again, but only need to authorize the new Consumer to that same OAuthHub. In effect, that OAuthHub has become an identity provider, analogous to OpenID's "Identity Provider" [5].

I. OAuthHub works independent of the version of OAuth (1.0 or 2.0).

OAuthHub is primarily designed with OAuth 1.0 in mind. However, regardless of whether OAuth 1.0 or OAuth 2.0 is used for its implementation, OAuthHub maintains the properties discuss above, and the challenges and limitations discussed below. Similarly, OAuthProxy also works with both OAuth versions.

VI. Challenges and Limitations

There are also challenges and limitations with the OAuthHub approach, which are discussed in the following.

A. The 4-party OAuthHub setup is prone to the "confused deputy" problem.

This problem arises due to the multitenant nature of OAuthHub. The user may authorize an OAuthHub to access protected resources held by *multiple* OAuth Service Providers. If the user also authorizes *multiple* Consumers to access that OAuthHub, that OAuthHub would need to properly keep track of which Consumer should be allowed to access resources at which Service Providers. An example could be that when the Consumer asks the OAuthHub for the user's human-readable name, the OAuthHub returns the name the user is known by at a *different* OAuth Service Provider from the appropriate one.

OAuthHub must also keep track of the level of authorization for each Consumer. For example, if a user's social media service provided both read and write permissions. One could imagine a situation in which a user would like their music service to be able to both read and write to their social media service but their calendar should only have read access.

In addition to the previous two issues, OAuthHub must ensure that Consumers are only accessing resources that belong to the user that authorized them. For example, if one user authorizes their music service to access their social media service then the music service should not be allowed to also access another user's social media service.

If the OAuthHub makes a mistake on this association, Consumers may end up accessing resources at a Service Provider which they are not supposed to be able to access — despite never reliably knowing *which* exact Service Provider the information is from due to compartmentalization (Section V-G).

B. Any OAuthHub would be an ideal target for attack.

Given that OAuthHubs are essentially identity providers (Section V-H), any OAuthHub would, after some amount of user usage, be authorized to access a vast amount of protected resources held by various OAuth Service Providers. If an attacker gains privilege to control resources held by an OAuthHub, they would be able to access resources owned by various resources owners, at various OAuth Service Providers.

*C. (Consolidation) There is only a small amount of information that an OAuthHub can **guarantee** to provide (although there is a large amount of information that the same OAuthHub **may** provide).*

An OAuthHub would need to define its own interface for OAuth Consumers to access users' information. There is only one such interface, but multiple potential data sources (namely the multiple OAuth Service Providers) providing information through this interface.

Consider any set S of OAuth Service Providers that a given OAuthHub supports, such as $S = \{\text{Google, Twitter, Facebook, } \dots\}$. Let R be a function that maps an OAuth Service Provider to the set of protected resources provided by that Service Provider. The set of resources that any OAuthHub can *guarantee* to provide is $\bigcap_{s \in S} R(s)$ – which can be very small, because even if we ignore the syntactic, interface differences across different OAuth Service Providers, the semantic content of the resources can be quite different. For example, you cannot post a tweet by only interacting with Google. However, the set of resources that any OAuthHub *may* be able to provide is $\bigcup_{s \in S} R(s)$, which may be very large and diverse, for the same reason.

This means from a practical point of view the following may happen. A Consumer may want to retrieve a resource, e.g. a "profile photo" of a user, via OAuthHub. However, OAuthHub may refuse (in fact, be incapable) to provide an interface to get a "profile photo", because some of the OAuth Service Providers the user has logged on with may not have the notion of a "profile photo".

D. The 4-party OAuthHub authorization interaction sequence would seem confusing to the end-user.

From a user experience point of view, it may be overly confusing for the user to need to make more than one "authorization". Seeing that when the end OAuth Service Provider asks the user to grant access to the OAuthHub, the UI would not (and could not) say anything about the end Consumer, hence the user might be confused as to what exactly he/she is authorizing. Because there are two different types of revocation that may be done (Section V-F), when the user wishes to revoke access of one Consumer to their information, they may end up revoking the entire OAuthHub's access to the identity-providing OAuth Service Provider. This may lead to unintended implicit revocation of other Consumers that makes use of the same OAuthHub.

E. The OAuthHub setup faces inherent scalability issues.

Most OAuth Service Providers pose a rate limit on their OAuth or REST API endpoints [17], [18], [23]. This highlights a scalability problem OAuthHub faces.

Consider an OAuthHub H that is authorized by the user to access a set S of Service Providers and to be accessed by a set C of Consumers. Let L ("Load") be a function that maps a (Consumer, Service Provider) pair (c, s) to the number of users who own some information that flows from the Service Provider s to the Consumer c. Then for all Service Providers s in the set S, the OAuthHub's load on s is: $L(H, s) = \sum_{c \in C} L(c, s)$.

This could lead to the following scenario. Let us assume for a given OAuthHub, a new Consumer that recently started using this OAuthHub has a very large number of users who prefer using Twitter identities. Then this OAuthHub's usage load on Twitter's APIs may significantly increase. This may in turn lead to Twitter blocking the OAuthHub's access.

F. Consumers may be unsatisfied by the (small) variety of resources available at any given OAuthHub.

This directly follows from the consolidation limitations discussed in Section VI-C that an OAuthHub can only provide the intersection of the resource-sets of the multiple Service Providers. If a Consumer application's business logic requires functionality as specific as, for example, "reading the user's

Facebook wall", this Consumer would be disappointed, because it is unlikely that any OAuthHub would provide Service Provider-specific interfaces like this.

It would be reasonable for developers of such Consumers to interface with "Facebook" (in this example) directly, instead of using a solution like OAuthHub. OAuthHub as a solution only attempts to be helpful in cases where the Consumer application is using OAuth mainly (if not entirely) for the purpose of identifying users, as mentioned before.

VII. Comparison with OAuthProxy

A. Description of OAuthProxy

In this variant of the OAuth trusted intermediary each Consumer generates their own Consumer key and secret with each Service Provider. We will call this variant "OAuthProxy" in the following. When Consumers register with OAuthProxy, they provide their Consumer key and secret for use in communication between OAuthProxy and each Service Provider. Each Consumer is given a new set of Consumer key and secret by OAuthProxy for all future requests between them and the OAuthProxy.

One property and potential benefit of OAuthProxy is that the Service Providers are aware of the Consumer and all requests are made under the Consumer's name. Thus, from the perspective of the Service Providers, OAuthProxy does not exist. This could be beneficial as users could then revoke permissions from Consumers through the Service Provider. The critical step that warrants scrutiny here is the passing on of credentials from the Consumer to OAuthProxy. This stands in contrast to OAuthHub where there would be no problematic sharing of Consumer keys and secrets.

Interesting questions around this scenario are about the general concept of SaaS and the implications for agreements with third parties, the Service Provider in this case. If we take a bold stand on SaaS, then OAuthProxy is deemed permissible and unquestionable, because OAuthProxy acts as part of the Consumer application IT, i.e. it is SaaS used by the Consumer. If we deem this as problematic then there are fundamental and hitherto unexplored limitations to the very notion of SaaS. The OAuthProxy concept surely highlights questions about SaaS that do not go away easily.

There is an existing solution in the market called "OAuth.io" [6], which plays a very similar role as OAuthProxy among the user, Consumer developers, and OAuth Service Providers. OAuth.io requires Consumer developers to directly self-register at all desired end OAuth Service Providers as Consumers, and then *hand the Consumer key and secret to OAuth.io*. OAuth.io indeed describes itself as "an OAuth backend-as-a-service for your applications" [24].

We discuss now some of the differences between OAuthProxy and OAuthHub.

B. OAuthProxy addresses some of OAuthHub's usability limitations from Section VI-D.

Now the end OAuth Service Provider would know which end Consumer intends to access resources, so it can present an authorization UI that informs the user of the Consumer. Note that this isn't possible with OAuthHub, due to there being only one OAuthHub, but multiple Consumers (Section VI-E). There would also be only one — rather than two — authorization prompt, which is good in a usability perspective. Similarly, "two different revocations" (Section V-F) no longer exist, which also reduces confusion.

C. OAuthProxy addresses scalability issues from Section VI-E.

This is because now that in the end OAuth Service Provider's perspective, for the same set of end Consumer applications, there are more registered OAuth Consumers (rather than just one, namely the OAuthHub), each Consumer application takes their own load, rather than affecting the overall performance of the intermediary.

D. OAuthProxy is less desirable in privacy or information compartmentalization aspects.

Consumers will know exactly which end OAuth Service Provider the user chose to use (Section V-G). However, the positive side to this is that Consumers will not face the problem of not being able to access resources they want due to consolidation limits as stated in Section VI-C.

E. OAuthProxy demands more trust from Consumers; OAuthHub demands more trust from users.

In both OAuthHub and OAuthProxy, the users must trust the Consumers not to abuse their resources.

In the case of OAuthProxy, when a user authorises a new Consumer, OAuthProxy will act on behalf of the Consumer. The user would also authorise each Consumer directly at whichever Service Provider is used. This means that Consumers must trust OAuthProxy to act responsibly as all requests will be under their name, but the users can directly revoke individual Consumers' access at the corresponding Service Provider.

In the case of OAuthHub, When a user authorises OAuthHub access to a Service Provider, the user must trust OAuthHub to act responsibly, since OAuthHub may demand a wide range of permissions in order to potentially expose resources to downstream Consumers. This could be an issue if on one hand, a Consumer offers OAuthHub but not traditional Service Providers as recognised identity providers for log-in, but on the other hand, users do not trust OAuthHub to have such power.

VIII. Conclusion

In this paper we have summarized the motivations for OAuth and the complications of different Service Providers having different APIs. We have proposed a solution to these complications, namely we have introduced a new party, OAuthHub, to the interaction in order to consolidate a vast array of APIs into one. All requests from the Consumer are passed through OAuthHub before being processed and sent to the Service Provider.

Furthermore, we have provided an analysis on various aspects of OAuthHub, including security, usability, cost, scalability, and compared it with a variant, OAuthProxy. OAuthHub shows desirable properties regarding development cost, compatibility with existing workflow, and information compartmentalization, while it may have potential problems regarding security, usability, and functionality.

The difference between OAuthHub and OAuthProxy includes where trust is placed and how it is managed. OAuthHub (Section IV-A) is not only an assistance to developers, but can also be a tool for users to manage their authorisations between many (Consumer, Service Provider) pairs. OAuthProxy (Section VII-A) is purely a service to handle the consolidation over various APIs or authentication procedures; it is only used (or known) by developers of Consumer applications rather than also by users.

Future research directions include the implications of the practice of using OAuth as an identity management solution where OAuth Service Providers are treated as identity providers.

REFERENCES

[1] E. Hammer-Lahav, "RFC 5849: The OAuth 1.0 protocol," *Internet Engineering Task Force (IETF)*, 4 2010.

[2] D. Hardt, "RFC 6749: The OAuth 2.0 authorization framework," *Internet Engineering Task Force (IETF)*, 10 2012.

[3] A. Pashalidis and C. Mitchell, "A taxonomy of single sign-on systems," in *Information Security and Privacy*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and J. Seberry, Eds. Springer, 2003, vol. 2727, pp. 249–264.

[4] B. Kissel, "OpenID 2009 year in review," OpenId. [Online]. Available: http://openid.net/2009/12/16/openid-2009-year-in-review/

[5] J. Bellamy-McIntyre, C. Lutteroth, and G. Weber, "OpenID and the enterprise: A model-based analysis of single sign-on authentication," in *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2011, pp. 129–138.

[6] "OAuth that just works." [Online]. Available: https://oauth.io/

[7] "Webshell.io: building the web operating system," http://webshell.io/blog, [Online; accessed 26-July-2015].

[8] C. C. Wood, "Effective information system security with password controls," *Computers & Security*, vol. 2, no. 1, pp. 5 – 10, 1983.

[9] R. Dhamija and L. Dusseault, "The seven flaws of identity management: Usability and security challenges," *Security & Privacy, IEEE*, vol. 6, no. 2, pp. 24–29, 2008.

[10] M. Atwood *et al.*, "OAuth core 1.0 revision a," 6 2009. [Online]. Available: http://oauth.net/core/1.0a/

[11] D. Recordon and D. Reed, "OpenID 2.0: A platform for user-centric identity management," in *Proceedings of the Second ACM Workshop on Digital Identity Management (DIM)*. ACM, 2006, pp. 11–16.

[12] E. Hammer-Lahav, "OAuth 2.0 and the road to hell," 2009, http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/ [Online; Accessed May 10, 2015].

[13] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, "Formal verification of OAuth 2.0 using Alloy framework," in *Proceedings of the International Conference on Communication Systems and Network Technologies*. IEEE, 2011, pp. 655–659.

[14] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 378–390.

[15] D. Spinellis, "Another level of indirection," in *Beautiful Code: Leading Programmers Explain How They Think*, A. Oram and G. Wilson, Eds. O'Reilly, 2007, ch. 17, pp. 279–291.

[16] W. Bin, H. H. Yuan, L. X. Xi, and X. J. Min, "Open identity management framework for SaaS ecosystem," in *Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE)*, Oct 2009, pp. 512–517.

[17] Twitter, "API rate limits." [Online]. Available: https://dev.twitter.com/rest/public/rate-limiting

[18] GitHub, "GitHub API v3." [Online]. Available: https://developer.github.com/v3/#rate-limiting

[19] T. Takahashi, G. Blanc, Y. Kadobayashi, D. Fall, H. Hazeyama, and S. Matsuo, "Enabling secure multitenancy in cloud computing: Challenges and approaches," in *Proceedings of the 2nd Baltic Congress on Future Internet Communications (BCFIC)*. IEEE, 2012, pp. 72–79.

[20] W. J. Brown, V. Anderson, and Q. Tan, "Multitenancy-security risks and countermeasures," in *Proceedings of the 15th International Conference on Network-Based Information Systems (NBiS)*. IEEE, 2012, pp. 7–13.

[21] G. Costantino, F. Martinelli, and D. Sgandurra, "How to grant less permissions to facebook applications," in *Proceedings of the 9th International Conference on Information Assurance and Security (IAS)*, Dec 2013, pp. 55–60.

[22] M. C. Barukh and B. Benatallah, "Servicebase: A programming knowledge-base for service oriented development," in *Database Systems for Advanced Applications*. Springer, 2013, pp. 123–138.

[23] Google, "Usage limits - Gmail REST API." [Online]. Available: https://developers.google.com/gmail/api/v1/reference/quota

[24] "OAuth.io frequently asked questions." [Online]. Available: https://oauth.io/faq

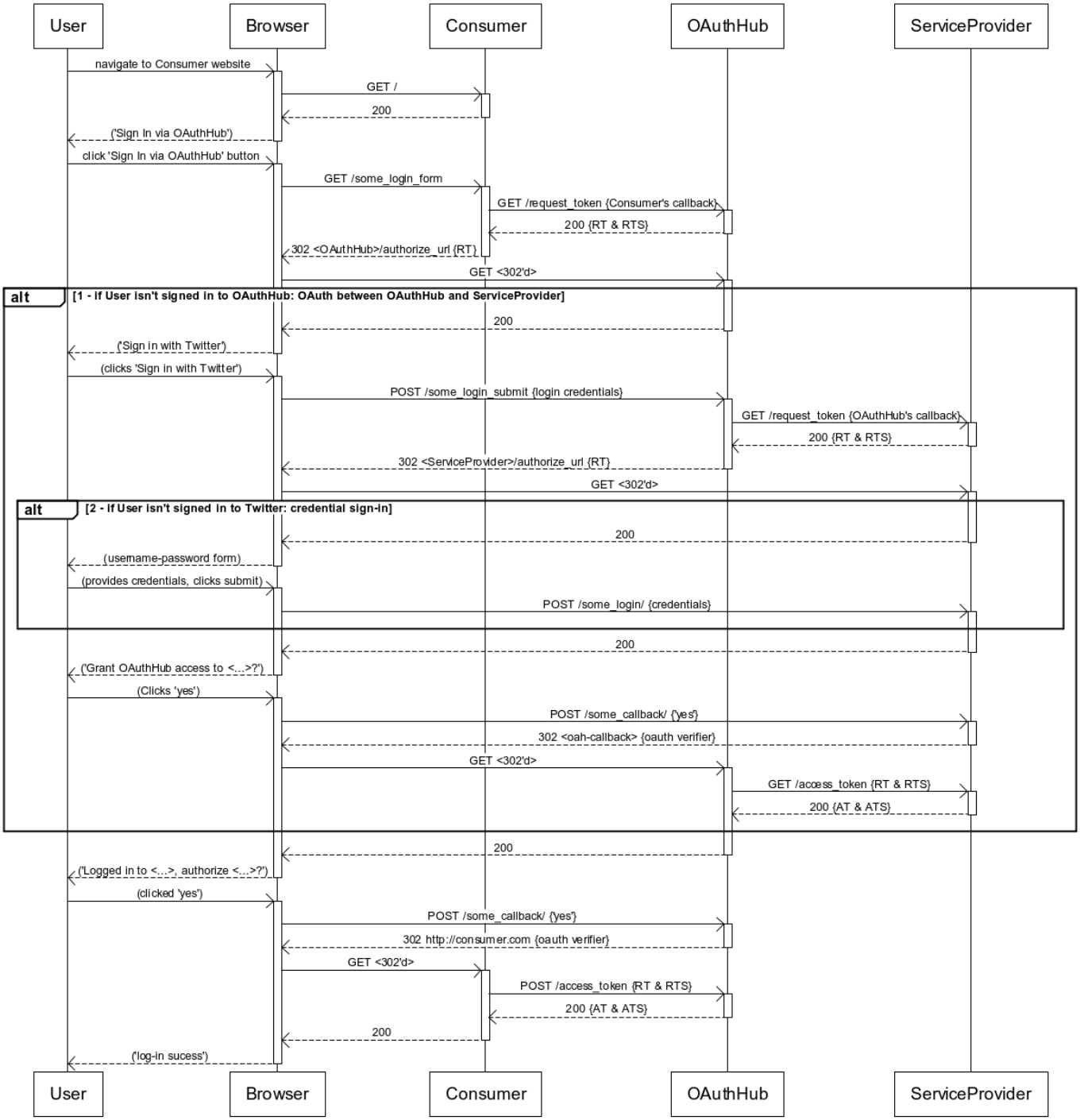

Fig. 5. Interaction diagram for OAuth protocol when logging in using OAuthHub