Please Permit Me: Stateless Delegated Authorization in Mashups

Ragib Hasan, Marianne Winslett
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{rhasan,winslett}@cs.uiuc.edu

Richard Conlan, Brian Slesinsky,
and Nandakumar Ramani
Google Inc.
Mountain View, CA 94043
{zeveck,skybrian,nramani}@google.com

Abstract

Mashups have emerged as a Web 2.0 phenomenon, connecting disjoint applications together to provide unified services. However, scalable access control for mashups is difficult. To enable a mashup to gather data from legacy applications and services, users must give the mashup their login names and passwords for those services. This all-or-nothing approach violates the principle of least privilege and leaves users vulnerable to misuse of their credentials by malicious mashups.

In this paper, we introduce delegation permits – a stateless approach to access rights delegation in mashups – and describe our complete implementation of a permit-based authorization delegation service. Our protocol and implementation enable fine grained, flexible, and stateless access control and authorization for distributed delegated authorization in mashups, while minimizing attackers' ability to capture and exploit users' authentication credentials.

1 Introduction

Mashups have become popular as a rapid method of creating a new service through composition of several different existing services on the Web [21]. Mashups provide the user with an integrated view of the information that they gather from back-end services, which can be stand-alone websites (such as the bank and financial service sites accessed by Mint.com [2]), or web services accessed via web-APIs (such as flickr's photo API). Since building a mashup from existing applications and services is much easier than building a comparable service from scratch, mashups have proliferated on the Internet [14, 27].

Providing a framework for authorization in a mashup is difficult. Typically, users give the mashup their authentication credentials for the back-end services that the mashup will access. The mashup then goes on to impersonate the user to those back-end services, and gain access to those services in the same manner as the user would. The back-end services do not differentiate between the user accessing the service and the mashup accessing the service *on behalf of* the user. Further, access to the back-end services is all-or-nothing – users delegate all of their privileges to the mashup, or else none of them.

A mashup built in such manner introduces many security and privacy risks. For example, if the mashup server is compromised, the attacker can take over all the accounts in the back-end services by capturing the user authentication credentials stored in the mashup. Also, mashups often get more privileges than they really need. For example, even if the mashup just needs to read a user's calendar, the mashup will receive all of the user's calendar privileges.

To solve these security vulnerabilities, finer-grained delegation of access rights is needed. Specifically, we need access delegation for mashups, in which users can *selectively* delegate their back-end service privileges to the mashup. In this paper, we make the following contributions toward solving this problem: we identify the mashup authorization requirements by examining the problem domain and existing solutions; we provide a scalable, stateless protocol for access delegation using *delegation permits*; and we describe our implementation of delegation permits and their associated protocols, which we developed for real applications.

The rest of the paper is organized as follows: we discuss existing mashup authorization models and their limitations in Section 2. We describe our permit-based approach, protocols, and the architecture of our prototype in Section 3. Section 4 describes related work in the field of distributed authorization. Finally, we conclude in Section 5.

2 Mashup Authorization: Problems and Current Solutions

In this section, we explore the mashup authorization problem and its associated security issues.

Figure 1 shows the architecture used by most mashups. When registering herself at the mashup application, the user

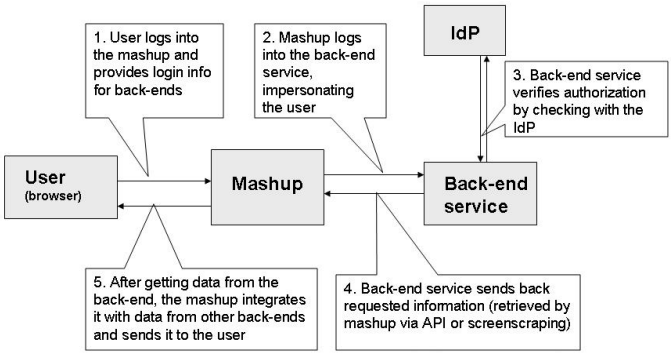

Figure 1. Architecture of a typical mashup

adds back-end services to her profile. Usually the user already has credentials (typically logins and passwords) that allow her to authenticate herself to each of these back-end services directly. During registration at the mashup, the user gives the mashup her authentication information at each of the back-end services. Later, during actual use, the user logs into the mashup. The mashup contacts an *identity provider* (such as the box labeled IdP in Figure 1), which verifies the user's identity. The identity provider can be a local service running in the same domain as the mashup, or a global service such as OpenID [28]. After the user logs in at the mashup, the mashup goes to the back-end services defined in the user's profile and accesses them on her behalf. Some of the back-end services may have mashup support via an API, while some others may not have any built-in support for mashups. In the latter case, the mashup impersonates the user, and logs into the back-end services via a simulated browsing session. Some mashups also rely on screen scraping, making them complicated and brittle. In any case, the back-end services cannot differentiate between the user logging in directly, and the mashup impersonating the user. In the remainder of the paper, we refer to this as the *strawman approach*. This approach is adopted by, for example, the personal finance mashups Mint.com and Yodlee.com, which provide a summary of a user's financial activities by accessing financial data from back-end services such as banks and credit card companies. A variant of the strawman approach uses HTML IFRAME tags. The mashup creates an IFRAME for each of the back-end services, and the user logs into them. The mashup extracts data from the IFRAMEs and displays it to the user. AuthSub [13] is a protocol designed and deployed by Google for authorization of Google services. In AuthSub, when the web application needs to access the user's Google service data, it makes an AuthSub call to the Google Accounts URL. Google Accounts responds with an "Access Consent" page. This page prompts the user to log into their Google account and grant or deny access to the Google service. If the user denies access, she is directed to a Google page rather than back to the web application. If the user successfully logs in and grants access, Google Accounts redirects the user back to the web application URL. The redirect contains an authentication token good for one use; it can be exchanged for a long-lived token. The web application contacts the Google service, using the authentication token to act as an agent for the user. If the Google service recognizes the token, it will supply the requested data.

OAuth [4] is a new protocol for distributed authentication, and is designed to allow a consumer application to get limited access to a service provider, as granted by the user. In OAuth, a consumer application wishing to access a service gets an unauthorized request token from the service. This token is given to the user, and the user is redirected to the service provider. The service provider gets user approval for the request, converts the token to an authorized request token, and redirects the user to the consumer application. The consumer extracts the authorized request token, and uses it to get an access token containing a shared secret from the service. The consumer can later use the token secret together with the access token to access resources at the service. OAuth requires the service to maintain the state of all previously issued token secrets and access tokens.

Discussion. While the authorization schemes described above are widely used in current-generation mashups, they have serious limitations. The strawman scheme requires the user to trust the mashup completely. The mashup receives all the privileges the user has in a back-end service, and there is no way for the user to restrict the mashup's capability. A compromised mashup can leak user credentials or impersonate the user maliciously. There is no explicit revocation; the mashup can impersonate a user at back-end services until the user changes her credentials manually at the back-end. Some mashups will, of course, let the user remove credentials from the system, but this requires trusting the mashup completely.

Phishing is another significant risk in the strawman schemes [12]. An attacker can set up a look-alike site impersonating a mashup and send phishing emails to users. Gullible users can visit the phishing site and reveal their usernames and passwords for the back-end services.

OAuth is a better approach in this regard, and promises to solve the trust delegation problem. However, it assumes a stateful server to provide nonces and ensure tokens are single-use, which does not scale well [4]. Many IdPs and in-house single sign-on solutions may be cookie-based with stateless servers; thus setting up an OAuth instance would require dedicated resources and support. The OAuth core-spec [4] also does not define access restrictions or scope.

Google's AuthSub [13], on the other hand, requires a RPC API call to validate the token. The token itself is an opaque identifier with no information about what back-end service, mashup, and user the token pertains to. Therefore,

to decipher a token, additional RPC calls are required. An AuthSub interaction only grants a single-use token. Getting a session token requires an additional step, and the client must explicitly revoke the session token at the end of the session. Explicit revocation of the token may be awkward in typical usage and in error situations, or even impossible; thus there may be many valid, long-lived tokens floating around. To allow the user to review and revoke any such leftover tokens from previous session, the AuthSub server must keep state for all issued session tokens. The resulting state maintenance problems make this approach unscalable. To remedy this problem, AuthSub has limited the number of tokens that can be issued at one time. The AuthSub specification does not allow "more than ten valid tokens per user, per web application" [13].

AuthSub requires registration of both back-end and front-end services. The latter is required in part so that all requests from the front-end applications (mashups) to AuthSub servers can be signed by the front-end in order to issue a secure token. The registration also allows the front-end to pre-supply a description of the authorization request, which is displayed by AuthSub to the user. Mashup developers consider registration to be cumbersome, and requiring each front-end to have its own certificate is burdensome.

Registration headaches aside, an AuthSub request is less verbose and flexible than the approach that we propose in this paper. Notably, AuthSub requires the mashup to issue a separate token request for *each* back-end service that it expects to access on the user's behalf, resulting in many RPC calls to the AuthSub server. AuthSub access decisions are all-or-nothing; there is no notion of requesting a particular level of service. Finally, AuthSub-based solutions do not, at present, handle key rotation on the server side.

Based on the discussion above, we can list the common issues and requirements in mashup authorization.

- **Delegation and trust.** Users should not have to provide their user authentication information to the mashup. Authorization information/tokens should not be stored in the mashup; rather, such sensitive authorization information should be stored at the user side. Authorization tokens should have a limited lifespan, and the user should be able to revoke them, if necessary, before their expiration time.
- **Maintaining state.** To be scalable, the mashup authorization system should be stateless. It is not reasonable to assume that the back-end services maintain the state for the authorization tokens that they have issued to various mashups. Such detailed state maintenance is not feasible in real life applications and services.
- **Fine grained control.** Users should be able to provide fine-grained delegation. In other words, mashups should receive exactly the permissions they require to function. For example, if all a mashup needs is to read user data from a back-end service, then the mashup should only receive READ permission instead of READ-WRITE permission.

3 Delegation Permits

In real life, people get permits for fishing, driving, entering national parks, and many other activities. Such permits typically include the name of the issuer, the person to whom it is issued, the issue date, expiration time, and purpose of the permit. The person who checks permits does not have to know the permit-holder, nor does she have to contact the permit issuer to decide whether the holder is authorized. This makes the authorization process decentralized and scalable.

Our approach mimics real-life permit-based authorization schemes. In our model, a mashup asks the user to grant it *delegation permits*, which are unforgeable, limited-lifetime, digital tokens specifying the access rights to specific services. The user can see what permissions the mashup is requesting, and must approve the request before the permit can be issued. Once a permit is issued, the mashup can use it until the permit expires or is revoked.

When the mashup accesses a back-end service, it sends the appropriate permit along with its request. The back-end service makes its own authorization decisions based on the permit. The user can review the permits they have issued as well as indicate whether any given permit should be renewed, revoked, or modified.

More formally, a delegation permit $P(U, M, A, I, E, R, sig)$ with issuance time I, expiration time E, and signature sig is an unforgeable token issued at the behest of user U to mashup M, in order to delegate selected access rights at the back-end application or service A. The access rights are represented by a set R of user-intelligible string-valued *permit descriptors*. The back-end service must trust the issuer who signs the permit; the back-end service can verify the signature using the public key of the issuer. In general, a mashup that presents an untampered, unexpired permit to A will be able to obtain certain authorizations as specified in R. However, permits do not directly represent authorization decisions; the back-end server has final authority over what actions the mashup can take.

Permit descriptors are free-form text defined by a back-end service, and made available to everyone, which denote the different levels of access supported by the back-end service. The format and semantics of the descriptors are decided by each back-end. For example, a back-end may use the descriptor "MyBugTracker READ-ONLY" to denote read-only access, while another back-end service may choose to use "MyProjectDB RD-ONLY". The back-end services also make available a human readable explanation string for each of the permit descriptors. The descriptors

also allow multiple levels of delegation: if a permit descriptor ends in a *, then it can be further delegated. For example, if a mashup receives a permit with "READ*", it can issue a new permit to another application that effectively encapsulates the old permit, and provides a subset of the functionality of the original permit. Descriptors can be separated with a /, and any subset can be further delegated. For instance: the permit descriptor "READ*/WRITE*" allows the recipient mashup to delegate the following permits to others: READ, WRITE, READ/WRITE, READ*, WRITE*, READ*/WRITE*, READ*/WRITE, READ/WRITE*.

Anyone can verify the signature on a permit, using the issuer's public key. When a back-end service is presented with an unexpired, untampered permit from an issuer that it trusts, the service can verify the permit locally with its copy of the issuer's public key, with no additional messages. Specifically, the back-end server will not need to interact with any remote party before authorizing the bearer of the permit to carry out its requested actions.

In general, possession of an unexpired, untampered permit with the appropriate permission descriptors is sufficient in itself to authorize a user at a service that trusts the permit issuer. Because knowledge of a permit is sufficient to authenticate a user to the service, permits must be kept secret. For this reason, we will only transfer permits across the internet using SSL, unless the back-end service that the permit is destined for requires authentication purely for informative, rather than security, purposes.

A major advantage of delegation permits is that the back-end can differentiate between the user logging in and a mashup logging in on behalf of the user. This allows the back-end to adapt output for computer consumption instead of human readability, and to define limitations specific to mashups, e.g. throttling. Also, if a security event occurs, an audit will at least reveal whether it was the user directly or a mashup that caused the event.

We use a client-server approach for issuing permits. As shown in Figure 2, the architecture is composed of the following components:

Permit Grant Service (PGS). The permit grant service is responsible for issuing permits. Upon request from a mashup, it prompts the user with a list of access rights requested by the mashup. The user can choose which permissions to delegate to the mashup. The user's choices are encoded into the permit, and the permit is timestamped and signed by the permit granting service. The permit is given to the mashup, which can attach it to access requests to the corresponding back-end service.

Permit History Service (PHS). The user can view, renew, and revoke her existing permits through the permit history service.

Permit Handler Service. The permit handler service runs at the mashup and handles permits once they are received from the permit granting service.

The PGS and PHS services can be centralized or distributed. In practice, there may be more than one permit server in an organization, and the user can specify what permit server to use when registering with a mashup. Users may also use different permit servers for different mashups (e.g., one for official use, another for personal activities).

To communicate with one another, the back-end applications and the mashup use a new library API that allows service requests to be accompanied by permits. On the back-end servers' side of the API, the application writers must provide additional code so that the applications can make authorization decisions based on the permit descriptors.

3.1 Protocols

3.1.1 Permit Grant Protocol

Suppose that a user is about to access a mashup running at `https://mycoolapp.com/app`, which in turn accesses back-end services `mybugtracker.com` and `myprojectdb.com`. The user's PGS is running at `https://permitserver.com/permit`. The *Permit Granting Protocol* ensures that the user will always be presented with a *delegate-permissions page* the first time she logs in to a mashup that will need delegated authorizations. The user can grant or deny the mashup's request for each permit, and can choose to have her decisions recorded for longer-term use. If the user opts to remember her decisions, then she will receive a signed permanent cookie so that repeated attempts to access the *same* back-end application with the *same* access requests (or a subset thereof) do not result in another trip to the delegate-permissions page.

Figure 2 shows the sequence of events that take place during an invocation of the Permit Grant Protocol:

1. The user points her browser to the mashup page at `https://mycoolapp.com/app`. During the login process, the mashup authenticates her using its IdP (this interaction with the IdP is not shown in the figure).
2. The mashup checks whether the user has the necessary permits already stored in her cookie for the mashup's domain. If not, the mashup redirects her browser to the PGS at `https://permitserver.com/permit?PERMIT_REQUEST_ARGS`, where `PERMIT_REQUEST_ARGS` indicates the requested permit descriptors for *all* the back-end services it expects to access on the user's behalf. The format of this redirect is described in the Appendix.
3. The browser follows the redirect to the PGS.
4. The PGS authenticates the user and parses the requested permit descriptors and back-end service infor-

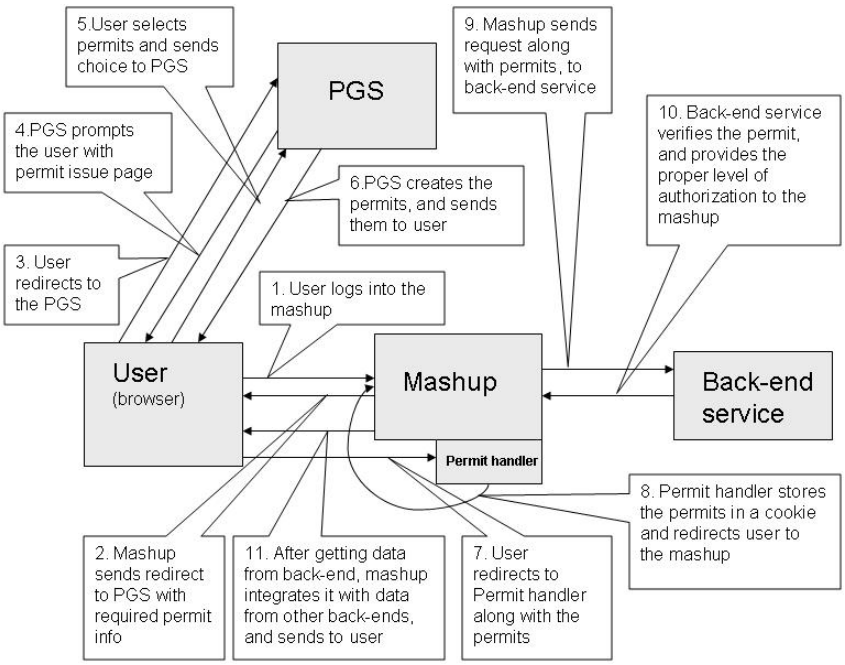

Figure 2. Permit-based delegation architecture, and the Permit Grant Protocol.

mation from the redirect URL. Using these permit descriptors, and corresponding human-readable descriptions obtained from the back-end services, the PGS renders its permission granting page with the list of requested permissions.

5. The user tells the PGS whether she is willing to delegate the requested permissions to the mashup and whether her decisions should be remembered for future sessions.
6. If the user approves the delegation requests, the PGS issues the appropriate permits, stores a signed list of the user choices in the *permit history cookie* for the domain of the PGS, and redirects her browser to the permit handler service at `https://mycoolapp.com/permithandler?PERMIT_RESPONSE_ARGS`. The format of the redirect is described in detail in the Appendix.
7. The browser follows the redirect to the permit handler service.
8. The permit handler service reads the permits, stores them in a *permit cookie* for the mashup's domain, and then redirects the user to `https://mycoolapp.com/app`.
9. Now MyCoolApp can find the required permits by reading the permit cookie for its domain. It sends service requests to MyBugTracker and MyProjectDB, with each request accompanied by the corresponding permits. (If the user did not approve all the requested permits, it is up to the mashup developer to decide what to do. The mashup can choose to provide the user with a restricted service based on the permissions granted, or the mashup can display an error message regarding unavailability of required permits.)
10. MyBugTracker and MyProjectDB extract the corresponding permits from the requests. After validating the permit signatures with a cached copy of the PGS's public key and checking the permits' expiration time, they provide the requested service to the mashup.

3.1.2 Permit review and revocation protocol

To review her permits, the user goes to the Permit History Server. The following protocol is used for permit history review and revocation:

1. The user points her browser at the PHS at `https://www.permitserver.com/history`.
2. The PHS parses the information in the user's permit history cookie for the PHS domain and presents the list of current permits to the user.
3. The user can ask the PHS to renew a permit. This causes the PHS to redirect to the PGS along with a request to renew the appropriate permits. The renewal is accomplished by redoing steps 3 to 8 of the Permit Grant Protocol.
4. The user can also revoke all or a subset of her permits. Revocation in this case means removal of the permits from the permit cookie belonging to the mashup's domain. To do that, the user selects the revocation option at the PHS. This causes the PHS to redirect to the

permit handler server of the chosen mashup, with a request to delete the permit. (The PHS cannot do this directly as the permit cookie belongs to the mashup's domain.)
5. Upon a request from the PHS, the permit handler service deletes from the user's permit cookie those permits that the user wants to revoke.
6. The permit handler service redirects the user back to the PHS.

3.2 Attack Resistance

A mashup authorization system can be attacked in several ways. The mashup application itself may be malicious, or the server hosting the mashup may be taken over by an adversary. In that case, we want to limit the extent to which the adversary can misuse the permits. Since we store the permits as browser side cookies, a compromised mashup server cannot divulge permits issued prior to the compromise unless it chose to cache the old permit cookies before the attack.

If a user logs into the compromised mashup, the adversary can obtain newly issued permits by capturing them at the permit handler. Since her permits give the mashup a limited set of rights, the attacker can only take the actions allowed by the permits. As permits are of limited lifetime, old permits will expire and become useless. As long as the user does not continue using the compromised server, the adversary has a limited window of opportunity to access the back-end services. Since the PGS authenticates the user when issuing permits, a compromised mashup cannot send fake user requests and get permits.

A compromised permit handler can also refuse to delete permits when requested to do so by the PHS, or could fail to insert new permits into the permit cookie when requested to do so by the PGS (denial of service). In those situations, the permit history cookie can become inconsistent with the actual permit cookie. A compromised handler could also update the permit cookie appropriately, but keep a copy of an old version and substitute it for the correct permit cookie during a subsequent session. Of course, reuse of revoked credentials in this manner is only possible within the lifetime of the original credential.

The other defensive measure against a compromised mashup is to use permits that are only good for a single session, as described in more detail in the Appendix. Such permits have to be reissued each time the user logs into the mashup, and have a very limited lifetime. Therefore, the window of opportunity during which the adversary can attack is quite limited. In general, the permit expiration time must be chosen to balance usability (users should not be prompted for permits too frequently) and risk (permits should expire quickly enough to prevent a malicious/compromised mashup from misusing old permits).

The user's own machine can also be compromised and the user's browser taken over. We limit the potential for permit leaks under these circumstances by using session cookies for permits stored in the browser. Once the user logs out and closes her browser, the permit cookies are not available to the adversary. To be able to impersonate the user at the mashup and exploit the permit cookies in that manner, the attacker must be able to authenticate as the user at the IdP, which will be very hard to do directly. A more effective approach is to wait until the user has already authenticated at the IdP and then hijack the user's session with the mashup. This same sort of attack can be employed with other approaches to mashup authorization as well, of course.

Attackers can also launch man-in-the-middle attacks by listening to the communication between the different components. Hence, we require all communication to take place over SSL. There is still a chance of a DNS spoofing attack, where an adversary spoofs the DNS to hijack the communication between the various components. However, such attacks are a problem for all web-based authorization systems.

Finally, adversaries can launch a denial of service (DOS) attack by overwhelming the Permit Grant Service. To prevent the PGS from becoming a single point of failure, our design allows multiple PGSs to be used. By running the PGS behind a load balancing mechanism such as a Netscaler [1], we can achieve proper load balancing, and help to defend against DOS attacks.

3.3 Implementation

To demonstrate our approach, we implemented a permit-based authorization system prototype using Java 1.5, and did a test deployment in a corporate network. We chose two existing applications, a bug tracking service and a project database system, and wrote a prototype mashup (MyCoolMashup) that integrated these two back-end services. In our test deployment, we used our in-house corporate single sign-on system as the identity provider IdP.

For simplicity, we created the permit granting service and the permit history service as servlets running under the IdP server. However, we could have used any IdP, or run the PGS/PHS as a separate service in a different server. Similarly, we implemented the Permit History Service as a servlet running under the IdP server. We implemented the permit handler service as a component servlet running in the mashup server. We implemented the Permit Granting Protocol and history access protocols as described earlier. We stored the permit history in a persistent cookie under the domain of the PHS, in the user's browser.

For passing the permit requests and the issued permits, we used URLs and GET/POST parameters. Permits are stored in the browser using a session cookie (for one-time permits) or a permanent cookie (for auto-approved permits)

Figure 3. Delegate-permits page from our implementation.

under the domain of the mashup application. Details of the delegation permit format, permit cookie format, permit history cookie format, and the various redirect formats can be found in the Appendix.

The implementation also involved providing an informative user interface for the *delegate-permissions* page, and the permit history page. The design was evaluated by our in-house user experience group for usability, and found satisfactory. Figure 3 shows a screenshot of our delegate-permits page, showing a request from MyCoolMashup for two permits from two back-end services.

4 Related Work

As an emerging application, mashups have been studied by many researchers [27]. Researchers have investigated the process of creating a mashup [30], the utility of mashups for data integration in an enterprise setting [17, 26, 31], a mashup fabric for intranet applications [5], and the advantages of mashups in learning environments [29]. Novel uses of mashups include deep web search [14], service composition [24], and service oriented computing [9].

Issues associated with mashup security have also been explored by researchers [11, 22]. MashupOS is a set of operating system abstractions for ensuring security and isolation of web services inside a browser [15]. Researchers have observed the threat of man-in-the-middle attacks for mashups [19], proposed a secure components model for mashups [20], and proposed an approach for secure cross-domain communication in Web mashups [16]. None of these works addresses mashup authorization and delegation problems; their main focus is providing security against cross-site scripting attacks, an important issue that is orthogonal to the topic of our work.

A good summary of authentication and authorization infrastructures can be found in [25]. Kahan offered an early proposal for a capability based authorization model for web-based applications [18]. Chadwick et al. proposed an authorization delegation scheme based on X.509 certificates [10]. The Grid computing community has explored approaches to credential delegation for grid services. The four main approaches for credential delegation are the delegation services of the Globus Toolkit, EGEE's gLite, the Java-based Commodity Grid (CoG) Kit, and MyProxy [3, 6]. However, none of these allow users to limit the rights of the recipient of the delegated authorization, which is one of the key issues in mashup authorization and delegation. Further, approaches designed for the Grid assume that users have cryptographic credentials such as X.509 credentials, which is not the case for most mashup users.

Researchers have also looked at the problem of limiting delegated rights in general. Delegation Logic [23] includes constructs that can be used to limit the depth of delegation, i.e., sub-delegation. This does not solve the more general problem of restricted delegation of authorization rights, though this might be useful for building a hierarchy of mashups. KeyNote [8] and PolicyMaker [7] do allow further restriction of a delegee's rights. However, these are capability based systems that are oriented towards environments where users have public keys and X.509 or SPKI/SDSI credentials, whereas with mashups, the users typically authenticate themselves via usernames and passwords at the back-end services, with no public keys.

5 Conclusion

As mashups become more widespread in the Internet, ensuring a proper authorization model for mashups becomes essential. In this paper, we presented a scalable, stateless delegated authorization protocol and a practical implementation using delegation permits. Our permit based access delegation model allows users to fine-tune their release of access rights to mashups, while the authorization servers and back-end applications do not have to maintain elaborate state information. This makes our approach scalable. Our practical implementation of a proof of concept proves the usability of such a permit based approach. Delegation Permits include a number of features not yet present in OAuth, and we expect some of them may well inform the development of OAuth extensions. While we do not necessarily expect permits to be adopted by the major players willing to handle distributed state management, we do believe it will be a useful model for those interested in maintaining stateless servers, either internally or on the Internet.

References

[1] Citrix NetScaler: Web application delivery with the highest availability, security and performance. On-

line at http://www.citrix.com/English/ps2/products/product.asp?contentID=21679.

[2] Mint.com personal finance website. Online at http://www.mint.com.

[3] Web services delegation via MyProxy. Online at http://grid.ncsa.uiuc.edu/myproxy/delegation/.

[4] OAuth Specification 1.0. Online at http://oauth.net/core/1.0, 2007.

[5] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y. Ng, D. Simmen, and A. Singh. Damia: a data mashup fabric for intranet applications. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1370–1373. VLDB Endowment, 2007.

[6] J. Basney, M. Humphrey, and V. Welch. The myproxy online credential repository. *Software- Practice & Experience*, 35(9):801–816, 2005.

[7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, 1996.

[8] M. Blaze, J. Ioannidis, and A. Keromytis. Experience with the KeyNote trust management system: Applications and future directions. In *Proceedings of the 1st International Conference on Trust Management*, pages 284–300. Springer, 2003.

[9] S. Cetin, N. Altintas, H. Oguztuzun, A. Dogru, O. Tufekci, and S. Suloglu. A mashup-based strategy for migration to service-oriented computing. In *Proceedings of the IEEE International Conference on Pervasive Services*, pages 169–172, 2007.

[10] D. Chadwick, A. Otenko, and E. Ball. Role-based access control with x.509 attribute certificates. *IEEE Internet Computing*, 07(2):62–69, 2003.

[11] M. Davidson and E. Yoran. Enterprise security for web 2.0. *COMPUTER*, pages 117–119, 2007.

[12] R. Dhamija, J. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM Press New York, NY, USA, 2006.

[13] Google. Google account authentication (authsub). Online at http://code.google.com/apis/accounts/AuthForWebApps.html.

[14] T. Hornung, K. Simon, and G. Lausen. Mashing up the deep web - research in progress. In *4th International Conference on Web Information Systems and Technologies (WEBIST) 2008*, pages 58–66, Funchal, Madeira, Portugal, May 2008.

[15] J. Howell, C. Jackson, H. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[16] C. Jackson and H. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th International Conference on World Wide Web*, pages 611–620. ACM Press New York, NY, USA, 2007.

[17] A. Jhingran. Enterprise information mashups: integrating information, simply. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 3–4. VLDB Endowment, 2006.

[18] J. Kahan. A capability-based authorization model for the world-wide web. *Computer Networks and ISDN Systems*, 27(6):1055–1064, 1995.

[19] P. Karger. Mashups legitimize man-in-the-middle attacks: A position paper. In *IEEE Web 2.0 Security and Privacy Workshop*. IEEE, 2007.

[20] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW '08: Proceeding of the 17th International Conference on World Wide Web*, pages 535–544, New York, NY, USA, 2008. ACM.

[21] N. Kulathuramaiyer. Mashups: Emerging application development paradigm for a digital journal. *Journal of Universal Computer Science*, 13(4):531–542, April 2007.

[22] G. Lawton. Web 2.0 creates security challenges. *IEEE COMPUTER*, 40:13–16, 2007.

[23] N. Li, B. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[24] X. Liu, Y. Hui, W. Sun, and H. Liang. Towards service composition based on mashup. In *Proceedings of the IEEE Congress on Services*, pages 332–339, 2007.

[25] J. Lopez, R. Oppliger, and G. Pernul. Authentication and authorization infrastructures (aais): a comparative survey. *Computers & Security*, 23(7):578–590, 2004.

[26] S. K. Makki and J. Sangtani. Data mashups & their applications in enterprises. In *Third IEEE International Conference on Internet and Web Applications and Services*, pages 445–450, Athens, Greece, June 2008.

[27] D. Merrill. Mashups: The new breed of web app. *IBM Web Architecture Technical Library*, 2006.

[28] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *DIM '06: Proceedings of the second ACM workshop on Digital identity management*, pages 11–16, New York, NY, USA, 2006. ACM.

[29] C. Severance, G. Hardin, and A. Whyte. The coming functionality: mash-up in personal learning environments. *Interactive Learning Environments*, 16(1):47–62, 2008.

[30] N. Zang, M. Rosson, and V. Nasser. Mashups: who? what? why? In *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, pages 3171–3176, New York, NY, USA, 2008. ACM.

[31] J. Zou and C. Pavlovski. Towards accountable enterprise mashup services. In *Proceedings of the IEEE International Conference on e-Business Engineering*, pages 205–212. IEEE Computer Society Washington, DC, USA, 2007.

Appendix

This appendix contains details of our implementation of the permit-based delegated authorization system.

Format of Delegation Permits

The format of the Delegation Permit is presented in Table 1, and an example is shown in Table 2. Each field in a delegation permit (DP) is utf8 text; binary objects are base64-encoded as noted. Fields are concatenated together into the DP, demarcated by "|". The DP data structure is intended to allow for optional fields and does not enforce field ordering (except for the prefix, which must come first). The initial optional fields that the PGS currently recognizes are included below, but this does not represent an exhaustive list of what may eventually be included. At this time the PGS will simply ignore fields that it does not understand and it is assumed that initial client libraries will do the same

Permit Grant Service (PGS)

The default behavior of the PGS is be to prompt for authorization every time the mashup requests it. Decisions will be remembered by storing a cookie in the PGS's domain, where the cookie will store the information shown in Table 3 (similarly to how that information is stored in the permit). An example permit cookie is shown in Table 4.

When the PGS receives a request, it will inspect any presented state cookies to determine if the currently requested permissions are a subset of those stored in the cookie. If so, the user will not be re-prompted. If not, then the user will be shown the delegate-permissions screen with indications of which permissions are currently granted and what has changed, with the ability to grant or deny changes.

A set of permissions is a subset of what is stored in the cookie only if:

- The user ID matches.
- The service matches.
- The destination URL in the request is a valid URL for the specified service.
- The time since the last interactive authorization is not too long (e.g., on the order of a month).
- Each resource in the request is in the cookie, paired with the same permit descriptor.

Permit History Service (PHS)

The PHS can be used by the user to view what permits the user has granted to various applications in the past. When the user goes to the Permit History Service, a list of application servers, and the group of permits granted to each

Param	Description
Permit prefix	This is a freeform text field that represents the permit's protocol variant. This document describes prefix "permit_v1", but it is assumed other permits will be similar
uid	User ID for whom this permit was created
g	Requested LDAP groups for which the user's membership is verified, delimited by ","
ng	Requested LDAP groups for which the user is *not* a member, delimited by ","
s	The service for which this permit was created. This will take the form of a DNS name plus an HTTP URI prefix denoting what subsection of that site the ticket applies to. For example: "abc.acme.com/" applies to the entire abc.acme.com web site, "www.acme.com/eng" applies to any URI on www.acme.com that begins with "/eng", "foobar.com:9999/" applies to any URI on foobar.com using the non-standard port 9999
ds	Delegated service label. A human-readable label representing the service which delegated access was requested
pd	Permit descriptor. A human-readable label representing some level of authorization (for example "MyBugTracker Read-Only Access" or "MyProjectDB Read Self Access")
lt	Login time. The time that the user's credentials were verified (GMT formatted "YYYYMMDDhhmmss", e.g. "20040308162144")
pt	Permit issue time. The time at which the permit was actually issued to the user (GMT formatted "YYYYMMDDhhmmss")
at	Permit approval time. The time at which the user last saw the delegate-permissions screen and actually authorized this permit (GMT formatted "YYYYMMDDhhmmss")
sig	A digital signature on the "|"-delimited concatenation of the other permit fields ordered as they appear in the permit (with no pipe at the beginning or end). The signature consists of three fields separated by "|": • alg – the signature algorithm (typically 'DSA') • kid – the key identifier (as passed in the request) • sig – the base-64-encoded SHA1withDSA digital signature

Table 1. Details of delegation permit format

```
permit_v1|uid=testuser|g=eng|ng=hip|
s=mycoolapp.com/|ds=MyCoolApp|pd=
MyBugTrackerRead-Only|lt=2007022912000|
pt=2007022912000|at=2007022912000|alg=
DSA|kid=FOO|sig=B64ENCODEDDSASIG|
```

Table 2. Example delegation permit

Parameter	Description
UserID	User for whom this permit was created.
service	The service for which this permit was created.
authTime	The last time the user actually authorized these permissions. In format "YYYYMMDDhhmmss", e.g. "20040308162144".
PermitReq	For each permit request, this field records the resource for which the permit is requested, and the permit descriptor.
sig	Signature for this cookie.

Table 3. Format of permit cookies

server, is displayed. The user can choose to delete any of the permission groups. Permit grant history is maintained in a cookie named "$SSO\_PERMIT\_HISTORY$". When the user clicks on delete for a permit group, the corresponding permit group is removed from the cookie. The PHS also redirects to the permit handler service of the corresponding mashup server and requests it to delete the selected permits from its permit cookie.

Redirects to the Permit Granting Service

When an application requires a permit to access another service on behalf of the user, it communicates that need by redirecting the client browser to the PGS along with its requirements. The parameters provided to the PGS as HTTP GET-encoded parameters are shown in Table 5:

Redirects to the Mashup's Permit Handler

Every relying application server is required to provide the following support at its permit handler URL, which is the name of its service as provided to the PGS + "permithandler", such as "`https://mycoolapp.com/app/permithandler`". The URL will receive requests as GET or as POST with the parameters given in Table 6.

Typically the mashup's permit handler will:

1. Parse the request to retrieve the permits (fails on parse errors).
2. Ensure that the permit prefixes all match what was expected.
3. Validate the signatures on the permits.

```
|alice|mycoolapp.com/app|https:
//mycoolapp.com/app/start.html|
MyBugTracker:MyBugTrackerRead-OnlyAccess|
MyProjectDB:MyProjectDBRead-OnlyAccess|
SIGNATURE|
```

Table 4. Example permit cookie

Param	Required?	Description
v	required	The prefix indicating the version/purpose of the permit
s	required	The name of the service issuing the permit request
d	required	The destination URL to which the user should be redirected after the permission granting process is complete
g	optional, repeated	Each parameter specifies a group for which the application wants to know the user's membership, using the same "cn" or "ou/cn" encoding as used in permits. If not present, no group info is returned.
		foreach i = 1 ... n (where n is the number of requested permits)
pi_res	required	The name of the resource for which the request is seeking a permit
pi_desc	required	The requested permit descriptor
k	required	The first three characters of the SHA-1 hash of the public key corresponding to the requested private signature key

Table 5. Redirect format

4. Ensure that the username matches across all permits and with the login ticket.
5. Copy the validated permits into domain-restricted session cookies.
6. Issue a redirect to the destination page.

Param	Description
p	A new permit for this server to use when making back-end requests (there can be one or more p parameters, each representing a different permit)
d	The URL to redirect to after processing the permits.

Table 6. GET/POST Parameters for Redirects to Mashup Permit Handlers