End-to-End Security for Enterprise Mashups

Florian Rosenberg[1], Rania Khalaf[2], Matthew Duftler[2], Francisco Curbera[2], and Paula Austel[2]

[1] Distributed Systems Group, Technical University Vienna
Argentinierstrasse 8/184-1, Vienna, Austria
florian@infosys.tuwien.ac.at
[2] IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY, 10532
{rkhalaf,duftler,curbera,pka}@us.ibm.com

Abstract. Mashups are gaining momentum as a means to develop situational Web applications by combining different resources (services, data feeds) and user interfaces. In enterprise environments, mashups are recently used for implementing Web-based business processes, however, security is a major concern. Current approaches do not allow the mashup to securely consume services with diverse security requirements without sharing the credentials or hard-coding them in the mashup definition. In this paper, we present a solution to integrate security concerns into an existing enterprise mashup platform. We provide an extension to the language and runtime and propose a Secure Authentication Service (SAS) to seamlessly facilitate secure authentication and authorization of end-users with the services consumed in the mashup.

1 Introduction

Mashups are an increasingly popular approach to develop new kinds of situational Web applications by combining content, presentation, and application functionality from disparate Web sources [1]. A vast number of mashup technologies and tools exist that provide a means of seamlessly "mashing" together several Web-based services and sources such as REST or SOAP services, feeds (RSS or ATOM) or plain XML or HTML sources. These mashup tools either provide a mashup language targeted for developers or provide an editor allowing a graphical mashup development such as Yahoo Pipes [2] or IBM Mashup Center [3].

In general, two different mashup types are dominant [4]: *Consumer mashups* are mostly for private use, combining data from several resources by unifying them using a common interface. *Enterprise mashups* combine different sources (content, data or application functionality) from at least one resource in an enterprise environment. An important distinction is the fact that enterprise mashups have some additional requirements such as security, availability or other quality of service items. Such enterprise mashups have an enormous potential by promoting assembly over development to reduce development costs and provision a new software solution within shorter time periods.

However, current enterprise mashup tools lack the ability to consume and integrate different services in a secure way when having completely diverse security requirements in terms of authentication and authorization [5, 6]. As a consequence, many mashup tools can only integrate security-free services and data sources or hard-code authentication data in the mashup code. Clearly, this is a problem in enterprise environments because users are more reluctant to give their authentication information to third parties (in fact, company policy may even prohibit that), resources typically have custom security requirements, and resources may support any of several authentication protocols such as HTTP basic authentication, custom application keys or more Web 2.0 like protocols such as OpenID [7] and OAuth [8].

We argue that seamless security support for enterprise mashups, in particular related to secure credentials management is required and needs to be integrated in the mashup language and/or tooling because users are not willing and should not disclose their credentials for different resources in the mashup definition. The mashup environment has to provide support for authentication and authorization, and delegation control allowing the execution of a service on behalf of a given user.

1.1 Illustrative Example

An enterprise mashup scenario depicted in Figure 1 is used to illustrate the problem and concepts. In this scenario, the hiring manager at Acme Inc (left) is hiring for a new position. He uses the enterprise mashup to schedule the interview with and get the resume of the candidate (bottom right).

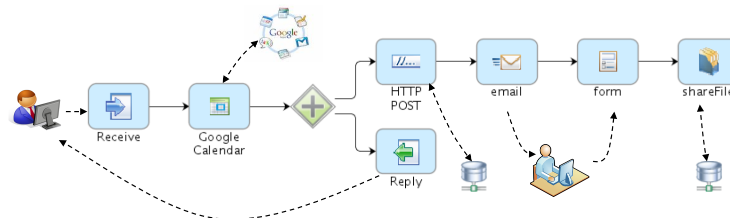


Fig. 1: Hiring Mashup Scenario

In order to do so, the mashup first makes a call to the hiring manager's calendar available via Google. Then, it forks: the bottom branch replies to the initial call and the top branch posts the available times to Acme's interview scheduling service, e-mails the candidate the final slot returned by that service and a link that should be followed to complete the process. Once the candidate clicks on the link, he finds a form where he fills in his personal information and attaches his resume. Finally, the mashup places the resume in the 'Files' file sharing service in LotusLive Engage, an online collaboration solution.

Interacting with multiple secured third-party services requires different sets of credentials and authentication protocols. For example, Google Calendar uses the OAuth protocol, Acme's scheduling service uses HTTP basic authentication, and the Files service requires an application key and the user in whose store the file is to be added. The Google Calendar call and the Files service call both require that the mashup interacts with them on the user's behalf - possibly after the user is no longer logged into the system.

1.2 Contributions

Seamlessly specifying and enforcing mashup security by supporting different authentication mechanisms requires both, a language extension and a runtime mechanism. In particular, this paper makes the following contributions to enable an end-to-end security solution for enterprise mashups:

- We provide a model and semantics for integrating security concerns directly into an existing business mashup platform (BMP), in particular the underlying lightweight workflow language Bite [9, 10] to address authentication and authorization from a language perspective.
- We describe a framework and implementation for homogenizing the authentication and authorization process within a mashup application for authorization mechanisms (e.g., basic authentication, custom application IDs, OAuth, etc) by leveraging a trusted Secure Authentication Service (SAS).
- We elaborate on the seamless integration and user experience of the authentication process by describing several mechanisms to allow mashup users to securely enter their credentials directly at the service provider (if possible) or by using the SAS.

The remainder of this paper is organized as follows: In Section 2, we describe the BMP project and the underlying Bite engine as the target enterprise mashup platform. Section 3 outlines the proposed security solution. Section 4 describes the Bite language extensions to enable security support followed by a detailed description of the SAS in Section 5. Section 6 evaluates and discusses the proposed approach followed by a discussion of related work in Section 7. Finally, Section 8 concludes this paper and outlines future work.

2 BMP and the Bite Language

The Business Mashup Platform (BMP) provides a hosted development environment for rapid development of situational business processes or enterprise mashups. It overlaps with the system in [11]. The graphical mashup development is browser-based, leveraging a BPMN-style editor, a forms designer and a catalog of extension activities that are offered to the designer in a palette. Once a mashup has been completely specified, BMP allows one click deployment of mashups that are immediately invokable. In the backend Bite code is generated and executed on the server.

Bite Language and Runtime. Bite is an XML-based REST-centric composition language designed to facilitate the implementation of lightweight and extensible flows[3]. The process model implements a subset of the WS-BPEL [12] execution semantics that consists of a flat graph (except for loops) containing atomic actions (activities) and links between them. Loops may be created using a dedicated `while` activity, the only construct allowed to contain other activities. Graph execution logic is encoded in conditional transition links between activities. Error handling is provided by special error links to error handling activities. Bite provides a small set of built-in activities: (1) basic HTTP communication primitives for receiving and replying to HTTP requests (`receiveGET|POST`[4], `replyGET|POST`, `receive-replyGET|POST`) and making HTTP requests to external services (`GET`, `POST`, `PUT`, `DELETE`), (2) utility activities for waiting or calling local code, (3) control helpers such as external choice and loops.

```
1  <process name="hiring">
2    <receivePOST name="hrInput" url="/hiring" />
3
4    <!-- get Google calendar data -->
5    <GET name="gcal" url="http://www.google.com/calendar/feeds/default/
         owncalendars/full">
6      <control source="hrInput" />
7      <input name=""></input>
8      <input name=""></input>
9      <security authtype="oauth" />
10   </GET>
11
12   <replyPOST name="hrReply" url="/hiring">
13     <control source="gcal" />
14     <input value=""/>
15   </replyPOST>
16
17   <!-- invoke interview scheduling service -->
18   <POST name="scheduleInterview" url="http://internal.acme.com/interview/
         schedule">
19     <control source="gcal" />
20     <input name=""></input>
21     <input name=""></input>
22     <security authtype="http_basic" notificationType="sametime"
           notificationReceiver="$:hrInput_User" />
23   </POST>
24
25   <!-- send an email to the candidate and collect candidate data using a
         special form activity -->
26
27   <!-- put all the collected candidate data on the Lotus file share -->
28   <shareFile name="storeApplication" ...>
29     <control source="collectCandidateData"/>
30     <!-- other parameter cut for brevity -->
31     <security user="$:hrInput_User" authtype="app_id">
32       <mapping>
33         <element name="par" label="Partner ID" applyTo="param" />
34         <element name="key" label="License Key" applyTo="param" />
35       </mapping>
36     </security>
37   </shareFile>
38 </process>
```

Listing 1.1: Hiring Mashup (simplified – without input parameters)

A Bite flow both calls external services and provides itself as a service. Sending an HTTP POST request to a flow's base URL results in the creation of a new flow instance that is assigned a new instance URL. This instance URL is returned

[3] We use 'mashup' and 'flow' interchangeably in this paper.

[4] The pattern `[x]GET|POST` denotes two different activities: `[x]GET`, `[x]POST`.

in the HTTP Location header field of the response. The instance URL contains a flow ID that is used for correlation of subsequent requests to that flow.

Each flow instance can define multiple receive activities corresponding to multiple entry points. These activities expose additional URLs as logical addresses of the instance's nested resources. POST requests directed to these URLs are dispatched to the individual receive activities in the flow model using the relative URLs defined in the activities' url attribute. This mechanism allows building interactive flows having multiple entry points for interacting with them. This behavior is leveraged by various activities such as Web forms that are designed as part of the mashup creation with the BMP.

A core concept of Bite is the extensible design that enables the developer community to provide additional functionality in a first-class manner by creating Bite extension activities and registering them with the Bite engine. This design allows keeping the language and its runtime very small and allows to implement other required activities as extensions. Extension activities can be created using Java or any scripting language supported by the Java Scripting API (e.g., Groovy, Ruby, Python, etc).

We show, in Listing 1.1, the (abbreviated) Bite code for the hiring sample in Figure 1. Each mashup has a root element called process (line 1). A new flow instance is created by sending a HTTP POST request to the relative URL /hiring of the initial receivePOST (line 2). The data associated with the POST request is implicitly available in variable hrInput_Output to all dependent activities. In this case the variable contains a map of all POST parameters. After completing the hrInput activity, the gcal activity is activated (lines 5–10). Transitions between activities are expressed by the control element (line 6). From lines 12–15, the mashup replies to the initial HTTP POST from the hiring manager informing him that he will receive an email with the selected interview date. The interview scheduling is executed in lines 18–23 by issuing a HTTP POST call to the interview scheduling service. Then, the other remaining steps are executed, e.g., sending an email using the sendMail activity and preparing the candidate form using the form activity – both implemented as Bite extension activities (not shown in the listing for brevity). Finally, the shareFile extension activity (lines 28–37) uploads the collected candidate data to LotusLive.

As stated in Section 1.1, the outgoing HTTP GET and POST call (gcal and scheduleInterview) and the shareFile activity require different security credentials that are required for successfully executing the mashup. The security element (lines 9, 22 and 31–36) and its semantics are presented in Section 4. For more details on Bite, its runtime and possible applications, see [9, 10].

3 Overview of the Enterprise Mashup Security Solution

Building security into an enterprise mashup platform requires to address (i) authentication of users at third-party services (i.e., verifying a user's claimed identity) and (ii) authorization in the sense that the user has to authorize the Bite engine to perform the task on the user's behalf. We have to distinguish two

aspects: First, security has to be addressed on a language level to integrate security concerns into the Bite language. A core requirement is to keep the language extensions minimal and provide extensibility support for various authentication protocols in a seamless user-centric way. Second, an extensible mechanism is needed to realize authentication and authorization of trusted services having different authentication protocols. This process is transparently handled by a *Secure Authentication Service* (SAS) that offers an OAuth interface, as described in detail in Section 5.

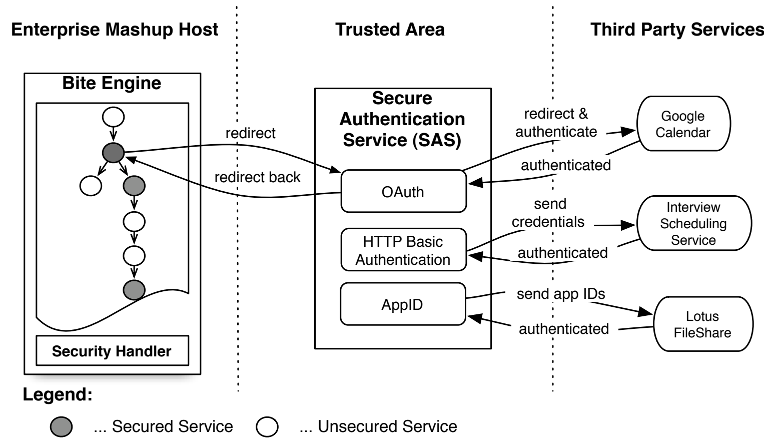


Fig. 2: End-to-End Security Solution Overview

In Figure 2, the basic overview of the security solution is depicted. The Bite engine including an executable flow is shown on the left (resembling the illustrative example from Figure 2). The white circles constitute services which do not require authentication, the gray ones require authentication. In the middle is the SAS which has to operate in a secure and trusted area within the company network as it manages credentials during the execution of a flow. On the right, the third-party services are depicted that will be invoked during the execution. Note that placement of the SAS is important: it must be in a trusted space. Some options include either at a third-party provider or a SAS at each service provider. As we are focused on enterprise mashups, it is viable that the SAS is a service provided by the enterprise itself making trust issues between users and the proxy infrastructure less of a problem. In this paper, we focus on an architecture and implementation whereby mashups can be secured using a security service; therefore, trust issues related to particular deployments are left to future work.

When the user triggers the execution of the flow by using the HTTP POST request in the Web form (or from another application), the mashup is executed and as soon as it reaches the first "secured" third-party service (cf. the `gcal` activity from Listing 1.1), the Bite engine will use a security handler to allow

the user to authenticate at the target service. The handler does this by interacting with the SAS. The SAS implements different security modules (OAuth, HTTP Basic Authentication and AppID) to provide support for different security mechanisms at the target service. The procedure for performing authorization and authentication has two cases:

- *Synchronous Authentication*: In this case the user is already interacting with the flow via a Web application and can thus be simply redirected to the SAS to perform the authentication at the target service. In the flow, this means that `receiveGET|POST` has been processed without yet reaching a corresponding `replyGET|POST` activity. For example, this is the case for the `gcal` activity from Listing 1.1 (lines 5–10) which is in between a `receivePOST` and a `replyPOST`.
- *Asynchronous Authentication*: In this case the flow already returned to the user by executing a `replyGET|POST` activity. Alternatively, an activity called `receive-replyGET|POST` is used to receive and immediately reply to an incoming request. Therefore, the user is no longer interacting with the flow and there is no connection that can be redirected to the SAS. For example, all activities from Listing 1.1 after line 18 (namely, `scheduleInterview`, `emailCandidate`, `collectCandidateData` and `storeApplication`). This requires contacting the user using asynchronous techniques to request him to authenticate at the third-party service. As shown later in the paper, we support email and instant messaging to do this.

The communication between the Bite engine and the SAS uses a slightly extended version of the OAuth protocol to seamlessly implement the handling of authentication and authorization between Web applications (in our case the Bite engine and the SAS). Therefore, the SAS design is generic and can be used by any mashup tool by implementing the corresponding OAuth connector that is capable of processing the proposed extensions.

4 Language Extensions for Security Specification

In order to enable security within Bite, the language needs to be extended to capture the security requirements such as authentication and authorization. A core requirement is to keep such language extensions minimal. On a language level, we focus on the *outbound security* in this paper, i.e., security support while calling an external service from Bite. *Inbound security*, i.e., authentication and authorization of users that want to execute a mashup is also supported but a detailed description is out of scope. For the sake of completeness, it is mentioned that it is done on a runtime level whereby the user authenticates using OpenID with an external authentication service and the authenticated user is injected into the process context where it can be checked against user restrictions on receiving activities (`receiveGET|POST`). If the user is allowed to access the receive, the activity activates, stores the message and the user in the appropriate variables, and completes. Otherwise, an error is sent back to the user and the

receive activity is not activated. If provided, user information from inbound security is stored at runtime in an implicit variable, `[activity_name]_User`. Hence, subsequent activities may use this variable to refer back to a particular user.

4.1 Security Extension and Semantics

A security extension element is provided and made optional for all outbound communication activities such as `GET`, `POST`, `PUT`, `DELETE` and all extension activities implementing custom behavior that may also require authentication.

Listing 1.1 has three security elements (lines 9, 22 and 31–36) in the flow. In Listing 1.2, we present the security element syntax.

```
1 <security authtype="http_basic|oauth|app_id" user="string|expression"?
2           roles="string (comma-separated)"? scope="activity|flow"?
3           notification="http|email|sametime"?
4           notificationReceiver="string|expression"?/>
```

Listing 1.2: Security Extension Element

Attribute Description: The attributes available for the security element are:

- `authtype`: Specifies the authentication type for authenticating a user at the target service. Currently, we support OAuth [8], HTTP basic authentication and customized application IDs that are frequently used by various service providers in the form of single or multiple GET or POST parameters. Handling these authentication types is transparently supported by the Secure Authentication Service (SAS), described in Section 5.
- `user`: Defines the name of the user (as a string or an expression) on whose behalf a specific service is executed. This user attribute is relevant especially for extension activities that support the “on behalf of” semantics. For example, the hiring flow from Listing 1.1 uses LotusLive to upload and share files. This application supports the “on behalf of” semantics by explicitly defining who uploaded a document indicated by the `user` attribute in the Bite flow (this username is then used in LotusLive’s file sharing service as the owner of the uploaded document).
- `roles`: Defines roles, that a user can have, in the form of comma-separated strings. If a role is used, role definitions must have been provided to the runtime.
- `scope`: It defines whether an activity’s security credentials are propagated to the other activities for re-use. If the attribute value is `flow`, credentials are propagated thereby avoiding repeated logins by re-using credentials to a service that is called more than once in a flow for the same user. In case of an attribute value `activity`, the credentials are not propagated.
- `notification`: It defines how a user should be notified that a service requires authentication. In case of a *synchronous authentication*, `http` can be used by redirecting to the SAS to request authentication and authorization. In the *asynchronous case*, the flow has to get back to the user to request

authentication. This can be done by blocking the activity requiring security, sending an email to the user (attribute value `email`) or sending an instant message (attribute value `sametime`) pointing him to the SAS, and resuming the activity once authentication/authorization is complete. Our approach supports Lotus Sametime, a messaging software used at IBM; other protocols may easily be added.
– `notificationReceiver:` It is only needed when using the `notification` type `email` or `sametime` because then it is necessary to have the contact details (e.g., email address or sametime contact). In case of `http`, it is not necessary, because the user is still interacting with the flow in the browser and is thus redirected to the SAS to perform the authentication.

4.2 Execution Semantics

The effect of the security elements on the execution semantics of the Bite language is as follows: Once an activity that has a security element is reached in a flow, the values of the security element's attributes are evaluated and stored in a *security context*, itself stored in the *process context* which maintains the state of execution for the flow instance. This information is used to lookup a corresponding security handler in a handler registry. The security context and the message payload are provided to this handler, which interacts with the SAS to provide the required authentication and authorization. If no credentials are available, the handler contacts the user sending them to the SAS. The handler then makes the secure call and returns the result to the activity implementation, which in turn stores it in its output variable. If the `scope` attribute value is set to `activity`, the security handler contacts the SAS through its OAuth interface to proceed with the required security handling and the OAuth connection tokens are destroyed after the authentication. If it is set to `flow`, these OAuth tokens are stored in the process context and can be reused in case the same service is called again in the flow for the same user. Reusing the same OAuth tokens for connecting to the SAS allows it to determine whether the user has previously authenticated and authorized Bite to invoke a given third-party service on its behalf. For more details on the OAuth handling see Section 5.

While the asynchronous case has no further effects on flow semantics, the synchronous (http) case is more involved because if credentials are not available then it needs to reuse one of the flow instance's open connections to contact the user, redirecting him to the SAS, and then back to the flow. Bite allows several receiving activities to be open (i.e., not yet replied to) at the same time. Therefore, the right open connection must be identified. To do so, open receive activities in the flow instance are checked for a matching '_User' variable value to the one in the security element being handled. The 'reply status' of a matching receive activity is set to 'awaiting_redirect' and a key is created for it against which the redirection from the SAS back to the flow can be matched. A reply is sent to the receive's open connection that redirects the user to the SAS. Once the user completes working with the SAS, a client-side redirect sends him back

to the flow. Also, the matched receive activity instance is found using the key and its reply status reset to 'open'.

If no match is found among open receives, then receives 'awaiting_reply' are checked as they will eventually become 'open' and may be used at that time. If no match is found among receives that are open or awaiting-redirect, the user is contacted as in the asynchronous case if contact information is provided in the security element definition. Otherwise, a fault is thrown.

A reply activity for a receive that is 'awaiting_redirect' must wait before it can send its response until the receive's reply status is again 'open' and no other security redirects are pending for that receive.

5 Secure Authentication Service

The Secure Authentication Service (SAS) is responsible for providing a proxy that can transparently handle various authentication types of different secure Web-based, e.g., RESTful services. Therefore, the SAS supports different security mechanisms and exposes itself using an OAuth interface, a popular protocol for managing authentication and authorization among Web-based APIs. The specification [8] defines it as follows: "OAuth protocol enables websites or applications (Consumers) to access Protected Resources from a web service (Service Provider) via an API, without requiring Users to disclose their Service Provider credentials to the Consumers."[5]. We provide a brief overview of the OAuth protocol and its extensions.

5.1 OAuth Principles

We leverage OAuth as the protocol for communicating with the SAS for two main reasons: OAuth is a well-understood and increasingly popular protocol for Web based applications and it implements a seamless way of handling authentication and authorization between a consumer and a provider. The consumer in our scenario is the Bite engine and the provider is the SAS itself. An OAuth provider has to provide three different request URLs: (1) a request token URL (relative URL `/request_token`); (2) a user authorization URL (`/authorize`); and (3) an access token URL (`/access_token`). A typical OAuth authentication and authorization is handled as follows: First, a consumer requests a request token using the request token URL (1) by sending a number of OAuth specific parameters, such a pre-negotiated consumer key to identify the consumer application, timestamp, nonce, signature etc. In case all parameters are correct and verifiable, the service provider issues an unauthorized *request token*. When the request token is received by the consumer, the user's browser can be redirected to the service provider to obtain authentication and authorization. This authorization ensures that the user sitting behind the browser explicitly ensures that the

[5] We are aware of the current security issue with OAuth [13], however, this will be fixed in a future version of the OAuth implementation that we currently use.

consumer Web application is allowed to access the service provider on its behalf. Once the authorization is performed, the service provider can redirect the user back to the consumer application (using a callback URL). Finally, the consumer has to exchange the request token for an *access token* at the service provider. This is typically granted if the user successfully performed the authentication and authorization in the previous step. This access token is one of the OAuth parameters that has to be sent with every further request to the protected service (among others such as consumer key, timestamp, signature, etc).

5.2 Third-Party Service Support

Transparently supporting a secure authentication and authorization of different third-party services through the SAS's OAuth interface requires extending the OAuth protocol. This allows the SAS to act as a "secure proxy" for various other authentication protocols. To do so, the SAS needs at least the URL and the authentication type of the target service. Since this information is available in the activity specification and the security extension in a Bite flow (e.g., Listing 1.1, lines 18–23), it just needs to be sent to the SAS to enable transparent third-party service authentication. Thus, a number of request parameters are added when the Bite engine requests a *request token* at the SAS as discussed below.

HTTP Basic Authentication. This type of authentication is widely used in practice although it is not very secure unless using SSL. It can be specified in Bite by setting the `authtype` to `http_basic` (cf., Listing 1.1, line 22). At runtime, the Bite engine contacts the SAS by requesting a *request token* by sending the following extended OAuth request:

```
http://sas.watson.ibm.com/request_token?oauth_consumer_key=bite_app
&oauth_timestamp=...&oauth_signature=...&oauth_...=...
&x-oauth_serviceurl=http://internal.acme.com/interview/schedule
&x-oauth_authtype=http_basic
```

The parameters `x-oauth_serviceurl` and `x-oauth_authtype` indicate the target URL of the secured third-party service and its authentication type from the `scheduleInterview` activity from Listing 1.1 (we prefix the extension with `x-` because this is a common pattern for HTTP header extensions too). In case of a synchronous authentication the user is redirected to the SAS Web interface, otherwise (in the asynchronous case) the user id specified in the `notification-Receiver` attribute receives a link that is used for authentication (basically the same that Bite redirects to in the synchronous case).

These two extension attributes are used by the SAS to make an outgoing call to the target URL in an iframe. It prompts the user for the credentials of the target service. If the authentication is successful, the HTTP `Authorization` header of the target service is intercepted by the SAS's proxying mechanism. A simple proxy servlet (`/proxy`) is used to achieve the proxying transparently at the SAS. The response of the target service is queued at the SAS, otherwise we would call the service twice: once for the authentication and once for the original

service invocation. When the first "real" service invocation is executed, the SAS will return the queued response during the authentication process.

Custom Application IDs. Support for custom application IDs requires adding another OAuth extension parameter called `x-oauth_appid_mapping`, that encodes details on how application IDs are queried from the user in a dynamically rendered Web form at the SAS and how this data is sent to the target service (e.g., in the HTTP header or as GET or POST parameter). Therefore, the security extension element in the Bite flow defines a `mapping` element (cf. Listing 1.1, lines 31–36). More specifically, this mapping states that the target service requires two parameters for a successful authentication, `par` and `key`, that need to be added as HTTP POST parameters (because this extension activity internally uses POST). Additionally, each element defines a label attribute used as a label for the HTML input element in the dynamically rendered authentication form.

Upon execution of such an application ID based service, the Bite engine serializes the Bite XML mapping into a simple text based form that is transfered to the SAS using the aforementioned OAuth extensions. Then the dynamically rendered authentication form is shown to prompt for the application IDs.

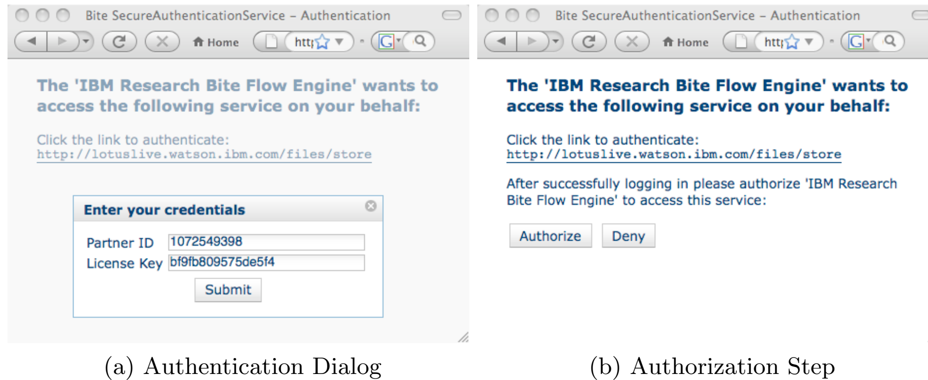
OAuth. Support for OAuth is also transparently supported by the SAS. In this case, the SAS just adds another layer of redirection between Bite and the target service provider without storing any information. It would be possible to implement a customized security handler to consume OAuth-based services directly (because Bite is already an OAuth consumer for the SAS). However, going through the SAS when consuming OAuth-based services has the advantage of handling multiple security mechanism transparently for the Bite engine.

5.3 Implementation Aspects

Bite and the SAS have been implemented in Java 1.6. Bite can be run on either a servlet container or WebSphere sMash server. The SAS implementation is based on Google's Java OAuth implementation providing multiple servlets for the different endpoints (request token, access tokens, etc). These servlets have been extended to support the above mentioned security protocols transparently. The Bite engine implements the OAuth client by using a specific security handler upon calling services from an activity with a security element (`SASSecurityHandler`). All other calls use a `NullSecurityHandler` that does not involve the SAS.

6 Case Study and Discussion

We have implemented the approach and provided a simple case study based on the illustrative example from Figure 1. It uses three different authentication mechanisms that are transparently handled by the SAS.



(a) Authentication Dialog (b) Authorization Step

Fig. 3: Custom Application ID Authentication and Authorization Process

Figure 3 illustrates the SAS's Web interface for the authentication and authorization for the `shareFile` activity from Listing 1.1 (lines 28–37) that uses custom application IDs as the "security" mechanism. Figure 3a shows the dynamically rendered authentication form based on the specification in Bite. When the user's browser is redirected to the SAS, the user sees the Web page as shown. By clicking on the link, the authentication box pops up and the user enters the credentials. After submitting the credentials, the user explicitly has to authorize Bite to call the service on its behalf (Figure 3b). When the user authorizes Bite, the flow proceeds its execution and the user is redirected back to the flow application (in the synchronous case), otherwise an error is thrown. The same user experience is available for HTTP basic authentication, however, the dialog box is not dynamically rendered but browser-specific.

The proposed approach based on the SAS effectively supports both, authentication and authorization of third-party services without the need to disclose the credentials to consumer applications (such as Bite in our case). A major focus was a seamless user-experience during the authentication and authorization process by automatically redirecting to the SAS to handle the authentication and authorization process. Therefore, it provides a mechanism for enterprise mashup solutions to transparently consume services in a secure way.

An important requirement for ensuring this end-to-end security is that the SAS has to run in a "trusted" environment because it stores intercepted credentials (for HTTP basic authentication) and stores the custom application IDs. Clearly, this is not an issue when using a third-party service supporting OAuth, because no credentials are disclosed to the SAS.

7 Related Work

Most existing mashup tools and products (e.g., Yahoo Pipes [2] or IBM Mashup Center [3]) do not address a secure end-to-end authentication and authorization

of different services within a mashup. Most approaches use plain text to manage user credentials within a mashup definition.

Pautasso [14] proposed BPEL for REST, an extension to the WS-BPEL language to enable language support for RESTful services in business processes. BPEL for REST does not provide any direct security support for invoking RESTful services. It allows the specification of custom HTTP headers which could be used to encode the HTTP basic authentication information. However, this would imply that password information is stored in cleartext in the BPEL definition.

Austel et al. [15] discussed the security challenges that need to be addressed for Web 2.0. Many of the challenges are addressed in our solution: protecting end-user credentials, secure and open delegation, authorization rules to limit delegation and a proxy to enable secure delegation to back end legacy systems. The paper mostly concentrated on OAuth as the wire protocol for secure delegation. It does not discuss proxy implementation details.

The approach introduced in this paper also shares several characteristics with identity metasystems (IMs) [16, 17], which also deal with the problem of users having multiple digital identities based on different protocols. IMs are typically used to allow clients to access Web applications on behalf of users. In the work presented here we consider the impact of multiple digital identities on the development and use of business mashups. The fundamental difference is our focus on a server side application (the mashup) acting on behalf of the end user.

A number of works have identified security issues for client-side mashups, i.e., running in a browser and communicating with other service through AJAX or related technologies. SafeMashups [18], for example, allows two web applications to communicate through a browser to securely authenticate each other and establish a trusted channel. Subspace [19] enables a secure cross-domain communication by providing a small JavaScript library to rule out a number of existing security flaws.

8 Conclusions and Outlook

In this paper we provided an end-to-end environment for securely consuming third-party services having diverse security requirements in a common service mashup application. The proposed approach was implemented as an extension to the Bite language and runtime by providing authentication and authorization transparently using a *Secure Authentication Service* (SAS) that can handle different security protocols common in the Web 2.0 area. The approach currently supports HTTP basic authentication, OAuth and customized application IDs that are frequently used in various RESTful services on the Web.

As future work, we plan to extend the support for further security mechanisms supported by the SAS, for example single sign-on approaches such as OpenID [7]. Additionally, we also want to reduce the need to explicitly specify the authentication type in the Bite flow, enabling automatic techniques to "guess" the security mechanism at the target service.

References

1. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding Mashup Development. IEEE Internet Computing **12**(5) (2008) 44–52
2. Yahoo! Inc.: Yahoo Pipes http://pipes.yahoo.com (Last accessed: May 19, 2009).
3. IBM Corporation: IBM Mashup Center http://www.ibm.com/software/info/mashup-center/ (Last accessed: May 19, 2009).
4. Hoyer, V., Fischer, M.: Market Overview of Enterprise Mashup Tools. In: Proc. of the International Conference on Service-Oriented Computing (ICSOC'08), Sydney, Australia, Springer-Verlag (2008) 708–721
5. Lawton, G.: Web 2.0 creates security challenges. Computer **40**(10) (2007) 13–16
6. Koschmider, A., Torres, V., Pelechano, V.: Elucidating the Mashup Hype: Definitions, Challenges, Methodical Guide and Tools for Mashups. In: Proc. of the Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM'09), Madrid, Spain. (2009) http://integror.net/mem2009/papers/paper14.pdf (Last accessed: May 21, 2009).
7. OpenID Foundation (OIDF): OpenID Authentication 2.0 - Final. http://openid.net/specs/openid-authentication-2_0.html (Last accessed: May 20, 2009).
8. OAuth Consortium: OAuth Core 1.0. http://oauth.net/core/1.0/ (Last accessed: May 20, 2009).
9. Rosenberg, F., Curbera, F., Duftler, M.J., Khalaf, R.: Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. Internet Computing **12** (September/October 2008) 24–31
10. Curbera, F., Duftler, M., Khalaf, R., Lovell, D.: Bite: Workflow Composition for the Web. In: Proc. of the International Conference on Service Oriented Computing (ICSOC'07), Vienna, Austria, Springer-Verlag (2007) 94–104
11. Lau, C.: BPM 2.0 – a REST based architecture for next generation workflow management. In: Devoxx Conference, Antwerp, Belgium (2008) http://www.devoxx.com/download/attachments/1705921/D8_C_11_07_04.pdf.
12. OASIS: Web Service Business Process Execution Language 2.0 (2006) http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel (Last accessed: May 28, 2009).
13. OAuth Consortium: OAuth Security Advisory 2009.1. http://oauth.net/advisories/2009-1 (Last accessed: May 20, 2009).
14. Pautasso, C.: BPEL for REST. In: Proc. of the International Conference on Business Process Management (BPM'08), Milan, Italy, Springer-Verlag (September 2008) 278–293
15. Austel, P., Bhola, S., Chari, S., Koved, L., McIntosh, M., Steiner, M., Weber, S.: Secure Delegation for Web 2.0 and Mashups. In: Proc. of the Workshop on Web 2.0 Security and Privacy 2008 (W2SP). (2008) http://w2spconf.com/2008/papers/sp4.pdf (Last accessed: May 21, 2009).
16. OASIS: Identity Metasystem Interoperability Version 1.0 http://www.oasis-open.org/committees/download.php/32540/identity-1.0-spec-cs-01.pdf/ (May 14, 2009).
17. Microsoft: Microsoft's Vision for an Identity Metasystem http://msdn.microsoft.com/en-us/library/ms996422.aspx (May, 2005).
18. SafeMashups Inc.: MashSSL https://www.safemashups.com (Last accessed: May 19, 2009).
19. Jackson, C., Wang, H.J.: Subspace: secure cross-domain communication for web mashups. In: Proc. of the International Conference on World Wide Web (WWW '07), Banff, Alberta, Canada, ACM (2007) 611–620