SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers

Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, Sachiko Yoshihama
{eb41704, sachikoy}@jp.ibm.com, {sbhola, msteiner, schari}@us.ibm.com
IBM Tokyo Research Laboratory, Kanagawa, Japan; IBM T.J. Watson Research Center, New York, USA

ABSTRACT

Mashup applications mix and merge content (data and code) from multiple content providers in a user's browser, to provide high-value web applications that can rival the user experience provided by desktop applications. Current browser security models were not designed to support such applications and they are therefore implemented with insecure workarounds. In this paper, we present a secure component model, where components are provided by different trust domains, and can interact using a communication abstraction that allows ease of specification of a security policy. We have developed an implementation of this model that works currently in all major browsers, and addresses challenges of communication integrity and frame-phishing. An evaluation of the performance of our implementation shows that this approach is not just feasible but also practical.

Categories and Subject Descriptors: D.2.0 [General]: Protection mechanisms, D.2.11 [Software Architectures]: Information hiding, domain-specific

General Terms: Security, design.

Keywords: Web 2.0, browser, mashup, component model, phishing.

1. INTRODUCTION

Web applications increasingly rely on extensive scripting on the client-side (browser) using readily available JavaScript libraries. One motivation for this is to enable a browser user experience comparable to that of desktop applications. The extensive use of scripting on the client side and programming paradigms such as AJAX [14] has also led to the growth of applications, called mashups, which mix and merge content[1] coming from different content providers in the browser. Mashups are now prevalent in a number of contexts, including news websites, which have integrity requirements, or web email, which handles confidential information. They are essential to an advertising supported business model, and for allowing user-generated content in Web 2.0 websites. The tremendous additional value that can be provided to users by mixing and merging content implies that such applications will eventually be prevalent in contexts with stricter data security requirements, like consumer banking sites and enterprise applications. Since existing browser security models were defined and developed without anticipating such applications, these applications are pushing the boundaries of current browser security models.

Since content in a mashup application can stem from mutually untrusting providers, it is clear that they should be built on a sound security foundation protecting the interests of the various involved parties such as the content providers and the end-user. For example, consider a mashup application scenario of a car portal where information from multiple car dealers, insurance companies and the user's bank could be combined and co-resident on the user's browser. It is clear that, at a minimum, we want certain security requirements to be enforced, such as the dealer's scripts not being able to modify each others car prices, nor should they be able to spy on a user's bank account information.

The traditional browser security model dictates that content from different *origins*[2] cannot interact with each other, while content from the same origin can interact without constraints (reading and writing of each others' state). This model does not support mashups, where controlled interaction is desirable. To overcome the no interaction restriction, mashup developers typically enable interaction by either (1) using a *web application proxy server* which fetches the content from different servers and serves it to the mashup, or (2) by directly including code and data from different origins (using `<script>` tags). In both cases, it appears to the browser that the mashup originates from a single origin, though it contains content from different trust domains[3], enabling uncontrolled interaction.

In this paper we examine the problem of building secure *componentized* mashups. We propose an abstraction on which security policies can be specified, and an implementation that realizes this abstraction for *unmodified* browsers.

1.1 Our Approach

Our abstraction involves encapsulating content from different trust domains as *components* running in a browser

[1] We use the term content to refer to active content, i.e., both data and JavaScript.

[2] Origin is a pair of hostname (DNS domain or IP address) and URI-scheme (protocol) from a given URL. The "same-origin policy" [17] usually includes also the port. However, this is not done by all browsers, e.g., Internet Explorer 6 ignores ports when comparing origins.

[3] Clearly trust domain is not the same as DNS domain. However, designing a secure solution for current browsers does involves some mapping between the two. In the remainder of the paper, the term domain used alone refers to DNS domain or IP address.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

window. Components can be loaded and unloaded dynamically, as the application in the browser window evolves. Components are wired together using communication *channels*, which are the only means for them to interact in the browser. The channel abstraction and security policies associated with channels are implemented by an *event hub* that is part of the Trusted Computing Base (TCB) from the mashup provider's perspective. Components are loaded from their own server (as opposed to being proxied or using a `<script>` element) and isolated from the mashup application code. This has security advantages in (1) not requiring the component to completely trust the mashup application, and (2) making the component abstraction compatible with password anti-phishing mechanisms that use the component (DNS) domain or the certificate of the SSL connection, like Passpet [28] and Microsoft CardSpace [15].

By creating a high level abstraction for cross-component interaction, we ensure that the different communication technologies used in our implementation can be replaced by, or even combined with, other technologies as they become available on the browser platforms.

We realize the component abstraction using the HTML `<iframe>` element (Section 16.5 of [19]), which was designed as a container for loading sub-documents inside the main document. The technical challenges that we address in the implementation are (1) enabling parent to child document communication *links* when the parent and child are from different trust domains, (2) ensuring integrity and confidentiality of information on these links, and (3) guarding against *frame-phishing*, where a malicious component in a mashup can change which component is loaded in another part of the mashup. Our implementation is a JavaScript library which is used by the mashup and component providers. It is available as open-source in the OpenAjax Alliance [1] SourceForge project. We have tested our library on Internet Explorer (IE), Firefox, Safari, and Opera.[4]

There are several competing technologies under development that attempt to address the problem of secure mashups. We briefly contrast them here and discuss them in more detail in related work (see Section 6). Many of these proposals require HTML and browser modifications, but the long timeline of adoption by standards committees, browser vendors, and eventually by end-users, makes these unviable for anyone wanting to build secure mashups in the near term. In addition, there are some proposals that work without browser modifications. These are either targeted at client-server cross-domain communication for mashups, which is a very different programming model, or web widget/gadget deployments. Our components have some conceptual similarity with web widgets, but need not have a user-interface. Also, web widget deployments offer a programming abstraction only to widget developers, not mashup developers. All these proposals typically, (1) are vulnerable to frame-phishing, (2) consider untrusted components but not untrusted mashup applications.

1.2 Building Secure Mashups

We believe that using our high-level abstraction of components which communicate through a mediated event hub, using channels, will provide a key primitive for secure componentized mashups. The mashup provider does not need to proxy all the components so that they appear to be from the same origin, nor resort to other unsafe practices, such as directly including scripts from different domains. It is also in line with the general principle of least privilege [22]: unavoidable programming and configuration errors mandate decomposition of web applications into small isolated units even when they come from the same administrative domain. Finally, it provides a general security mechanism to guard against cross-site scripting (XSS) attacks, as discussed next.

In the context of attacks on web applications, XSS attacks [23] are getting significant attention. These affect a special case of a mashup, where a website insecurely combines content generated by the website with content generated by its users, which are different trust domains. XSS attacks allow user-generated malicious content to not just read and write website-generated (trusted) content, but also gives it access to browser-cached credentials (e.g., cookies) for that trusted content. The common XSS mitigation approach is to disallow users to generate content that contains JavaScript, using content filters, but this is both (1) difficult to implement, resulting in many attacks against incomplete content filtering, and (2) limiting for user creativity. Using our component abstraction, user-generated code can be safely placed in a separate, less-trusted component.

The paper is structured as follows. In Section 2 we present our secure component model, followed by a discussion of security policies in Section 3. Then we describe our implementation and how we addressed various attack vectors in Section 4. We present a performance evaluation in Section 5, discuss related work in Section 6, and conclude in Section 7.

2. SECURE COMPONENT MODEL

In this section, we describe our secure component model for mashups. As discussed in the introduction, the current browser security model either allows different content to arbitrarily interact if they are from the same origin, or disallows all interaction if they are from different origins. Thus, it is clear that the current browser security model is insufficient for mashup applications. What is desirable is a new security model that allows content to be separated by trust domain, with a carefully mediated interaction between such separated content. Furthermore, to make this accessible from a programmer's perspective, there must be a high level programming interface that allows creation of secure mashups, and makes this easy.

Both from a programmability and usable security perspective, it is essential to follow the concept of information hiding [18] by limiting the exposure of internal design and state. Thus, our model is analogous to using a message-passing interaction style instead of a shared-memory style. We first present the model, and then explicitly specify the security requirements.

2.1 Model

Figure 1 graphically represents, in an abstract manner, our proposed model for secure mashups. The model consists of *components,* with input/output *ports*, and an *event hub,* with mediated communication *channels*.

The *mashup application*, provided by the *mashup provider*, consists of an event hub and one or more components, which can be provided by third-party *component providers*. Com-

[4]More specifically, the tested browser versions are Internet Explorer 7.0, Firefox 2.0.0.7, Safari 3.0.3, and Opera 9.24. There is no technical reason, though, that the library would not work on different versions or other browsers.

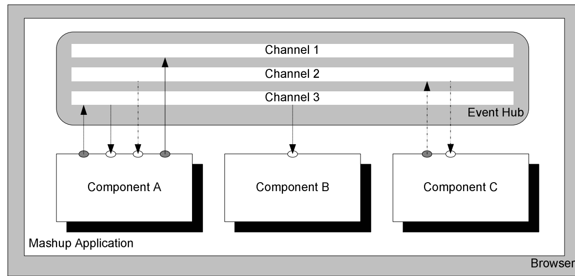

Figure 1: Component Model

ponents could be visible or invisible. Visible components share the browser display in a manner determined by the mashup application. A component contains (active) content from one trust domain. The components, and therefore also the trust domains, are logically separated and can only communicate with each other through the mediated channels implemented by the event hub. A component may specify input/output ports (respectively depicted in the figure by white and gray ellipses) that define the types of input and output that the component expects in order to function properly. The event hub wires the component ports to the appropriate channels (depicted by the arrows in Figure 1).

The event hub is a publish/subscribe system with many-to-many channels on which messages are published and distributed. This model differs from a traditional publish/subscribe model by separating the namespace of the component ports from the namespace of the channels. This avoids clashes in the namespace of the component ports and enhances the re-usablity of the components. In Figure 1, Component A can publish to Channel 1 and Channel 3 and is subscribed to Channel 2 and Channel 3. The channels support asynchronous pass-by-value communication. An asynchronous model, though typically considered harder to program to, is a natural one for applications running in the browser, since they have significant logic for asynchronous event handling related to user-interface (UI) events. In addition, an asynchronous model allows more implementation choices, since application code running in a browser is single-threaded.

The model can be extended hierarchically, i.e., a component could contain sub-components, and an event hub which mediates between these sub-components.

2.2 Security Requirements

We explicitly list the security requirements for the previously presented model, when embodied in a browser.

1. The DOM (Document Object Model) [26] tree of each component is completely isolated from other components. That is, there is no reading and writing of DOM elements across trust domains.

2. The JavaScript namespace of each component is completely isolated from other components.

3. Components can be loaded directly from the component provider. From a security perspective, this has the following advantages: (1) A component is protected (isolated) even from the mashup application code; (2) It is compatible with password anti-phishing mechanisms, like Microsoft CardSpace [15] and Passpet [28], that take into account the (DNS) domain of the component, or the certificate of the SSL connection. This is important if the component needs to authenticate the user to perform its function; (3) It retains the possibility of using cookies, secured by the DNS domain, as is still common practise for session authentication. This also has the benefit of not needing to deploy an additional web proxy.

4. Inter-component communication is secure: Malicious components can not affect the integrity and confidentiality of communication between other components. Specifically, inter-component communication in our model is a two-step process: component to mashup application and vice versa. The model requires one-way authentication: component to mashup application. Two-way authentication could also be implemented, however the context in which we have used our client-side library did not require it: we integrated that requirement into server-side access control for component loading (section 3.2).

5. Component loading and unloading is completely under the control of the mashup application. A malicious component cannot replace a peer component in another part of the application without detection and appropriate (context-specific) remediation by the mashup application.

6. Channel access control is completely under the control of the mashup application.

Limiting Trust in Mashup Application Based on these security requirements, a mashup application cannot directly attack the integrity or confidentiality of a component. However, a component has to trust the mashup application for the integrity of its user interface (excluding authentication, as mentioned above). This is reasonable from an end-to-end perspective since the end-user trusts the mashup application for integrity of the overall user interface (UI), and there is a simple delegation of trust from the mashup application to components for parts of the UI. Note that the mashup application cannot attack the confidentiality of information displayed by a component in its UI.

A component needs to trust the mashup application for integrity and confidentiality of communication with its peer components. Removing this trust is expensive without inbuilt browser support. The cross-document messaging proposal in HTML 5 (section 6.2) addresses this.

3. SECURITY POLICY

The component model described in the previous section provides component isolation and mediated inter-component communication. To describe which component interactions are permitted and which are forbidden, we need to define a security policy for the model.

The policy specification can be derived from many sources of high level policies. This is especially true for enterprise-class mashups, where the security policy may be a combination of enterprise-level policy, department policy, and end-user policy. Component providers may want to express how their components are to be integrated into a mashup

application. For example, a component provider may constrain what data is exposed by the component to a mashup provider, and what the mashup provider may do with it. On the other hand, mashup providers may not trust components equally, and may want to prevent certain components from communicating with each other. In general, security policies for componentized mashups can have many facets, depending on which entity specifies the policy.

We have identified two different levels at which policies can be expressed: low level policies that express basic access control, and high level policies which express interactions at a more semantic level. We consider basic access control policies in the remainder of this section. We envision that basic policy is derived by inputs from high level policies.

Next, we describe two complimentary aspects of access control: component interaction in the browser, and component to server interaction. Both are necessary for end-to-end control on what accesses are permitted to whom.

3.1 Component Access Control in the Browser

This controls who is allowed to (1) create and destroy named components, (2) create and destroy named channels, and (3) which components can publish or subscribe to events on a named channel. The policy is enforced by the event hub. The subjects in the policy are named components or trust domains.

We consider a special case of such a policy, where component and channel creation (and destruction) can only be done by the mashup application. This special case is in line with current practice, and has the advantage that it makes policy specification very straightforward. The policy is implicitly specified by the mashup application code by, creating channels, loading the different components, and then wiring them together by specifying the channels that connect to the input/output ports of each component.

Another alternative is for the event hub to read security policy constraints from a configuration policy file. This decouples the policy definition from the actual coding process of the mashup application and enables scenarios in which the mashup developer is not the one defining the policy for the component interaction.

3.2 Component to Server Access Control

For non-public services exposed by web servers, user authentication is typically required in order to perform the access. Components should not have to trust the mashup application to acquire user credentials. Following the principle of least privilege [22], the authentication should also be coupled with limited delegation logic, e.g., as in SecPAL [3].

While the details of our current solution are outside the scope of this paper, we give a brief overview here. In addition to traditional user authentication, our access control model authenticates a component and its associated (limited) access rights, as well as the mashup application it is loaded in. In particular, it also makes sure that components are loaded in a proper iframe of the component's domain and not vulnerable to a man-in-the-middle attack. This approach allows us to have a unified access control policy specification for expressing limited delegation within and across trust domains, and for both client-side mashups (mashup happens in browser) and server-side mashups (mashup happens at server). Our solution can be considered an extension of current authentication and delegation protocols, such as Yahoo's BBAuth [27] or Google's AuthSub [9].

4. IMPLEMENTING THE MODEL

In this section, we discuss our design and implementation of the secure component model. Our approach is to leverage the current browser security model to isolate components belonging to different trust domains, and then to enable interaction by sending-receiving on named component ports. These get translated to send-receive on named shared channels. Our approach does not require any browser modifications, hence adoption can be immediate.

In section 4.1 we give some background on browser abstractions and their security, which is relevant to our solution. Section 4.2 provides an overview of our solution, and section 4.3 gives details on the key aspects of the solution.

4.1 Background

We review a few concepts from HTML documents and their properties, including the "same-origin" security policy of browsers.

An HTML document loaded into a browser is represented in an object model called the Document Object Model. A document has a `domain` property which is the hostname of the server it was accessed from. The hostname could be a numeric IP address or a DNS domain name. DNS names are hierarchical, and browsers allow a document to relax its origin via the `domain` property. For instance, a document with domain `foo.bar.baz.com` can change it to `bar.baz.com` or `baz.com`. A document has a `window` object with a `location` property that represents the URL of the document. It is possible to change the `location` property of the document, by updating the `window.location`. This will cause a new document to be loaded to and to replace the current document in that window.

Documents can include frames using HTML `<frame>` and `<iframe>` tags where each frame is a document with `domain` and `location` attributes (and a `window` object). Frames can include sub-frames, forming a hierarchy, which we will refer to as a *frame hierarchy.* For consistency of terminology, we will refer to even the top-level document in the frame hierarchy as a frame. The "same-origin" policy says that documents from different domains that are in the same frame hierarchy cannot examine or alter each others internal state. Code running in a frame can read/write the internal state of all documents from the same origin (that is, domain) that are part of the same frame hierarchy. For purposes of our discussion, we will use the terms origin and domain interchangeably.

Even if frames are from different domains, a frame can typically write the `location` property of any frame in the same frame hierarchy, regardless of origin. Such browsers are labeled *permissive browsers* [12], and include Firefox, Safari, and some configurations of IE 6 and IE 7. However code from one domain can never read the `location` property of a frame from another domain.

4.2 Solution Overview

In this section, we give an overview of the key features and issues addressed in our solution, which include component isolation, enabling and securing component-mashup communication links, and protection against frame-phishing.

Component isolation using iframes: In our solution, we load components which come from different trust do-

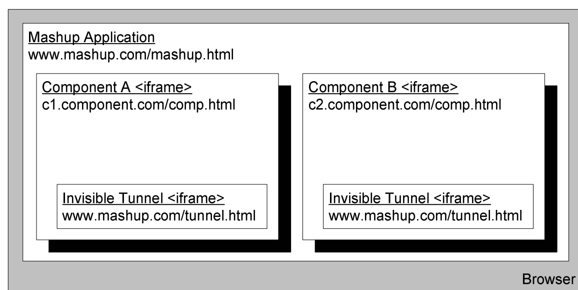

Figure 2: Isolated components with component-mashup communication

mains into different iframes, i.e., they represent different sub-frames. Even in the case where a single host is serving components from multiple trust domains, isolation is obtained by having the host use multiple DNS names, one for each trust domain.[5] However, given that components can change their `domain` properties as described earlier, the DNS domains used are such that a component cannot relax its `domain` property to attack another component. For instance, a server `foo.bar.com` that serves components from trust domain t1 and t2 can create two DNS domains `t1.foo.bar.com` and `t2.foo.bar.com`. Unless both components relax to the same super-domain no attacks are possible.

Component-mashup communication links: While isolation can be achieved by placing components in different frames, the challenge is to enable cross-domain inter-component communication, since it is not explicitly supported in browsers. Recently, an approach to communicate between iframes using the fragment identifier of the URL of the iframe has been discovered [4]. The communication is based on the observation that even though the parent and child iframes have different origins, the parent can write to the child's `location` property. Note that if we only modify the fragment identifier of the `location` property the document will not be reloaded and hence the document state can be preserved because the fragment identifier was designed to be used for navigation inside a document. This technique has been used in the Dojo JavaScript toolkit [7] to circumvent the same-origin policy for client-server communication, and more recently in XDDE [13]. We reused this technique to enable communication between components.

As mentioned previously, browsers vary in the constraints they place on navigating the frame hierarchy. Hence, one needs to organize the iframes in a way that allows for fragment communication. Figure 2 illustrates the iframe configuration used with three trust domains in the mashup. The first trust domain, represented by origin `www.mashup.com`, is that of the mashup application. It uses two components, both from the same server with DNS domain `component.com`, but from two different trust domains. To leverage the iframe isolation, the components are loaded from different origins: component A is loaded into `c1.component.com` and component B into `c2.component.com`. In each of the components, there is a tunnel iframe loaded from `www.mashup.com`. This tunnel can access the JavaScript context of the mashup application since it is in the same origin as the mashup application. The tunnel and the component iframes communicate cross-domain using fragment identifiers. We have also investigated the use of other inter-frame communication mechanisms, details of which are described in Section 5.

Link Security: Permissive browsers allow complete navigation of the frame hierarchy. For example, in Figure 2, component B can get direct access to the `window` object of component A even though they both come from different origins. Note that it is not possible for B to read the `location` property of an iframe not in the same origin. This ensures confidentiality of the link between component A and the mashup. However, integrity is not guaranteed.

To ensure integrity, we extend the fragment communication protocol with an additional security token. The details are discussed in the next section.

Protection from frame-phishing: Since it is possible for a malicious component to write to the `location` property of another component's iframe, it can navigate a component away from its URL to another, possibly malicious, URL. We refer to this as frame-phishing, and is a common vulnerability of current sites that use iframes for isolating untrusted content.

Frame-phishing is dangerous for a number of reasons, (1) it compromises integrity of a part of the application view that was not meant to be controlled by the malicious component, (2) it can be used to steal information entered into the phished frame, much like the more common phishing of sites, which could include personally identifiable information (PII) or passwords. Since the URLs loaded by iframes are not visible in the browser's URL bar, one cannot rely on the end-user to detect frame-phishing, and so it has to be done in an automated manner.

Note that this attack is not only limited to changing the `location` of a component's iframe. It can be performed on the tunnel iframe, and even the main mashup document. One could argue that such an attack on the mashup document could be detected using the browser's URL bar, but (1) the URL bar has been shown to offer very weak phishing protection [6], and (2) the timing of the attack can be arbitrary, unlike conventional phishing attacks. For instance, a user that keeps a web application open in one of the tabs of their browser could be attacked at an arbitrary point in the lifetime of the application, after the user has verified the URL in the URL bar.

We use a combination of event handlers, timeouts, and communication using the tunnel iframe, to detect such attacks. The details are discussed in the next section.

4.3 Solution Details

Our implementation is the SMash JavaScript library used by both the mashup application and component programmer. The library is structured as a layered communication stack, where the peer layers in the component and mashup application communicate using the lower layers. This is shown in Figure 3: the event hub layer, event communication layer and the fragment communication layer.

The *event hub layer* is aware of component ports and channels wiring these ports. In the mashup application, this layer is responsible for loading and unloading components, creating and deleting channels, and wiring the ports of the components to channels. In a component, this layer is aware of the component's ports and handles sending and receiving events on these ports. The *event communication layer* is re-

[5]HTTP virtual hosts and DNS aliases, e.g., wildcard resource records, allow easy realization of such co-residence.

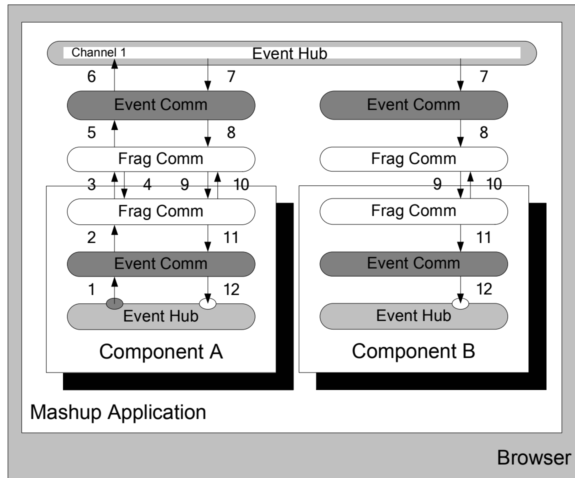

Figure 3: Layered communication stack

sponsible for composing the messages which are used to multiplex the multiple component ports on a single link. The *fragment communication layer* is the only layer aware of the use of fragments to communicate between the component and the mashup application. By substituting this layer, it is possible to employ another low-level communication mechanism, e.g., inter-iframe communication potentially built into future browsers (see Section 6).

In the following subsections, we describe the API offered to mashup and component programmers, provide some details on fragment communication and corresponding integrity protection, and describe how we detect frame-phishing.

4.3.1 API

The API offered by SMash is shown in Table 1. Most of the API is self-explanatory. We draw, though, attention to the component state lifecycle, shown with the `getComponentState` function. This state management is offered as a way for the programmers to deal with the asynchrony of component loading/unloading, and the concurrency of wiring a loaded component. The state transition from *loaded* to *wired* is done by the mashup application, to indicate to the component that it can start publishing events. The transition to *startedCleanup* is also done by the mashup application, when it intends to unload the component, while the transition to *doneCleanup* is typically done by the component (subject to a timeout, to handle misbehaving components). The remaining transitions are initiated inside SMash. To avoid polling of component state, the constructors of the hub (not shown) allow the mashup application and the component to register listener functions on the state transitions.

4.3.2 Fragment Communication and Link Integrity

As mentioned earlier, fragment communication is used to implement the links between the mashup application and a component. However, such communication is limited since current browsers implement a limit on the URL length. In SMash, we work with a conservative limit of 4000 bytes, which works well in all major browsers. Long messages have to be split correspondingly into segments which, with payload and protocol overhead, do not exceed that limit.

To send a segment to the tunnel, the component writes it to the fragment in the tunnel's `location` property, and vice versa. To write the next segment, the component has to wait till the previous segment has been read by the tunnel. This is implemented by using a periodic timer at both the component and the tunnel, and polling for changes to one's own `location` property. As the component cannot read the `location` of the tunnel, it needs to get an explicit acknowledgment when the tunnel has read the previous segment. This is accomplished using acknowledgment messages.

Link Integrity: As mentioned previously, link confidentiality is implicit in fragment communication. For link integrity, we have to guard against a malicious component modifying the `location` property of a peer component or the peer's tunnel.

We cannot prevent a malicious component from modifying `location`. However, it can only do so atomically, i.e., it has to overwrite the complete value of the current fragment (overwriting the non-fragment part is discussed under frame-phishing, in the next section). Therefore we use the approach of embedding a shared secret, in each fragment, to authenticate the sender of the fragment, and hence ensure its integrity. This shared secret is embedded in the clear, as the attacker cannot read the `location`.

We also need to securely distribute this shared secret between the mashup application and the component. The SMash library in the mashup application creates the secret, an unguessable random value. When creating the component, it includes the secret in the fragment of the component URL. When the component creates the tunnel iframe it passes the secret in the same manner. One of the reasons for doing the initial secret distribution to the component using the fragment part of the URL is that RFC 2616 forbids fragments in the `<Referer>` header. So this secret will not be leaked by the component to remote parties when it loads data from them.

To ensure that the attacker cannot mount an attack by deleting fragments, we use sequence numbering on consecutive fragments, so missing fragments will be detected and attributed to an attack.

4.3.3 Protection from Frame-Phishing

As described earlier, frame-phishing is a dangerous attack on mashups that use iframe isolation. We describe how we detect such attacks in SMash.

Since the mashup application cannot read the `location` property of its components, we cannot simply poll for changes to this value. The parent iframe could set `onload` and `onunload` event handlers on the child iframe. But browser implementations vary and are unreliable in invoking these handlers when a child event happens. We use a combination of `onunload` event handler on one's own frame (which in our experience is reliable), timeouts, and communication using the tunnel iframe, to detect such attacks.

There are three targets to consider with respect to such attacks: (1) component, (2) tunnel, (3) mashup application.

When the component is replaced by an attacker, the component's `onunload` event handler is called. At this point, it could try to notify the mashup application. However, since it communicates with the mashup application asynchronously, using the tunnel, there is no guarantee that this communication will succeed before the unload completes. Instead, we utilize the fact that replacing the component will also cause

Mashup API	
`loadComponent(componentId,inPortNames,outPortNames)`	load a component with a given id
`createChannel(channelName)`	create a channel with the given name
`deleteChannel(channelName)`	delete a channel
`addReader(channelName, componentId, inPortName)`	wire an input port to a channel
`addWriter(channelName, componentId, outPortName)`	wire an outport port to a channel
`removeReader(channelName, componentId)`	unwire an input port
`removeWriter(channelName, componentId)`	unwire an output port
`broadcastOnChannel(channelName, message)`	event broadcast by mashup application
`getComponentState(componentId)`	get component state: *start*→*loaded*→*wired* →*startedCleanup*→*doneCleanup*→*unloaded*
`componentWired(componentId)`	transition to *wired* state
`startCleanupComponent(componentId)`	transition to *startedCleanup* state
Component API	
`getComponentState()`	get component state
`registerCallback(inPortName, callback)`	register function to handle event on input port
`publish(outPortName, message)`	send event on output port
`doneCleanupComponent()`	transition to *doneCleanup* state

Table 1: API

the `onunload` event on the tunnel to fire, as it is a child of the component iframe. The tunnel can communicate synchronously with the mashup application using a JavaScript function call, and informs it of the unloading before it returns from the `onunload` event handler. Hence, we handle cases (1) and (2) in a unified manner.

Another possibility for the attacker is to replace the component *before* the tunnel iframe is loaded. This case is handled by setting a timeout, in the mashup application, for successfully establishing initial communication with the tunnel. If this timeout expires, an application specific error-handler is called, which could decide to unload and reload the component.

Using the tunnel's `onunload` event handler, though, poses a challenge as the event is also called when the user navigates away from the mashup application. To avoid false positives, we briefly delay the alert issuing in a timer. The timer will only fire and notify the application of a potential frame-phishing attack, if the user remains in the mashup application.

Finally, we need to detect frame-phishing of the mashup document. We have found it hard to distinguish between the two cases of either a frame-phishing attack, or the user voluntarily navigating away from the document. In the interim, we use an alert box to notify the user about the change in the document URL. This has a negative impact on usability, but in the absence of browser support to distinguish the two cases, it seems the only way to ensure protection of the user.

The authors acknowledge that providing the end-user with pop-up warnings, albeit being a best effort, is hardly a fail-safe solution. An alternative solution would be to stop all interaction within the mashup application when an attack is detected. However, browser level fixes are the ideal solution to these problems. Recently, there has been an in-depth study of the navigation behavior of browsers resulting in several improvements [2]. Until these fixes become widespread, our solution can be applied on current browsers.

5. PERFORMANCE EVALUATION

In this section we describe a preliminary performance evaluation of our implementation. The evaluation consists of micro-benchmarks in which we vary the number of components, to measure the scalability of the implementation (assuming all components are in different trust domains). Additionally, we vary a system parameter, the polling timer interval used for fragment communication, to see how it impacts communication throughput. The metrics used in the experiments are:

1. *Event Rate*: This measures the maximum event rate we can sustain, for small event payloads. This is typical for many mashup applications where components exchange short events in response to user actions.

2. *Data Throughput*: This measures the maximum rate in kilobytes/s, and is important for data intensive mashup applications that want to transfer large volumes of data between trust domains, inside the user's browser.

3. *Component Load Latency*: This measures the latency to load a component and setup the communication link between the component and the mashup application.

Next, we describe our testbed, followed by the results for each of the above metrics. Finally, we briefly discuss using an alternative inter-frame communication mechanism that uses Java applets, in the context of SMash.

5.1 Testbed

To do a fair comparison across the different browsers, all tests were performed on exactly the same test platform. The underlying test machine was an Intel Core 2 Duo T7300 @ 2.0 GHz with 2 GB of RAM. This machine was running Windows XP with VMware player 2.0.1. The actual performance evaluation was done on a VMware virtual machine emulating a single processor @ 2.0 GHz, 1 GB of RAM and running a clean fully patched Windows XP. Within this machine, we had Apache Tomcat 6.0.14 serving the data to the browsers. All of the tests were performed on Firefox 2.0.0.7, Internet Explorer 7.0 (IE), Safari 3.0.3, Opera 9.24. To minimize measurement errors, all tests were performed multiple times until the standard deviation of the results became acceptably small. Due to space limitations, we only present the event rate and data throughput results for Firefox and

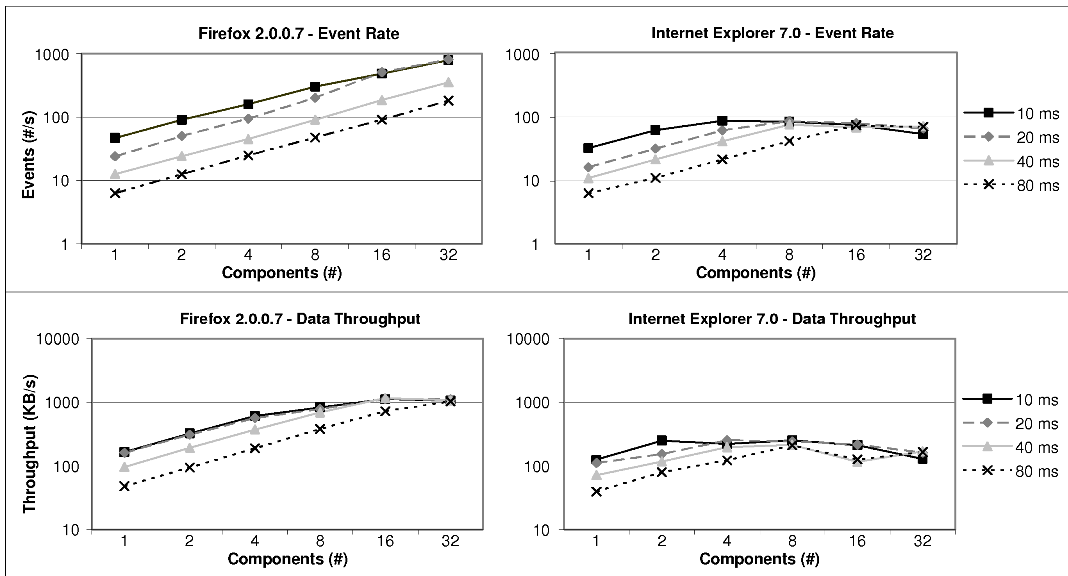

Figure 4: Event rate (top) and data throughput (bottom) for Firefox and IE.

IE, since they are the most popular browsers.[6] [7]

5.2 Event Rate

We used a small payload of 13 characters, representing a simple event name/value pair. Figure 4 (top) gives a summary of the measured performance as the number of components is increased from 1 to 32 and the polling timer is varied from 10ms to 80ms. Both the X and Y axis are on a log scale.

For each browser we observed a saturation point. This point, mainly caused by CPU load, varies across browsers, timer intervals, and number of components. Prior to reaching this saturation point, there is a linear increase in aggregate rate with increasing number of components and timer frequency. Once the saturation point is reached, and more components are added, or the timer frequency is increased, performance loss due to over-saturation becomes apparent. An adaptive timer frequency, that detects and reacts to over-saturation, may be able to avoid this behavior. Overall, FireFox shows significantly better performance than IE.

For many mashups the inter-component events are triggered by user actions, requiring only a low event rate. The measured rates are sufficient for such applications.

5.3 Data Throughput

The data throughput was tested by transferring a total of 1 megabyte (MB) of data from the mashup application to the components. Given the approximately 4 kilobyte (KB) limit of the fragment payload[8], this resulted in transferring 256 messages from the mashup application to the components. Figure 4 gives a summary of the measured performance.

Like the event rate experiments, the throughput increases linearly with the number of components and timer frequency, prior to the saturation point. At the saturation point, adding additional components or increasing the timer frequency results in degraded performance.

5.4 Component Load Latency

For evaluating the component loading performance, we measured per component loading latency while increasing the number of components being concurrently loaded. The overall loading latency is measured from the moment at which the mashup application creates the components until the moment at which the mashup application receives confirmation of a successful load of all components. This confirmation is communicated across domains using fragment communication. Figure 5 gives an overview of the measured performance.

We observe that load latency does not increase with an increase in number of components. In fact, the loading latency per component initially decreases and and then starts to increase back towards its initial value. The latter can be explained by the additional system load due the concurrent loading of many components. Note that caching was disabled on all browsers except for Safari [9].

5.5 Alternative Inter-Frame Communication

As mentioned in section 4.3, it is possible to employ alternative inter-frame communication mechanisms in SMash, by replacing a layer in the communication stack. In particular, alternatives such as Java applets, Adobe Flash, or Google Gears, though lacking universal support in browsers, could be used to increase throughput when available. In this experiment, we investigated the use of Java applets.

Our implementation employs different components that load applets from the same location, and use the static vari-

[6]Results for all browsers are available in the Appendix of the long version available from `http://domino.research.ibm.com/comm/research_projects.nsf/pages/web_2.0_security.smash.html`

[7]For Opera, we observed some anomalies in system load, that are probably due to a bug in the rendering engine when using fragments in invisible frames.

[8]This limit was chosen based on the limit for IE, which had the smallest fragment length limit of the tested browsers.

[9]We were unable to disable caching on Safari due to the unavailability of cache configuration parameters.

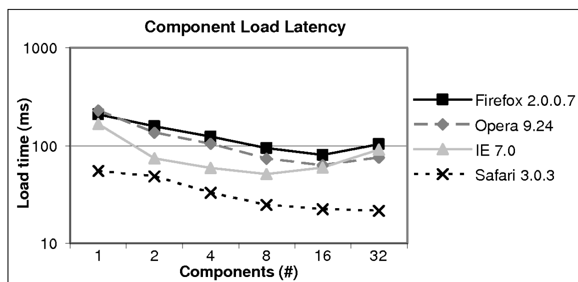

Figure 5: Component loading latency.

ables of the applet to communicate in a secure manner with the mashup application (ensuring that components cannot interfere with each other's communication). The LiveConnect feature in browsers is used to communicate between a component written in JavaScript and the low-level communication layer written in Java. We use polling from JavaScript to Java to detect the reception of a new message. This is necessary since browsers enforce the same-origin policy, and prevent a method call from Java to JavaScript from crossing origins.

Results: On both Firefox and IE there were stability problems in loading large number of applets on the same page, so we limited the throughput tests to pages which had no more than 8 applets, which implies no more than 8 components. For Firefox and Opera we observed aggregate throughput of 16 MB/s, independent of the number of components. For IE, we observed an aggregate throughput of 8 MB/s. Our conclusion is that for browsers that do support Java applets and LiveConnect, componentized mashup applications could use this mechanism to get significantly higher throughput than fragment communication.

6. RELATED WORK

Related work can be largely divided into three groups: proposals working with unmodified browsers; extensions to HTML requiring browser modifications; solutions based on browser plugins. In the following subsection, we will have a closer look at each of them.

6.1 Unmodified Browser

Alternatives for unmodified browsers fall into three categories: (proxied) iframes with client-side communication, (proxied) iframes with server-side communication, and server-side induced JavaScript rewriting or restrictions.

6.1.1 Iframes with Client-Side Communication

Closest to our work are XDDE [13] and the recently announced Google PubSub [8]. Both use iframes and fragments for client-side communication similar to our approach. Neither of them, though, seems to address the integrity and frame-phishing issues raised and addressed in this paper.

A second technique for client-side communication is Subspace [12]. It exploits domain-relaxation of the `domain` DOM attribute, mentioned in Section 4.1, to place a request to a foreign service safely in an iframe served from a one-time-use sub-domain of the mashup server. While their primary goal is secure cross-domain client-server communication, it could also be used for secure cross-domain component communication within the browser. After some setup cost involving multiple iframes, this solution allows exchange of JavaScript objects of arbitrary length and hence scales very well in terms of bandwidth. Depending on the particular browser, though, the event rate will be bounded by the limits of polling similar to our approach. As a drawback, Subspace requires that all connections are setup in the very beginning, prohibiting dynamic on-demand loading of components. [10] More importantly and contrary to our approach, Subspace requires complete trust of the component providers in the mashup provider as their code is executed in DNS domains controlled by the mashup provider.

6.1.2 Iframes with Server-Side Communication

An alternative to the above is to do server-side communication between client-side components that are represented as iframes. Each client-side component has a corresponding server-side communication object. Communication between component A and B would occur by component A sending a message to its server-side communication object which would pass it to component B's communication object, which would then pass it back to component B. This solution requires asynchronous server to client communication, which could be implemented using emerging techniques like the Bayeux protocol [21] and Cometd [24]. We did not choose this approach since it needs a complicated connection setup, and all component-component communication in the client needs to go through two (logical) servers, with increased server load and considerable latency.

6.1.3 JavaScript Rewriting or Restrictions

An alternative to browser isolation mechanisms is to use language analysis and rewriting. A combination of static analysis, language restrictions and dynamic code rewriting could achieve isolation of JavaScript code [20, 29, 25, 16]. However, due to the complexity of rewriting self-modifying code in weakly documented and varied runtime environments, it makes it hard to verify that the isolation achieved is complete. Furthermore, it requires a trusted server, limiting the use to a single administrative domain. Finally, it faces considerable performance and integration challenges on client as well as server side.

6.2 HTML Extensions

There are a number of competing proposals to extend the HTML specifications with goals similar to our secure component model: the `<module>` tag proposal [5], cross-document messaging in the HTML 5 working document of the Web Hypertext Application Technology Working Group (WHATWG) [10] and the `<friv>` element proposal [11].

All three proposals provide an isolated DOM container element – essentially variations of `<iframe>` – which enables cross-domain communication by offering a message passing interface. Invoked in a controlled fashion and assisted by additional meta-data about the caller, the receiving element can then process such messages in a secure way.

While the granularity of isolation in HTML5 is based on the same-origin policy, different `<module>`s and `<friv>`s are isolated even when they are loaded from the same origin. This will simplify administration of servers hosting multiple components — not each of them has to be assigned a

[10] The open-source implementation CrossSafe [30] relaxes this requirement. However, it essentially requires all components to trust each other, clearly unacceptable for many contexts.

separate DNS sub-domain — and encourage finer-granular components. However, it raises the question of how well it integrates with traditional authentication schemes, often tied to the same-origin policy due to cookies.

The key difference from a programming model perspective is that these proposals implement one port for each component, which multiplexes all communication, while we support multiple ports. In addition, we support (multi-cast) channels directly between components, and not just between a component and the parent. This allows us to define fine-grained access control policies, which cannot be directly defined using these proposals. However, as pointed out in Section 4.3, these proposals could easily be integrated into our overall architecture by replacing the fragment communication layer in Figure 3.

These proposals all require browser modifications and hence there will be a considerable time lag before they are adopted by all end-users.

6.3 Plugins

A third communication approach is to use browser plugins to facilitate cross-domain communication. Good candidates are Adobe Flash, Java Applets or Google Gears. An implementations based on these can be more efficient (see Section 5.5). However, the required plugins might not be installed in each browser, or not even available on a given platform, and are mostly based on proprietary technology. That said, as discussed above for HTML extensions, such plugin-based cross-domain communication could serve, where available, as an alternative provider in our lowest communication layer with the fragment-based communication as a guaranteed fallback.

7. CONCLUSION

This paper addresses the problem of securing mashup applications which mix active content from different trust domains. As a solution, we propose a secure component model comprising a central event communication hub and governed communication channels which mediate the communication between isolated components. We illustrated how such a model can be used to enforce basic access control policies which define the allowed interactions between components.

We described SMash, an implementation of this model on current browsers, which can be used right away in building secure mashup applications. Our implementation depends on iframes for isolation while bootstrapping a publish-subscribe model of communication using URL fragment identifiers. Our programming model is intentionally general enough that other communication techniques could be used instead of URL fragments. SMash is resilient to attacks such as channel spying, message forging, and frame-phishing. We have evaluated our implementation and find that it scales well with increasing number of components in the mashup, and has enough data throughput to be useful in a number of mashup application scenarios. Our implementation is available as an open-source JavaScript library.

8. REFERENCES

[1] OpenAjax Alliance Open Source Project. `http://openajaxallianc.sourceforge.net`.

[2] A. Barth and C. Jackson. Protecting browsers from frame hijacking attacks. `http://crypto.stanford.edu/frames/`.

[3] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. Technical Report MSR-TR-2006-120, Microsoft Research, Sept. 2006.

[4] J. Burke. Cross domain frame communication with fragment identifiers. `http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html`, June 2006.

[5] D. Crockford. The `<module>` tag. `http://www.json.org/module.html`, Oct. 2006.

[6] R. Dhamija, J. Tygar, and M. Hearst. Why phishing works. In *Conference on Human Factors in Computing Systems (CHI 2006)*, 2006.

[7] Dojo Foundation. Dojo javascript toolkit. `http://www.dojotoolkit.org/`.

[8] Google. Gadget-to-gadget communication. `http://www.google.com/apis/gadgets/pubsub.html`.

[9] Google. Google account authentication (AuthSub). `http://code.google.com/apis/accounts/AuthForWebApps.html`.

[10] I. Hickson (Editor). HTML 5. Technical report, Web Hypertext Application Technology Working Group HTML 5, 2007. Working Draft, `http://www.whatwg.org/specs/web-apps/current-work`.

[11] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of HotOS XI: The 11th Workshop on Hot Topics in Operating Systems*. USENIX, May 2007.

[12] C. Jackson and H. Wang. Subspace: Secure cross-domain communication for web mashups. In *16th International Conference on the World-Wide Web*, 2007.

[13] G. Lee. Personal communication on XDDE. `http://www.openspot.com`, 2007.

[14] B. McLaughlin. Mastering Ajax. *IBM developerWorks*, 2005 – 2007. `http://www-128.ibm.com/developerworks/views/web/libraryview.jsp?search_by=Mastering+Ajax+Part`.

[15] Microsoft. Windows cardspace. http://cardspace.netfx3.com, http://www.identityblog.com.

[16] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja — safe active content in sanitized Javascript. `http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf`, Oct. 2007.

[17] Mozilla.org. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[18] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[19] D. Raggett, H. Le Arnaud, and I. Jacobs (Editors). HyperText Markup Language (HTML). W3C Recommendation 4.01, W3C, Dec, Dec. 1999.

[20] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[21] A. Russel, D. Davis, G. Wilkins, and M. Nesbitt. Bayeux protocol. Technical Report 1.0draft0, Dojo Foundation, 2007.

[22] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[23] K. Spett. Cross-site scripting — are your web applications vulnerable? Technical report, SPI Dynamics, 2005. `http://www.spidynamics.com/whitepapers/SPIcross-sitescripting.pdf`.

[24] Teknikill, Shadowcat Systems, and SitePen, Inc. Cometd. `http://www.cometd.com/`.

[25] K. Vikram and M. Steiner. Mashup component isolation via server-side analysis and instrumentation. In *Web 2.0 Security & Privacy Workshop*. IEEE Computer Society, Technical Committee on Security and Privacy, 2007.

[26] World Wide Web Consortium. Document Object Model. `http://www.w3.org/DOM/`.

[27] Yahoo! Browser-based authentication (BBAuth). `http://developer.yahoo.com/auth/`.

[28] K.-P. Yee and K. Sitaker. Passpet: Convenient password management and phishing protection. In *Symposium On Usable Privacy and Security*, 2006.

[29] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *34st ACM Symposium on Principles of Programming Languages (POPL)*, pages 237–249, 2007.

[30] K. Zyp. CrossSafe. `http://code.google.com/p/crosssafe/`.