Privacy-aware Identity Management for Client-side Mashup Applications [*]

Saman Zarandioon Danfeng Yao Vinod Ganapathy
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{samanz, danfeng, vinodg}@cs.rutgers.edu

ABSTRACT

This paper concerns the problem of identity management in modern Web-2.0-based mashup applications. Identity management supports convenient access to information when mashups are used in sensitive environments, such an banking, investment and online shopping, by providing services such as single sign-on.

We present Web2ID, a new identity management protocol tailored for mashup applications. Web2ID leverages a secure mashup framework and enables transfer of credentials between a service provider and a consumer. We also describe a new relay framework in which communication between two service providers is mediated by a relay agent within the mashup. We show that Web2ID is privacy-preserving and prevents service providers from learning a user's surfing habits.

We present an implementation of Web2ID and the relay framework using a JavaScript-based library that executes within the browser. Our implementation does not require client-side changes and is therefore fully compatible even with legacy browsers. We also highlight the key challenges faced in creating a portable, in-browser library to support identity management in mashups.

Categories and Subject Descriptors: D.4.6[Operating Systems]: Security and Protection - Authentication K.6.5[Management of Computing and Information Systems]: Security and Protection

General Terms: Security, Design, Human Factors

Keywords: Mashup, Security, Communication, AJAX, Web, OMOS.

1. INTRODUCTION

Mashup applications integrate information from multiple autonomous data sources within the Web browser. For example, iGoogle allows users to create a personal page containing "gadgets" from multiple Web domains, such as NYTimes, Weather.com and Google Maps. As such, mashups have gained in popularity because they provide a seamless browsing experience.

[*]This work has been supported in part by NSF grant CCF-0728937, CNS-0831186, and the Rutgers University Computing Coordination Council Pervasive Computing Initiative Grant

Despite their popularity, mashups are still not in widespread use for sensitive Web applications, such as banking, investment, online shopping and bill payment. Such mashup applications currently require user authentication to prevent unauthorized access to sensitive information. A Web user who includes such sensitive applications in a mashup must authenticate herself individually to each of these applications. For example, `mint.com` and `yodlee.com` allow users to view a summary of their financial activities by accessing back-end services such as banks and credit card companies. However, they require the user to authenticate with each of these services and grant the mashup service provider access to her private financial information. Studies show that users who manage multiple Web identities often use weak passwords, or write them down to alleviate the burden of having to memorize their passwords for each domain—both of which can potentially compromise security.

Existing techniques to ease user authentication leverage federated identity management solutions, such as *Single Sign-On* (SSO). In conventional SSO, a user authenticates herself to an *identity provider (IdP)* website. The IdP allows the user to sign into other participating *service provider (SP)*[1] websites without requiring her to authenticate again. For example, suppose that Alice authenticates using a secure logon session with an identity provider `IdP.com`. She may later wish to access services provided at `CarRental.com` and `Airline.com`. Instead of requiring Alice to authenticate herself again to `CarRental.com` and `Airline.com` (the SPs), `IdP.com` could assert Alice's identity to these websites. `IdP.com`, which acts as the IdP in this case, can provide Alice's user name and other attributes that verify her identity to `CarRental.com` and `Airline.com`. Note that in this case, because of direct communication between the IdP and SPs, `IdP.com` may learn that Alice has accounts at `CarRental.com` and `Airline.com` and her usage pattern. Moreover, `CarRental.com` may query user's flight schedule (or other attributes) from `Airline.com`. However, using any conventional communication over network (e.g. HTTP) enables `Airline.com` to learn which car rental service is being used by the user.

Unfortunately, two key difficulties prevent existing identity management protocols from being directly applicable to modern Web-2.0-based mashup environments. First, these protocols implement SSO via a series of HTTP redirections within the user's browser. These redirections perform inter-domain communication between the IdP and SP and transmit the user's credentials from the IdP to the SP. However, redirections are ill-suited for stateful AJAX-based applications, such as Web desktops and Web-based office applications, because they involve unloading/reloading the application

[1]Identity providers have also been known as authorities or asserting parties. Service providers have also been known as replying parties.

upon each redirection. Without application-level support, unloading/reloading operations will result in the loss of unsaved data. Second, the identity provider used to manage users' credentials and personal information causes privacy concerns. It can learn a lot about a user's online activities because IdPs are involved whenever a user authenticates herself to a service provider.

There have been several existing identity management work addressing IdP-related privacy concerns by anonymizing user credentials, defining privacy policies, or hiding a user's digital trail [4, 7, 11, 21, 23]. There are two main differences between these privacy-aware identity management solutions and our work. First, we study identity management in the client-side mashup environment through a secure and efficient in-browser framework. Second, we propose a novel single sign-on protocol *without* requiring the participation of a trusted party identity provider.

We present *Web2ID*, a new protocol for identity management (and SSO) in mashup applications. By leveraging a secure mashup framework, Web2ID enables transfer of credentials without requiring webpage redirections, and works seamlessly with AJAX-based Web-2.0 applications. We also describe a new *relay mashup framework*, based on which a trusted, client-side *relay application* can be built in the mashup to transmit credentials between providers. Our Web2ID protocol offers the following main benefits:

- **Privacy.** We protect user privacy by eliminating the requirement of a trusted party identity provider in online single sign-on process. To realize this property, we utilize public-key cryptography, in our authentication protocol. Moreover, we provide a client-side communication framework called *mashlet relay* which enables a service provider to send a query to another service provider without revealing its identity. A mashlet is a HTML page hosted by an iframe in the browser (described in Section 2). The mashlet-relay framework protect user privacy in mashup environments because service providers that host user's data cannot learn how user consumes her data. This feature is especially important when the user wants to provide her identity attributes (certified by a trusted party) to another service provider.

- **Compatibility with legacy browsers and modern AJAX oriented mashups.** Our framework is implemented as a JavaScript library that is incorporated with mashup applications. It does not require browser modifications or specialized plugins to operate and is fully portable across browsers and execution platforms. We illustrate the portability of our framework by incorporating it with several popular browsers, including Firefox, Opera, Apple Safari, IE and Google Chrome. Moreover, we avoid using HTTP redirections for communication; consequently, our protocol is compatible with modern AJAX-based Web applications.

  Our implementation of the relay framework required in-browser support for both symmetric and asymmetric cryptography in the form of a JavaScript library. We highlight the key difficulties in creating such a library for a legacy browser and also consider this library as an independent contribution of this paper.

2. BACKGROUND AND DEFINITIONS

In this section, we present background material on mashup frameworks and discuss the problems addressed by our identity management protocol.

2.1 Mashups and Mashlets

Mashup applications aggregate content from a number of providers and display them within Web browsers. Such applications can be designed either as *server-side* mashups or *client-side* mashups. In server-side mashups, a proxy (called the mashup server) aggregates content from multiple sources. The Web browser loads the mashup application by visiting a URL corresponding to the proxy. For example, Facebook applications use the RESTful API provided by Facebook to query user's social information and aggregate them with other data to provide their users with some interesting social oriented services. In contrast, client-side mashups directly aggregate content within the Web browser. Several frameworks have recently been proposed to support safe yet expressive client-side mashups [3, 10, 13, 15, 25, 27].

The client-side components of a mashup application are called *mashlet*. Mashlets represent the service provider that is hosting them in the client side and run in the browser with the privilege of domain name of their host. To be more concrete, a mashlet is simply a HTML page which loads to an iframe and contains some JavaScript code that enables it to communicate with other mashlets in the page. A *mashup application* is a Web application that aggregates a number of mashlets, possibly from different sources on the Web. We also use the term *mashlet container* to refer to the mashup application.

A number of recently-proposed frameworks allow mashlets to securely communicate with other mashlets executing in a mashup application [3, 13, 15, 25, 27]. A secure inter-mashlet communication protocol is a protocol that guarantees mutual authentication, data confidentiality, and message integrity of mashlet. *Mutual authentication* in inter-mashlet communication means that two mashlets that are communicating with each other must be able to verify each other's domain name. *Message integrity* requirement means that any tampering of the messages between two mashlets should be detected/prevented. *Data confidentiality* here means a mashlet should not be able to listen to the communication between two other mashlets running under different domains.

For concreteness, the rest of this paper describes mashups and mashlets in the context of OpenMashupOS (OMOS) [20, 27], a secure client-side mashup framework. The concepts developed in this paper are applicable to any client-side mashup framework.

2.2 Identity Management in Mashup Applications

In the following discussion, we consider three problems in identity management and discuss how the *Web2ID* protocol and our mashup relay framework address each of these problems. Where applicable, we also discuss why existing techniques fail to address the problem.

User Authentication.

When sensitive Web applications, such as those for banking, investment and tax services, are integrated into a mashup environment, it is highly desirable to use an authentication protocol that provides single sign-on (SSO). SSO enables these service providers (i.e., the bank or the investment company) to authenticate the user without requiring her to prove her identity separately to each provider. The goal of an authentication protocol in a mashup environment is for the user to prove the ownership of an identity to a service provider without revealing any information that can be misused by a malicious service provider to impersonate the user.

Most existing identity management protocols for Web, including OpenID [1], use a unique URL to represent the identity of a principal. The advantage of using a URL as opposed to a name or email address is that a URL is tangible, clickable, user-friendly, and can contain information that facilitates the authentication process. This URL is called the principal's *identity URL* . The static page that is located at identity URL is called the *identity page*. The server that hosts the identity page is called the *identity host*. Therefore, during the authentication, user claims ownership of an identity URL and proves her claim to a service provider by following the corresponding authentication protocol. However, all these authentication protocols require a trusted third party called *Identity Provider (IdP)* to validate user's claim. Users first create an account with IdP and then in identity page delegate the authentication of their identity URL to that IdP. But this violates user's privacy as IdP can learn surfing habits of the user.

Web2ID uses asymmetric cryptography to enable users to prove ownership of their identity URL without relaying on any services by third parties. In *Web2ID* , users are represented by a mashlet hosted at their identity URL, in much the same way that service providers are represented on the client-side by their mashlets. We call the mashlet that is hosted at the identity URL of a user *identity mashlet*. Basically, in *Web2ID* , identity page is a mashlet (i.e. includes JavaScript libraries required for communication and providing authentication service in the client-side).

During the authentication protocol, the user first presents her identity credentials to her identity mashlet. In turn, the identity mashlet acts on behalf of the user and interacts with other mashlets to prove that the user owns the identity that corresponds to its URL. The identity mashlet enables other desirable features including authorization delegation and attribute exchange. We define both problems below.

Attribute Exchange.

An important feature supported by most identity management frameworks is that of *attribute exchange*, in which one service provider requests a user's identity attributes (or preferences) from another service provider. Attribute exchange is especially important for mashup applications, in which interaction between mashlets is the norm. We refer to a service provider that requests user's attributes as an *attribute requester* and the service provider that stores user attributes and settings as an *attribute provider* (also called a wallet [22]). An attribute provider may optionally certify user attributes (e.g., for attribute-based authorization) or simply send non-certified values (e.g., for providing settings and preferences).

An identity attribute exchange protocol should ideally accommodate three privacy requirements:

- **Requirement 1:** An attribute provider should share a user's attributes only upon explicit consent from the user.
- **Requirement 2:** An attribute requester should be able to query a user's attributes without necessarily knowing the identity of the user.
- **Requirement 3:** The protocol should be able to anonymize an attribute requester to prevent an attribute provider from learning identity of the requester (and thereby, the user's Web surfing habits).

Designing a browser-based protocol that can satisfy all these requirements is challenging. Existing browser-based attribute exchange protocols use a series of HTTP redirections to keep users in the loop to acquire their consent without revealing their identity to the requester (Requirement 1). However, in such protocols, the requester must send a callback URL to the provider; as a result, HTTP redirection-based communication discloses the identity of the requesters, thereby violating Requirement 3. State of the art techniques to remove the need for communication between the attribute requester and provider use sophisticated cryptographic techniques (e.g., idemix [6]). However, these solutions are not currently suitable for practical use in browser-based protocols [22]. Web2ID uses our proposed mashlet relay framework to anonymize the attribute requester.

Authorization Delegation.

Web application mashlets included in a mashup typically access resources hosted at other domains. In this context, the mashlet that accesses resources is typically called the *Consumer*, while the domain that hosts the resource is called the *Service Provider*. Consumers should not be able to access a user's protected resources unless the user grants them the required access permission.

An *authorization delegation protocol* allows a user to delegate permissions to a consumer to access her resources hosted at a service provider. For example, a user may be able to delegate permissions needed to access her files on a photo-sharing website (the service provider) to a website that provides photo editing utilities (the consumer). An authorization delegation protocol should be privacy-preserving in that it must not reveal the user's identity. In the example above, for instance, the user may wish to grant the photo editing service read access to her photos hosted on the photo sharing website without revealing her identity to the photo editing service.

3. THREAT MODEL

In this section, we present the threat model for Web2ID.

Users may be malicious.

As is standard with AJAX-based applications, some messages of the Web2ID protocol are exchanged on the client-side, within the user's browser via inter-mashlet communication. Because the user has complete control over the browser, a malicious user may alter the client-side component of the Web2ID protocol, for example, by forging the identity of another user or providing forged identity attributes to an attribute requester. Consequently, for transactions in which the user must not be trusted, the correctness and integrity of the Web2ID protocol must not rely on the client-side portion of the protocol executing correctly. Web2ID uses cryptographic techniques to ensure the integrity of data that passes through the client.

Service providers may be malicious.

When a service provider authenticates a user, it must receive certain information that enables it to ensure the authenticity of the user. A malicious service provider may misuse this information to impersonate the user to a second service provider using a *relay attack*. For example, a malicious service provider `attacker.com` that authenticates Alice may use her credentials to impersonate her to another service provider `honest.com`. In this attack, `attacker.com` tries to log into `honest.com` claiming the ownership of Alice's identity URL (e.g., `alice.me`). When `honest.com` challenges `attacker.com`, it relays that challenge to Alice when she tries to prove her identity to `attacker.com`. In turn, `attacker.com` uses this information to convince `honest.com` of Alice's identity.

In attribute exchange, a malicious attribute provider may try to violate a user's privacy by learning the identity of requesters that try to obtain the user's attributes. As a result, the attribute provider may learn the user's surfing habits. Similarly, a malicious attribute requester may also try to learn the user's identity or attributes without her agreement.

Finally, in the authorization delegation protocol, a malicious consumer may try to convince a service provider to give it access to a user's protected resources without possessing appropriate authorization (i.e., explicit consent from the user). In the case where the user wishes to protect her privacy from the consumer, a malicious consumer may try to learn the user's identity during the course of authorization.

Man-in-the-Middle (MitM) attacks.

Based on their capabilities, man-in-the-middle attackers (MitM) can be either active or passive. A passive MitM attacker only listens to the conversation between two parties in the protocol. The goal of a passive attacker is to obtain information that can be used to impersonate the user, get unauthorized access to her private resources or violate her privacy. In contrast, an active attacker can also modify the content of conversation. An active MitM may try to change the result of an authentication or authorization check by modifying data transmitted in the protocol. MitMs can also be classified based upon their location in the network. Client-side MitMs involve a malicious mashlet that tries to spoof mashlet-to-mashlet communication in the protocol. A network MitM spoofs network communication, such as those between a mashlet and its server, or between two servers. In Web2ID protocol, we assume that the point to point network communications are safe against active MitM attacks, which can be guaranteed by using secure lower level protocols like SSL. Finally, malicious mashlets may also try to subvert the protocol by launching frame phishing attacks against the user [15].

4. BASIC WEB2ID PROTOCOL

The basic Web2ID protocol enables users to prove their identity to a service provider website without the use of a trusted third party. This enables users to independently prove their identities and prevent any third party from learning their surfing habits. The Web2ID protocol achieves this goal using public-key cryptographic primitives in a manner akin to public-key client authentication in SSH (RFC 4252 [2]). Suppose that a principal P (e.g., Alice) wishes to adopt an identity I (e.g., an identity URL, such as `alice.me`) and prove her ownership of that URL to a service provider `SP.com`. We explain below how identity adoption and authentication work in Web2ID.

Identity Adoption.

To adopt an identity URL I, say `alice.me`, Alice first hosts an identity mashlet at this URL. The identity mashlet is a component that is trusted by Alice and represents her within a mashup application. To configure her identity mashlet, Alice must navigate to her identity mashlet using a browser. When the identity mashlet loads for the first time, it detects that it is not configured, and generates a public/private-key pair ($Pu(I)$,$Pr(I)$). The public key is embedded within the identity mashlet, while the private key must be stored safely by Alice.

Authentication.

When a user such as Alice attempts to authenticate herself with a service provider, she claims the ownership of an identity URL, such as `alice.me`. In turn, the service provider sends a session token encrypted under the public key associated with the identity URL `alice.me`. When the user then sends requests to access resources, she must prove ownership of the session token corresponding to her claimed identity. Figure 1 illustrates how service provider `SP.com` assigns a session token to the user who claims the ownership of identity `alice.me`. As this Figure illustrates, authentication happens in seven steps, as described below:

1. The user claims to own an identity I. For instance, this identity could be an identity URL `alice.me`. This claim can be communicated to the mashlet of the service provider `SP.com`. For example, the user may enter the URL in a form provided by `SP.com`.

2. The service provider's mashlet sends the claimed ID I to the service provider and, if not already loaded, loads the identity mashlet located at the claimed identity URL (i.e. `alice.me`).

3. The service provider extracts the public key $Pu(I)$ and the type/version of the corresponding public-key encryption algorithm Alg from the claimed identity page.

4. The service provider first generates a session token χ, and encrypts χ and the domain name of its mashlet `SP.com` with the public key $Pu(I)$. It then sends the result $\Delta = E_{Pu(I)}(\chi,$ `SP.com`$)$ back to the service provider's mashlet as response. Note that the domain name of the service provider must be included in Δ to protect users against relay attacks by malicious service provider (see Section 3). In practice, since the length of identity URL is unbound and may be longer than the public key, a two step encryption needs to be used. For the sake of clarity, these details are omitted.

5. The service provider's mashlet sends $\Delta = E_{Pu(I)}(\chi,$ `SP.com`$)$ to the identity mashlet for decryption.

6. If the identity mashlet does not already have the private key $Pr(I)$, it asks the user to provide her login credentials. Using the user's login credentials identity mashlet computes the private key. For example, the user can load the encrypted value of her private key from a USB memory stick and provide a passphrase that can be used by the mashlet to compute the private key. Alternatively, the user may enter the private key directly by swiping her smart card that contains her private key.

   Once the identity mashlet has the private key and user permits the authentication, the identity mashlet decrypts Δ and verifies that the domain name of the service provider (`SP.com`) matches the domain name in the token.

7. The identity mashlet sends the computed session token χ back to the service provider mashlet.

In our implementation, inter-mashlet communication is facilitated by the OMOS framework which provides mutual authentication, and confidentiality and integrity guarantees for the data exchanged between two mashlets. This ensures that a malicious mashlet in the mashup application will not be able to compromise communication between the identity mashlet and the service provider's mashlet.

Upon the completion of the above protocol, the service provider can verify that the value of the session token received from the identity I is valid. This proves the user's claim of ownership of I to `SP.com`. Existing identity management protocols prove the possession of the session token by including it with each request, and are therefore vulnerable to session hijacking via MitM attacks.

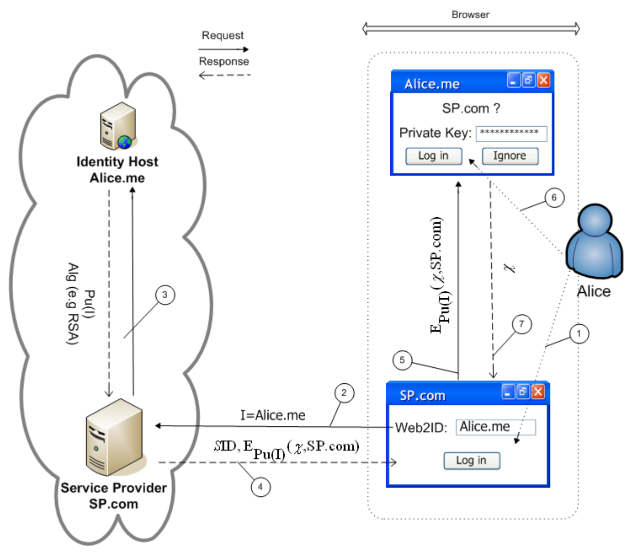

Figure 1: An identity mashlet represents the user within the application. The user can prove ownership of the identity mashlet by proving the possession of the private key that corresponds to the public key located at URL of the identity mashlet.

Our implementation of Web2ID uses a MAC (Message Authentication Code) to prove possession of the session token. [2] In this approach, MAC value of each XMLHTTRequest request is computed using the session token and is included in every request. The service provider serves a request only if the included MAC value is correct. Note that during the above protocol does not require the service provider to keep any protocol-specific state, thereby ensuring a stateless implementation of the web application at the service provider. In addition, the user's credentials are never transmitted over the network; instead such communication happens on the client-side, where communication is secured using OMOS.

The Web2ID authentication protocol can also be used by a service provider to prove the ownership of its mashlet. We use this feature as part of authorization delegation protocol that we describe next. The authorization delegation and attribute exchange protocols build upon the authentication protocol described above.

The above basic Web2ID protocol supports user authentication. It can be generalized to support more complex operations such as identity attribute exchange and authorization delegation. In Section 5, we will present a mashup relay framework and explain how it facilitates attribute exchange in Web2ID. Our authorization delegation protocol is described in Section 6.

Security Analysis of User Authentication Protocol.

Because *Web2ID* uses client-side inter-mashlet communication, its security relies on the confidentiality of mashlet and the security of the client-side communication protocol that is used in its implementation. We assume that the mashlet framework that is used for implementation of *Web2ID* guarantees confidentiality of mashlets and security of their communication. This assumption implies that the mashlet framework protects the protocol against MitM attacks by malicious mashlets. Next, we analyze how the user authentication protocol resists against attacks launch by adversaries.

[2]To do so, we ported the necessary cryptographic functions HMAC-SHA1 and HMAC-SHA256 (RFC2104 [16], RFC3174 [9]) into the OMOS framework.

Since the session token χ is encrypted by the public key that is associated with the claimed identity URL (located at the identity page), the user can get access to the session token only if she owns the corresponding private key. Therefore, assuming that only the owner of an identity URL has access to the private key that corresponds to the public key embedded in the corresponding identity page, she will be the only person that can use that session token. This prevents malicious users from forging identities that does not belong to them.

To protect users against replay attacks, *Web2ID* requires service providers to encrypt the domain name of their mashlet besides the session token. This way the identity mashlet can verify and make sure that the mashlet that is requesting the session token is not replaying an encrypted session token issued by another service provider. Finally, since user's credentials and session tokens are never sent over network in clear text, *Web2ID* authentication is immune to passive MitM attacks. As discussed earlier, Web2ID relies on the underlying communication protocol (e.g HTTPS) to protect users against active MitM network attackers.

5. RELAY MASHLET AND ATTRIBUTE EXCHANGE IN WEB2ID

In this section, we introduce a new mashlet relay framework which is a simple-yet-general mashup application that enables user-centric client-side communications between two domains. Then we explain why such a mashlet relay framework is useful in the implementation of identity attribute exchange in Web2ID.

5.1 Mashlet Relay Framework

We define *mashlet relay framework* as a special client-side mashup framework with three mashlets within a browser environment where the communication of two mashlets, each hosting contents of a remote server, is indirect and realized through a third mashlet that is hosted by the localhost. We refer to the two mashlets hosted by remote servers as *server mashlets*. A server mashlet also communicates to its corresponding remote server via the mashlet-to-server communication mechanism. We refer to the mashlet that bridges the communication of the two server mashlets as the *relay mashlet*. All inter-mashlet communication follows the mashlet-to-mashlet messaging mechanism. The relay mashlet effectively passes messages between two server mashlets and is able to modify the messages based on user's inputs. Figure 2 shows a schematic drawing of such a mashlet relay framework, where the mashlet in the middle (Mediator) mediates the communication between a requester (e.g., SP.com) and a provider (e.g., AttProvider.com). The mediator mashlet is launched by the localhost of the individual user. It anonymizes the identity of the requester (e.g., SP.com), as the provider (e.g., AttProvider.com) learns nothing about who issues the request. Such a mashlet relay framework, although simple, possesses two important properties. First, it supports a user-centric design where the user is able to monitor and actively control the messages being communicated among server mashlets. Second, the client-side relay mashlet eliminates the need of direct communication between the two server mashlets; this feature plays a key role in enabling privacy-aware identity management in Web2ID.

This mashup-based relay framework naturally facilitates the construction of a privacy-aware identity management protocol, namely identity attribute exchange in SSO, that enables the exchange of user's identity credentials without the direct communication between the identity provider and service provider. In existing (fed-

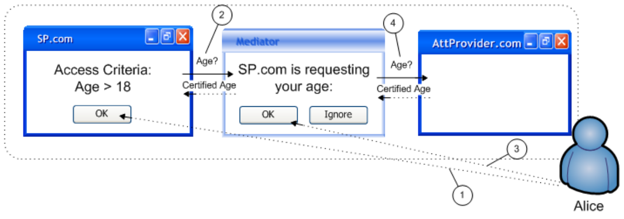

Figure 2: Web2ID users mashlet relay communication framework for attribute exchange. In mashlet relay framework, a mashlet (center) mediates the communication between requester (left) and provider (right) and anonymizes the identity of requester.

erated) identity management systems, *direct communications* between providers on user's ID information are typically required, which, however, is undesirable as providers may learn about the user beyond necessary. Therefore, the segregation of providers in their communication protects user privacy and prevents providers from colluding to discover user activities. Yet, in the meantime, proper message exchanges among providers should be allowed, e.g., a service provider may need to verify Alice's identity attributes hosted by an identity provider. Next, we explain why such a mashlet relay framework is useful in the identity attribute exchange in Web2ID.

5.2 Identity Attribute Exchange

When a service provider requests a user's identity attributes from another service provider, the user may wish to anonymize the identity of the provider requesting these attributes. Doing so prevents the attribute providing service from learning the user's surfing habits. To implement privacy-aware identity attribute exchange, Web2ID avails of the mashup relay framework. In particular, the relay mashlet mediates the exchange of identity attributes between service providers. Because the relay mashlet forwards the request to the attribute provider only after obtaining the user's consent, users have full control over what attributes can be exchanged.

Figure 2 presents an example that shows how using *Web2ID* a service provider `SP.com` can query user's age certified by `AttProvider.com`. If the attribute requester already knows the user's identity, the identity mashlet of the user can itself be used as a relay mashlet. Alternatively, a mashlet loaded from a trusted third party or the local machine can act as the relay mashlet. We omit the security definition and analysis for our identity attribute exchange protocol, as they can be easily deduced following the analysis in the basic Web2ID protocol.

6. WEB2ID EXTENSION: REALIZING AUTHORIZATION DELEGATION

We defined in Section 2 the problem of authorization delegation in web single sign-on. We can realize authorization delegation as a natural generalization of the user authentication procedure in basic Web2ID protocol. The details are presented in this section.

A user may wish to delegate to a consumer the rights to access her resources hosted on a service provider. There are two cases that arise in the implementation of authorization delegation, based upon the privacy guarantees that the user requires.

Case 1: Protecting user identity from consumer.

In the first case, the user may not want to disclose her identity to the consumer. For example, a user Alice may wish to print her photos hosted at a photo sharing website `sp.com` by allowing a printing website `consumer.com` to access her photos at `sp.com`. Yet, she may not wish disclose her identity (i.e. `alice.me`) to `consumer.com`. To support this case, the authorization delegation protocol should not give any information to the consumer that reveals her identity.

Figure 3 illustrates the authorization delegation protocol, via which `consumer.com` acquires an opaque token AC to access Alice's resource (e.g., `/a/v.jpg`) without learning her identity I (e.g., `alice.me`). As this figure illustrates, the service provider `sp.com` uses a secret key SK, known only to the service provider, to generate an opaque token AC = E_{SK}(`consumer.com`, GET, `/a/v.jpg`, I) that grants `consumer.com` read access (i.e., a GET request) to the resource `/a/v.jpg`, which belongs to I.

When the service provider `sp.com` receives a request from `consumer.com` (via back-end server-to-server communication) containing the access token AC, it first decrypts AC and ensures that that the identity of the requester matches the principal that the token is granted to (`consumer.com`); if so, it allows the request.

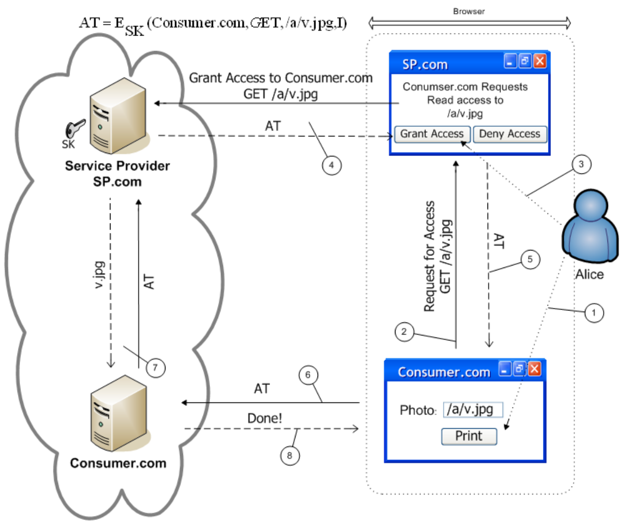
AT = E_SK (Consumer.com,GET,/a/v.jpg,I)
Browser
SP.com
Conumser.com Requests Read access to /a/v.jpg
Grant Access
Deny Access
Grant Access to Consumer.com GET /a/v.jpg
AT
SK
Service Provider SP.com
v.jpg
AT
Request for Access GET /a/v.jpg
AT
Alice
Consumer.com
Photo: /a/v.jpg
Print
AT
Done!
Consumer.com
1
2
3
4
5
6
7
8

Figure 3: In Web2ID, a service provider can issue an opaque token to a consumer to access user's resources. In doing so, Web2ID does not reveal the user's identity to the consumer.

Case 2: User identity known to consumer.

In this case, the consumer already knows the user's identity (e.g., because the user has authenticated herself to the consumer). Figure 4 illustrates the protocol used in this case. The user's identity mashlet can independently issue an access delegation certificate using the user's private key to grant the consumer access to her protected resources hosted on a service provider. In turn, the service provider can validate the certificate using the user's public key. The service provider can obtain the public key using the identity URL of the user that the resource belongs to.

The Web2ID authorization delegation protocol does not require the consumer to pre-register with the service provider. This property is in sharp contrast to similar protocols, such as OAuth and SubAuth, which require the consumer to pre-register with the service provider. Additionally, Web2ID does not require the service provider or the consumer to maintain protocol-related state during delegation, therefore it is scalable and easy to implement.

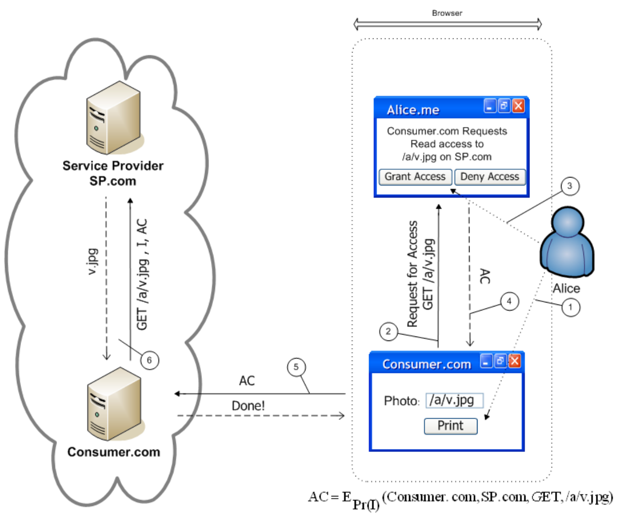

Figure 4: The identity mashlet issues a delegation certificate for read access to resource `/a/v.jpg`. Using this certificate the consumer can access `/a/v.jpg` on `sp.com`.

Security Analysis.

Before serving a request, service providers verify that the access tokens are either issued using their own secret keys or the private key of the owner of the resource. Since these types of tokens can be issued only with user's consent, consumers will not be able to access users resources without agreement of their owner.

Moreover, to prevent MitMs from using hijacked access tokens, *Web2ID* requires that all access tokens be bound to the domain name of the mashlet that the token is granted to; therefore, these tokens can be used only by the service provider that owns the mashlet. Service providers can use Web2ID authentication to prove ownership of the mashlet that the token is issued for.

In access tokens issued by service provider, the identity URL of the user is encrypted by service provider's secret key. Therefore, the consumer will not be able to learn the identity of user and this protects the privacy of the user.

7. IMPLEMENTATION AND EVALUATION

As described in Section 4, realizing Web2ID requires in-browser symmetric and asymmetric cryptographic primitives. However, we could not find any JavaScript cryptographic libraries that provide all the operations that are required for implementation of Web2ID (i.e., HMAC, asymmetric encryption and public/private key generation). The only JavaScript-based library that implemented asymmetric cryptography [24] did not support asymmetric key generation, which is required by Web2ID.

Therefore, we developed a JavaScript-based cryptographic library that not only supports operations that are required by Web2ID but also can be easily extended to support other cryptographic operations. Our library is fully compatible with commodity browsers, such as IE, Firefox, Chrome, Opera and Safari, and does not require any browser modifications.

7.1 Implementation Details

We based our implementation of the JavaScript cryptographic library on the Java Cryptography Architecture (JCA) [14, 19], an open-source Java-based cryptographic toolkit. We used Google Web Toolkit (GWT) to translate code from Java to JavaScript. However, in implementing this library and porting it to commodity browser platforms, we encountered three key challenges, namely *performance*, *browser interference*, and *code complexity*, that we describe below.

Performance.

Directly compiling the JCA library into JavaScript resulted in extremely poor performance of cryptographic operations. We found that the main performance bottlenecks were BigInteger operations, such as `modInverse`, `mod`, and `multiplication` operations, that are frequently used in cryptographic operations. We addressed this problem by replacing the JCA implementation of BigInteger with the native JavaScript code using the JavaScript Native Interface (JSNI). This replacement significantly improved the performance, with encryption and decryption operations consuming less than a second (see also Section 7.2).

Browser Interference.

Recall that the implementation of the Web2ID protocol requires generation of public/private key pairs when the identity mashlet is first loaded. We observed that key generation algorithms for asymmetric cryptographic algorithms such as RSA were quite expensive. Because most browsers (and JavaScript interpreters) are single-threaded, users cannot interact with the browser during key generation. Most browsers time out JavaScript functions that execute for long durations of time (typically about 10 seconds). As a result, key generation algorithms are interrupted by the browser.

To overcome browser interference during our key generation operations and keep the browser responsive, we used an incremental and deferred command technique. We observed that the most expensive operation during the generation of public/private RSA key pairs was the generation of probable prime numbers p and q. The `BigInteger.getProbablePrime` function continuously generates random odd integers until it finds a one that passes Miller-Rabin primality test, thereby resulting in long execution times. We changed this procedure so that each iteration ran in a continuous time slice. We then scheduled the next iteration for another time slice and returned control to the browser. This process continues until the key generation algorithm finds a number that passes Miller-Rabin test. We found that this approach was effective at keeping the browser responsive and preventing browser timeouts of JavaScript execution.

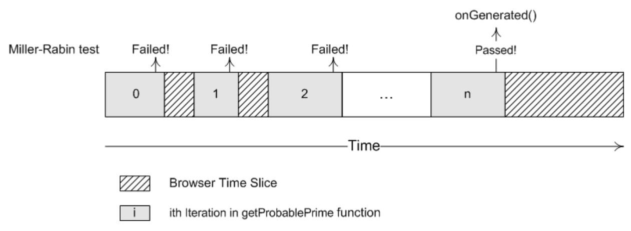

Figure 5: Deferred execution of prime number generation.

The above approach causes all procedures invoked during key generation to be asynchronous. Consequently, the key generation algorithm takes as input a call back function that returns the result to the browser. The following code snippet provides an example. Instead of directly returning `KeyPair`, the function `generateKeyPair`, accepts a callback object of type `KeyPairCallback` and returns `KeyPair` by calling `onGenerated` function once the key is generated.

```
KeyPairGenerator keyGen = null;
try
{
      keyGen = KeyPairGenerator.getInstance("RSA");
}
catch (NoSuchAlgorithmException e)
{
      Log.error(e.toString());
}
keyGen.initialize(512);  // Key Length
KeyPair keyPair = keyGen.genKeyPair();
// original signature: KeyPair generateKeyPair();
keyGen.generateKeyPair(new KeyPairCallback()
{
      public void onGenerated(KeyPair keyPair)
      {
            PrivateKey privateKey = keyPair.getPrivate();
            Log.debug("Private key : " +privateKey);
      }
});
```

Code complexity.

JCA, upon which our JavaScript library is based, uses several Java features, such as reflection, that are not supported by GWT. Consequently, we first modified JCA to a set of core components that were sufficient to implement cryptographic operations needed for Web2ID. We then used this stripped-down version of JCA with GWT to produce our JavaScript library.

7.2 Experiments

In this section we report on the performance of in-browser cryptographic operations that are required for the implementation of Web2ID. Our goal is to study feasibility and overhead of using in-browser cryptographic operations. We ran experiments on a machine with the following configurations. Intel Core 2 CPU, 980 MHz, 1.99 GB RAM, Microsoft Windows XP 2002 SP2. Google chrome v1.0.154.53 Firefox v3.0.8, Internet Explorer v7.0.5730.13, Opera v9.27, and Apple Safari v3.1.1.

The most expensive cryptographic operation that is required by Web2ID is asymmetric-key generation. Figure 6 shows the runtime of our RSA keypair generation function for keys of size 512 and 1024 bits. Since asymmetric key generation is a probabilistic process, the values reported are averaged results over ten runs. As this Figure shows, Google Chrome, which uses a fast JavaScript Engine (V8), generates a 1024-bit key pair in under 4 seconds. The slowest browser was IE, which took about one minute to generate a 1024-bit key pair. Because key generation is a one-time operation and the browser stays responsive during this time, we feel that this delay is acceptable.

Figure 7 shows the performance of RSA encryption/decryption using keys of length 1024 bits. As expected, decryption is more costly compared to encryption and the performance is quite reasonable for web applications. Of the browsers that we tested, Google Chrome had the best performance (less than 100ms for decryption using 1024-bit key).

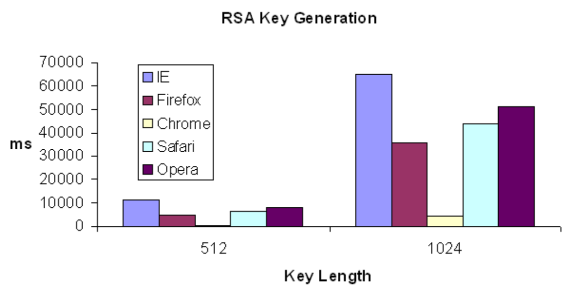

Figure 6: Key generation performance of our cryptographic library on different browsers.

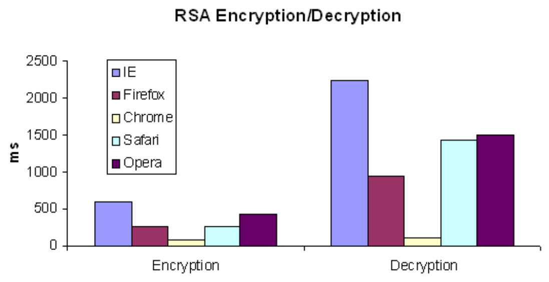

Figure 7: Performance of RSA encryption and decryption on different browsers.

8. RELATED WORK

Our Web2ID protocol can be realized with any secure mashup frameworks. They provide general infrastructure and environments for content providers to communicate in our identity management applications. There have been a couple of recent work that proposed secure mashup solutions including MashupOS [25], SMash [15], PostMessage method [3], and OMOS [27]. The main goal of these solutions is two-fold: to isolate contents from different sources in sandbox structures such as frames and to achieve frame-frame communication.

SMash [15] uses the concepts in publish-subscribe systems and creates an efficient event hub abstraction that allows the mashup integrator to securely coordinate and manage contents and information sharing from multiple domains. SMash mashup integrator (i.e., the event hub) is assumed to be trusted by all the web services. MashupOS [25] applies concepts in operating systems in mashup and develops sophisticated browser extensions and environments that enable the separation and communication of frames similar to inter-process communication management in the operating system. As mentioned earlier, the OpenMashupOS (OMOS) framework contains a key-based protocol providing secure frame-to-frame communication [27].

Camenisch *et al.* presented the architecture of PRIME (Privacy and Identity Management for Europe), which implements a technical framework for processing personal data [7]. PRIME focuses on enabling users to actively manage and control the release of their private information. Privacy policies for liberty single sign-on [17, 8] have been presented [21] by Pfitzmann. The paper identifies a number of privacy ambiguities in Liberty V1.0 specifications [18] and propose privacy policies for resolving them. A good article on the issues and guidelines for user privacy in identity management systems was written by Hansen, Schwartz, and Cooper [12].

In the federated identity management (FIM) solution by Bhargav-Spantzel *et al.*, personal data such as a social security number is never transmitted in cleartext to help prevent identity theft [4]. Commitment schemes and zero-knowledge proofs are used to commit data and prove the knowledge of the data. BBAE is the federated identity-management protocol proposed by Pfitzmann and Waidner [23]. They gave a concrete browser-based single sign-on protocol that aims at the security of communications and the privacy of user's attributes. Goodrich *et al.* proposed a notarized FIM protocol that uses a trusted third-party, called notary server, to effectively eliminate the direct communication between identity provider and service provider [11]. The main difference with these proposed privacy-aware ID management solutions and our approach is that we study ID management in the client-side mashup environment through a novel and efficient mashlet relay framework.

In the access control area, the closest work to ours is the framework for regulating service access and release of private information in web-services by Bonatti and Samarati [5]. They study the information disclosure using a language and policy approach. We designed cryptographic solutions to control and manage information exchange. Another related work aiming to protect user privacy in web-services is the point-based trust management model [26], which is a quantitative authorization model. Point-based authorization allows a consumer to optimize privacy loss by choosing a subset of attributes to disclose based on personal privacy preferences. The above two models mainly focus on the client-server model, whereas our architecture include two different types of providers.

9. CONCLUSIONS

As mashup applications increase in popularity, we expect that they will also be used with sensitive Web services, such as financial and banking applications. When mashups are used in such scenarios, it is key to provide features such as identity management and SSO. Existing identity management protocols are ill-suited for modern AJAX-based Web applications

This paper presented Web2ID, an identity management protocol for mashup applications. Web2ID preserves the privacy of the end user and eliminates the need for a trusted identity provider in the online single sign-on process. We described how this feature can be realized with conventional public-key cryptography. We also described a mashlet-relay framework that enables efficient yet indirect communication between two server mashlets via a local relay mashlet controlled by the user. Such a relay framework allows for attribute exchange without disclosing the user's surfing habits to service providers. Our implementation of Web2ID and the relay framework is implemented as an in-browser library and is fully compatible with commodity browsers.

10. ACKNOWLEDGEMENTS

The first author would like to thank the help of professors at Bahai Institute for Higher Education (BIHE).

11. REFERENCES

[1] OpenID Specification. `http://openid.net/developers/specs/`.

[2] RFC 4252, The Secure Shell (SSH) Authentication Protocol `http://tools.ietf.org/html/rfc4252`.

[3] Adam Barth, Collin Jackson, and John C. Mitchell. Securing Browser Frame Communication. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[4] Abhilasha Bhargav-Spantzel, Anna Cinzia Squicciarini, and Elisa Bertino. Establishing and Protecting Digital Identity in Federation Systems. *Journal of Computer Security*, 14(3):269–300, 2006.

[5] Piero A. Bonatti and Pierangela Samarati. A Uniform Framework for Regulating Service Access and Information Release on the Web. *Journal of Computer Security*, 10(3):241–272, 2002.

[6] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM Computer and Communication Security 2002*. ACM, 2002.

[7] Jan Camenisch, Abhi Shelat, Deiter Sommer, Simone Fischer-Hübner, Marit Hansen, Henry Krasemann, G. Lacoste, Ronald Leenes, and Jimmy Tseng. Privacy and Identity Management for Everyone. In *Proceedings of the 2005 ACM Workshop on Digital Identity Management*, pages 20–27, November 2005.

[8] S. Cantor, F. Hirsch, J. Kemp, R. Philpott, E. Maler, J. Hughes, J. Hodges, P. Mishra, and J. Moreh. Security Assertion Markup Language (SAML) V2.0. Version 2.0. OASIS Standards.

[9] Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). In *RFC3147*.

[10] Robert Ennals and Minos Garofalakis. MashMaker: mashups for the masses. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1116 – 1118. ACM, 2007.

[11] Michael T. Goodrich, Roberto Tamassia, and Danfeng (Daphne) Yao. Notarized federated ID management and authentication. *Journal of Computer Security*, 16(4):399–418, 2008.

[12] Marit Hansen, Ari Schwartz, and Alissa Cooper. Privacy and Identity Management. *IEEE Security and Privacy*, 6(2):38–45, 2008.

[13] Collin Jackson and Helen J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proceedings of the 16th International Conference on World Wide Web*, pages 611–620, 2007.

[14] Java Cryptography Architecture. `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html`.

[15] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.

[16] Krawczyk, Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. In *RFC2104*.

[17] Liberty Alliance Project. `http://www.projectliberty.org`.

[18] July 2002. Liberty Alliance Project: Liberty Protocols and Schemas Specification, Version 1.0.

[19] OpenJDK, `http://openjdk.java.net/`.

[20] OpenMashup. `http://www.openmashupos.com/`.

[21] Birgit Pfitzmann. Privacy in Enterprise Identity Federation - Policies for Liberty Single Signon. In *Proceedings of the Third International Workshop on Privacy Enhancing Technologies (PET 2003)*, volume 2760, pages 189–204, 2003.

[22] Birgit Pfitzmann and Michael Waidner. Privacy in browser-based attribute exchange. In *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 52–62. ACM, 2002.

[23] Birgit Pfitzmann and Michael Waidner. Federated Identity-Management Protocols. In *Security Protocols Workshop*, pages 153–174, 2003.

[24] RSA JS library. `http://www-cs-students.stanford.edu/~tjw/jsbn/`.

[25] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *ACM Symposium on Operating Systems Principle (SOSP)*, pages 1–16. ACM Press, 2007.

[26] Danfeng Yao, Keith B. Frikken, Mikhail J. Atallah, and Roberto Tamassia. Point-Based Trust: Define How Much Privacy Is Worth. In *Proc. Int. Conf. on Information and Communications Security (ICICS)*, volume 4307 of *LNCS*, pages 190–209. Springer, 2006.

[27] Saman Zarandioon, Danfeng Yao, and Vinod Ganapathy. OMOS: A Framework for Secure Communication in Mashup Applications. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference*, December 2008.