SMash: Secure Cross-Domain Mashups on Unmodified Browsers

Abstract.

Mashups 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. The existing browser security model was not designed for supporting such applications, and therefore they are typically implemented in an insecure manner. 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 security policy. We have developed an implementation of this model that works for all major current browsers, and addresses challenges of communication integrity and component phishing. To the best of our knowledge, this is the first work that achieves the goal of secure mashups without any modifications to current browsers, and hence has the potential to achieve immediate and widespread adoption.

1. Introduction

Web applications increasingly rely on extensive scripting on the client-side (browser) using readily available client-side JavaScript libraries. One of the motivations for this is to enable a browser user experience which is comparable to that of desktop applications. The extensive use of scripting on the client side and programming paradigms such as AJAX [1] 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 application domains, including news websites, which have integrity requirements, and 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 website. The tremendous additional value that can be provided to users by mixing and merging content implies that such applications will eventually be prevalent in domains with stricter data security requirements, like consumer banking sites and enterprise applications. Since the existing browser security models were defined and developed without anticipating such applications, these applications are pushing the boundaries of current browser security models.

Given that the content in a mashup application stems from potentially 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 at the same time. It is clear that, at a minimum we want certain security requirements to be enforced, such as the car dealer 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. This model does not support mashups, where a controlled interaction is desirable. To overcome this 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 from different origins (using `<script>` tags). In both cases, it appears to the browser that the mashup originates from a single site, though

---

[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" [2] usually includes also the port. However, this is not done by all browsers, e.g., IE6 ignores ports when comparing origins!

it contains content from different trust domains. Unfortunately, since the content is considered to be from the same origin, the browser security model allows the content to arbitrarily interact with each other, including reading, writing and modifying the other domain's content. Furthermore, even for situation such as enterprise portals where information comes from the same origin and same administrative domain, (unavoidable) programming errors, e.g., resulting in cross-site-scripting vulnerabilities, mandate secure component separation to achieve "security-in-the-depth". Hence, the "all-or-nothing" browser security model [2] is clearly an undesirable situation.

In this paper we examine the problem of building secure componentized mashups. We propose an abstraction that allows security policies to be specified and a design and implementation that realizes this abstraction in *unmodified* browsers. By creating a higher level abstraction for cross-component communication, we have ensured 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 have tested our approach on Internet Explorer (IE), Firefox, and Opera[3]. To the best of our knowledge, this is the first approach that works without browser modifications. There are multiple proposals for HTML and browser modifications to realize secure mashups, however the long timeline of adoption by standards committees, browser vendors, and eventually by users, makes these unviable for anyone wanting to build secure mashups in the near term. We discuss these proposals in detail, in related work.

Our abstraction involves encapsulating content from different trust domains as *components* running in a browser. Components are wired together using *channels*. The channel abstraction and security policies associated with channels are implemented by an *event hub* that is part of the Trusted Computing Base (TCB). The only inter-component communication in the browser is realized using these channels.

We realize the component abstraction using the HTML `<iframe>` element [3], which was designed as a container for loading sub-documents inside the main document. Some technical challenges that we address 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 component *phishing*, where an untrusted component in a mashup can change which component is loaded in another part of the mashup.

We believe that using our abstraction of components which communicate through a mediated event hub, using channels, will provide a key primitive for secure componentized mashups. First, the developer can use a high-level abstraction which is natural to build mashups. Secondly, the use of this abstraction will substantially enhance the security of 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 as with JSONP [4]. Thirdly, it provides a general security mechanism to guard against cross-site scripting (XSS) attacks, as discussed next.

From the long lists of web-based threats, attacks and vulnerabilities that are being published every day [5]-[7], XSS attacks [8] are getting significant attention. These affect a special case of a mashup, where a website combines content generated by the website with content generated by its users, which are different trust domains. XSS attacks leverage the fact that the browser views all this content to be from the same origin, and allows user-generated malicious content to not just read/write website-generated (trusted) content, but also gives it access to browser-cached credentials (for example, cookies) for that origin. 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)

---

[3]When referring to Internet Explorer, Firefox, and Opera throughout this paper, we are referring to Internet Explorer 7.0, Firefox 2.0.0.4, and Opera 9.20 respectively. In addition, there is no technical reason the library would not work on other browsers such as Safari, however this has not been fully verified.

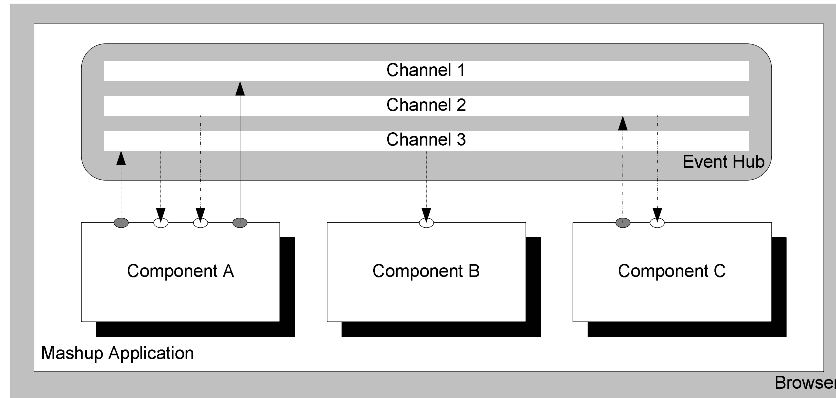

Figure 1. Secure component model for mashups

limiting for user creativity. Using our component abstraction, user-generated content can be contained 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 the requirements and design of our implementation of the secure component model, and how we address various attack vectors, in Section 4. We discuss alternative techniques for implementing the security requirements in current browsers, and motivate the choices we made, in Section 5. Section 6 discusses related work, and we 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 higher level programming interface that allows creation of secure mashups, and makes it easy.

Both from a programmability and usable security perspective, a desirable feature in this interface would be to hide interaction patterns of reading, writing and modifying across trust domains, while providing mediated interaction at the service level. Clearly, for modularity, a component from one trust domain should avoid having to know about the internals of a component from another domain. Thus, our model is analogous to using a message-passing interaction style instead of a shared-memory style. We can then define a centralized communication mechanism which allows governed communication channels to which policies can be attached.

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 governed 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*. Components could be visible or invisible. Visible components share the browser window in a manner determined by the mashup application. A component contains content (both data and code) 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 interaction channels implemented by the event hub. The components trust the mediator (the mashup application and the event hub, both of

which come from the mashup provider) and do not need to deal with authenticating the other components sharing the channel. 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 correctly. The event hub wires the ports of components to the appropriate channels (depicted by the arrows in the figure).

The event hub is a publish/subscribe system with many-to-many channels on which messages are published and distributed. In Figure 1, Component A can publish to Channel 1 and Channel 3 but is only subscribed to Channel 2 and Channel 3.

The mediated channels form the last part of the model. They allow the different trust domains to interact with each other through the trusted event hub. 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.

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 the security policy that will be enforced by the model.

Defining permitted interactions between components based on policy is particularly important to enable the mashup provider to declaratively specify desired interaction patterns. Furthermore, such declarative policy specification can be derived from many sources of high level policies. This is especially true for enterprise-class mashups [9], where the security policy may be a combination of enterprise-level policy, department policy, and end-user policy. Even component providers may want to express how their components are supposed to be integrated into a mashup application. For example, a component provider may constrain what a mashup provider may do with the data exposed by the component. 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.

For an example of a set of policies that could be specified and enforced, consider a mashup with three components: Component A providing banking services, Component B providing car information, and Component C providing advertisement for insurance products. When the user decides to buy a car, the car component B is allowed to supply detailed information for creating a bank transfer to the banking component A, and very general information about the type of car to the insurance advertisement component C. In no case should the insurance component be allowed to interact directly with the banking component.

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. The exploration of this high level policy space, the policy languages, and the derivation of the basic access control policy based on high level policies is outside the scope of this paper.

3.1 Basic Access Control Policies

Basic Access Control Policies specify 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. These policies are enforced by the event hub. The subjects in these policies 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 mashup provider does not have to deal with explicit policy specification or policy languages. 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. For example, the implicit wiring policy for the mashup application in Figure 1 is described in the table below.

Component	Channel 1		Channel 2		Channel 3	
	pub	sub	pub	sub	pub	sub
Component A	x			x	x	x
Component B						x
Component C			x	x		

Another alternative would be for the event hub to read security policies from a configuration policy file. This policy file could possibly be provided outside of the mashup application, potentially even by third parties. In this scenario, when input ports are being wired to channels (i.e., components are being subscribed) or output ports are being wired to channels (i.e., components are given permission to publish) the event hub checks if this is an allowed interaction according to its 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. A policy file for the example in figure 1 can be expressed in the JSON format [10] as follows:

```
{
  "EventHub" : {
     "Channels" : [
        { "name" : "Channel 1",
          "pub" : "Component A",
          "sub" : ""},
        { "name" : "Channel 2",
          "pub" : "Component C",
          "sub" : "Component A, Component C"},
        { "name" : "Channel3",
          "pub" : "Component A",
          "sub" : "Component A, Component B"},
     ]
  }
}
```

4. Implementing the Secure Component Model

In this section, we discuss our design and implementation of the secure component model. The core ideas underlying our approach is to leverage the current browser security model to isolate components belonging to different trust domains and then to allow interaction by publishing-subscribe to named channels which are managed by the mashup application. Our approach does not require any browser modifications, hence adoption can be immediate. Section 4.1 describes security and usability requirements that constrain our design. We will revisit these requirements in a later section, when comparing our security enforcement approach with other potential options. Section 4.2 gives an overview of the solution components and underlying communication primitive used in our prototype. Section 4.3 discusses the channel communication mechanisms and the secure hub abstractions. In Section 4.4 we evaluate the security of our solution with a

discussion of a number of ways in which one could attempt to attack our model and communication primitives.

4.1 Security and Usability Requirements for the Implementation

Prior to describing the details of the prototype for isolation and secure communication, we would like to informally write down the requirements which the prototype aims to fulfill. The motivation for this is (1) to guide our design and implementation, (2) to compare with other design alternatives (discussed in later sections), and (3) to allow other developers to make an informed choice in choosing a different implementation approach for the secure component model, depending on their security needs and environment. The following are the requirements we tried to cover in our prototype:

- The DOM (Document Object Model) [11] tree of each component is completely isolated from components from other domains. This specifically means that no reading and writing of DOM elements belonging to cross-domain components is possible.
- The JavaScript namespace of each component is completely isolated from components of other domains. This means that no or only governed reading and writing of JavaScript objects across components is possible.
- The inter-component communication is secure. That is, a malicious component can not affect the integrity and confidentiality of communication between two other components.
- A component cannot launch denial of service attacks against communication between two other components.
- Component phishing can be detected by the mashup application.
- Initiation of component loading and channel access control is completely under the control of the mashup application.
- The mashup application can dynamically create and delete components and create and delete channels. In this context, *dynamic* means that these changes happen after the user has started interacting with the application. In contrast, *static* would mean that component and channel creation can only happen when the application first loads into the user's browser, as part of the initialization process.
- Components are loadable directly from the component provider. This means that no additional web proxy is required and the component is protected (isolated) even from the mashup application code.
- It is possible to implement the technology safely on all common browser platforms.

4.2 Solution components and underlying communication primitive

In this section, we will describe the components to be used in our solution for secure interaction. Before we do this we need to review a few basic concepts from HTML documents and their properties and understand the current "same-origin" security policy of current browsers. An HTML document has a `domain` property that is the hostname of the server it was accessed from, and a `location` property that represents the URL of the document. The hostname could be a numeric IP address or DNS 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. It is possible to change the `location` the `frame` in which a document is loaded. This will cause a new document to be loaded to replace the current document. `Frames` can include sub-`frames` using HTML `<frame>` and `<iframe>` tags where each `frame` has its own document with `domain` and `location` attributes. The "same-origin" policy says that documents from different origins 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 are part of the same `frame` hierarchy. However, code can write the `location` property of any `frame` in the same `frame` hierarchy, regardless of origin.

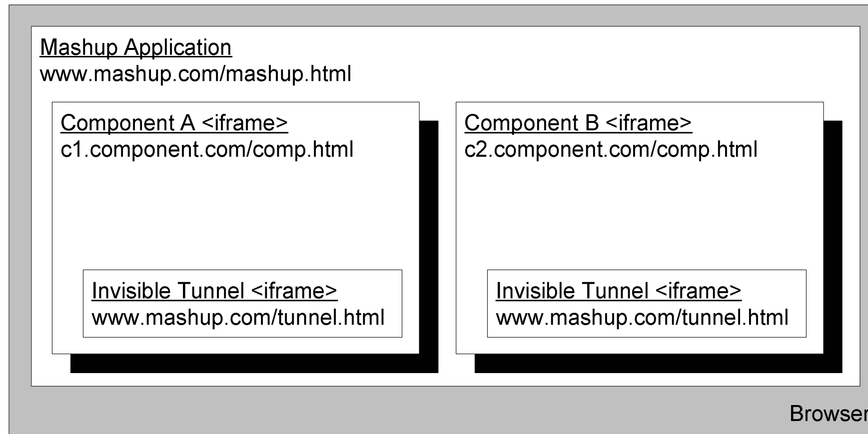

Figure 2. Fragment communication architecture

In our prototype, we load components which come from different trust domains into different `<iframe>s`, i.e., they represent different sub-frames. Even in the case where a single host is serving components from multiple trust domains, isolation can be obtained by having the host use multiple DNS names, one for each trust domain. However, in light of the fact that components can change their `domain` properties as described earlier, the DNS domains used should be 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.

While isolation can be easily achieved by placing components in different `<iframes>s`, the challenge is enabling cross-domain inter-component communication, since it is not explicitly supported in browsers. Recently, an approach to communicate between `<iframe>s` using the fragment identifier of the URL of the `<iframe>` has been discovered [12]. The communication is based on the observation that even though the parent and child `<iframe>s` have different origins, the parent can write to the child's `window.location` property. Note that if we only modify the fragment identifier of the `location` property the document will not be reloaded and hence the application 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 [13] to circumvent the same-origin policy for client-server communication.

Thus the fragment identifier communication method can be leveraged to create a communication link between the mashup application and a component, which can then be used to create end-to-end inter-component channels. However, we note that the fragment communication as such does not inherently guarantee integrity of the communication links, and suffers from component phishing vulnerabilities. A number of such issues and the technical challenges of guaranteeing integrity will be discusses and addressed in Section 4.4.

In the prototype implementation we leverage the separation provided by `<iframe>s` to realize isolation of the DOM tree and the JavaScript namespace of the components. One of the challenges for an architecture based on `<iframe>s` is to organize the `<iframe>s` in such a way that they provide isolation while still allowing fragment communication. The extent to which current browsers can navigate the frame structure is browser dependant and tends to differ quite a lot. For example, suppose the mashup application is in origin A and it has two child `<iframe>s` in origin B and C. On Firefox the `<iframe>` in origin B would be able to navigate the frame structure to get access to the `window` object of the `<iframe>` in origin C. On Opera however, this would not be permitted since Opera only allows navigating to ancestors and descendants. A complete discussion on the behavior of cross-domain frame access for the most common current

browsers is given in [14]. In this paper, we choose an `<iframe>` configuration which works all the common browsers.

Figure 2 illustrates an example of the `<iframe>` configuration that we use for three different trust domains. The first origin, `www.mashup.com`, is the origin of the mashup application which we assume is a trusted application. To leverage the `<iframe>` isolation, the components are loaded into two addition 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 `<iframe>s` communicate cross-domain using fragment identifiers. Note that this also makes sure that the messaging does not interfere with the mashup application.

For cross-domain communication, the components and the tunnel `<iframe>s` poll for incoming messages which are communicated through the fragment part of their URL and write to the fragment part of the URL of their communication partner. For example, if the component A wants to send a `message` to the mashup application, the component modifies the URL of the tunnel to `www.mashup.com/tunnel.html#message`. The tunnel `<iframe>` which is polling for modifications in its fragment notices this and delivers the message to the mashup application, which it can access through `window.parent.parent` since it is in the same origin as the tunnel. The next section discusses how this can be extended to the event hub communication.

4.3 Higher level abstractions for communication

In the previous section, we described how a component in an `<iframe>` could communicate with its tunnel. However, this basic communication primitive is not sufficient for fully realizing our secure component model. We would like to layer over this primitive a broader cross-domain abstraction along with policies governing the inter-component communication and a cleaner programming abstraction which hides the details of the underlying primitive. This is desirable not only from a usability perspective but it can also be extended to work with future browser which more readily support cross-domain communication. This higher layer is represented by the event hub in our secure component model. Communication at this level is realized by multiplexing different event hub communication channels over the single fragment communication channel. In our prototype we opted for a layered approach for realizing this so that in the future on of the layers could be replaced by a more secure version as new technology becomes available on the browser platforms. Section 5 and Section 6 discuss a number of alternatives and proposals for cross-domain communication technologies.

Figure 3 pictorially depicts the layers of the prototype: the event hub layer, the event communication layer and the fragment communication layer. The upper layers in the stack abstracts away from the low level details of the core fragment identifier communication. As in standard multi-layered protocols each layer provides additional functionality by wrapping messages as payloads of the layers below it. Note that since the JavaScript namespaces of the communication partners are distinct the three layers are present on both sides of the communication.

The uppermost layer is the *Event Hub layer* which is part of the mashup application. On the mashup side, in this layer, the mashup developer wires the ports of the components to the different channels of the event hub so that a governed environment for cross-domain communication is created. It is also in this layer that the policies are defined and enforced. At the component side, the component developer defines the input and output ports of his component in this layer. Listing to the input ports and publishing on the output ports is realized in this layer.

The middle layer is the *Event Communication* layer. This layer is responsible for composing the messages which are used to multiplex the multi port components on the single channel fragment identifier messaging.

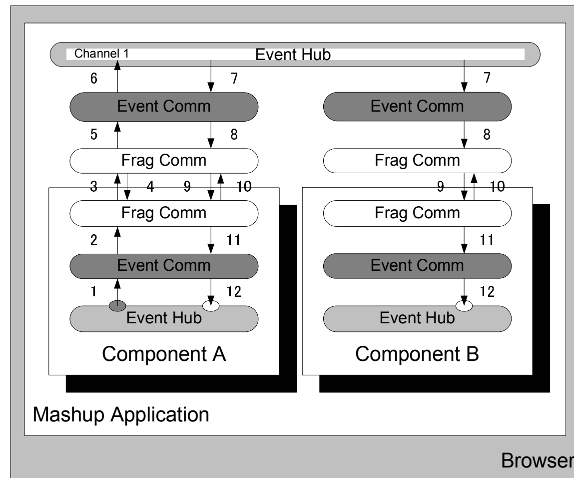

Figure 3. Layered component communication

The lowest level layer is the *Fragment Communication* layer. This layer hides all the implementation details of the single channel communication mechanism from the event communication layer. In this layer, the polling and fragment signaling is implemented. Note that this is the only layer that needs to be replaced when the component model is implemented using a different cross-domain communication technology.

To illustrate how the layered approach works in the prototype implementation, we consider an example with 2 components A and B and follow the sequence when Component A publishes a message on a port `out1` which is then consumed by port `in1` in both A and B. The following is the 12 step sequence a message goes through. We assume that the ports `out1` of Component A is wired to `channel1` on the event hub which is subscribed through port `in1` by both components.

1. Component A sends data on port `out1`, provided by the event hub at the component side. In the example, the event hub layer tells the event communication layer that it wants to publish `Hi 1` on `out1`.
2. The event communication layer wraps this data into a string and instructs the fragment communication layer to deliver the message to the mashup application. The encoded message looks like this: `type%3Dpublish%26port%3Dout1%26payload%3DHi%201`. This data can be interpreted as the message is of the type `publish` and needs to be delivered to the port with name `out1` and with the payload `Hi 1`.
3. In the fragment communication layer this message is wrapped into another message which is used for delivery over the point to point connection between the tunnel `<iframe>` and the component `<iframe>`. The final message which will be written as a fragment identifier of the tunnel `<iframe>` in Component A is: `end:3:type%3Dpublish%26port%3Dout1%26payload%3DHi%201`. This message is of the type `end` which indicates that the message is self contained[4]. It has the sequence number 3. Sequence numbers are used for differentiating between messages. This allows the polling mechanism to see two different messages even if they have the same payload. It has a payload equal to the message passed to it by the event communication layer.
4. After the fragment communication layer at the mashup application side received the message from the other fragment communication layer, it replies with an acknowledgement that it received the data correctly. The message used for this is `ok:3`.
5. After sending the acknowledgement, the fragment communication layer decodes the incoming message and delivers the message to the event communication layer.

[4] Due to restrictions on the length of a URL in browsers this is not always the case.

6. The event communication layer decodes the next layer and detects that it is a publish message on `out1` and extracts the payload of the message. Afterwards it delivers the publish command for `out1` to the event hub layer.
7. The event hub layer knows that `out1` is wired to `channel1` in the event hub. In the event hub a lookup for the subscribed components on `channel1` is done, and a `distribute` message is issued to the event communication layer which targets the `in1` ports of Component A and B.

8-11. The delivery of the `distribute` message from the event communication layer at the mashup application side to the component side is identical to the delivery of the `publish` message as discussed in step 2 to step 5.

12. In the final step, the event hub layer delivers the message to the components. The components can receive this message by registering a callback for processing incoming messages on `in1` to the event hub layer at the component side.

The example sequence described above illustrates quite well the core ideas of how the layers in our prototype interact with each other to secure inter-component communication. We have omitted a number of details in the description of the layers and we defer those details to a future publication.

4.4 Attacking Fragment Identifier Messaging on Permissive Browsers

In this section, we discuss the security issues we identified with the integrity of the cross-domain communication links when implemented by fragment identifier messaging. As discussed in Section 4.2, different browsers have different behavior when navigating through the frame structure of a web application. Some of the browsers, called permissive browsers in [14], allow full frame navigation even though the frames are actually coming from different origins. Well known browsers that support this are Firefox, Safari and some configurations of Internet Explorer 6 and Internet Explorer 7. This can lead to several vulnerabilities and the loss of the integrity of the fragment identifier communication. We discuss a few such issues in the remainder of this section. The fundamental issue is that with permissive browsers it is possible to access the `window` object of any frame inside the mashup application. For example in Figure 2, this means that the Component B can get direct access to the `window` object of Component A even though they both come from different origins. Note that even though one can get access to the window object, it is not possible to read the window.location property of the `<iframe>` not in the same origin. This definitely prevents a component from spying on the communication between a components and the secure hub. However there are other exposures:

4.4.1 Forging Message Origin

A potential attack is to insert a spurious message, in effect forging a message, by writing to the `window.location` of another component's `<iframe>` or one of the tunnel `<iframe>`s. Since browsers permit writing it is not possible to prevent cross-domain writing by malicious components. For example, when a malicious Component B wants to send a message to the event hub on behalf of Component A, it traverses the frame structure to the tunnel `<iframe>` associated with Component A. Then it writes its message to the tunnel's `window.location`. The tunnel `<iframe>` will detect this incoming message but will not be aware of the fact that this is a forged message which did not originate from Component A.

To counter this attack we can extend the fragment communication protocol with an additional security token, i.e., an unpredictable random value, which is transferred through the different layers of the communication protocol and used to verify the sender of the message. Upon creation of the component, the mashup application generates a unique security token which is passed to the component as part of the initialization parameters. When the component creates the tunnel `<iframe>` it passes the token in the same way. After receiving the token, the tunnel `<iframe>` provides the token to the mashup application which verifies it if the received token is the same as the originally generated token.

When the token is the same, it can be used in all future communication between the tunnel and the component to make sure that no malicious component is forging messages. Note that after verification, it is sufficient to use the token at the fragment communication layer since this is the only layer at which a malicious component can attack. If the token is not the same, a malicious component has been interfering with the initialization process of the component and the attacked component can be reinitialized until a verified shared token exists, hence until integrity can be assured.

It is not possible for a malicious component to read the security token of other components because it can not read the `window.location` property across origins. Nor does it have access to the token in the JavaScript namespaces of the component, mashup application, or tunnel since they are coming from different origins and protected by the same-origin policy. However, a malicious component could inject its own security token into another component by overwriting the target component's initialization parameters before they were read by the target component. This injection is not a problem since it will be detected by the mashup application when it compares the generated token with the forged one. Upon detection of a forged token during the initialization phase, the component can be reinitialized as described earlier. Once the verification is successful, the origin of the sender of messages can be assured at all time.

4.4.2 Phishing a Component

Since it is possible for a malicious component to write to the `window.location` property of another component's `<iframe>`, it is possible for the malicious component to navigate a component away from its location to a location with a phished component. Since the URLs loaded by `<iframe>s` are not visible in the browser, such an attack can not be graphically detected. Therefore the mashup application will need to prevent this kind of attacks.

For this, the mashup application must be able to detect changes in the location of the component. Since the mashup application can not read the `window.location` property of its components it is not possible to simply poll for changes in this location. Nor is it possible for the mashup application to attach event handlers for detecting changes of this property.

Due to the fact that the tunnel `<iframe>` is a direct child of the component `<iframe>` it will be unloaded whenever the component is unloaded. Therefore, whenever the malicious component modifies the `window.location` of another component to a phished location, the tunnel `<iframe>` of the original component will be unloaded. The tunnel `<iframe>` and the mashup application are loaded from the same origin so they therefore communicate directly with each other. By registering event handlers for the unloading of the tunnel `<iframe>`s of components and providing feedback to the mashup application when this occurs, it is possible to detect phishing, prevent the phished component from being loaded and potentially stop the application from further execution. A final twist is the case when the component is replaced *before* the tunnel `<iframe>` is loaded. This case is handled by the hub which verifies that the secure token handshake really takes place via on `onload` events and timeouts.

4.4.3 Denial of Service

Another way to attack the fragment identifier messaging system is by launching a Denial of Service (DoS) attack against it. The simplest way to realize this would be for a malicious component to make the target component navigate away from its location. This type of DoS attack can be detected and prevented in the same way as the phishing attack can be prevented. An alternative attack would be for a malicious component to flood another component with messages. This would prevent legitimate messages to reach their destination and therefore prevents the mashup application from functioning correctly. Another strategy an attacker could follow would be to create an endless loop in a component. Due to the single threaded nature of JavaScript in the browser environment no other scripts would ever be executed. This list of potential DoS attacks is most likely not complete but serves as an illustration that it is not possible to prevent DoS attacks

among components on the current browser platforms. If such prevention is necessary, browsers will need to be modified.

5. Alternative Implementation Techniques for Unmodified Browsers

In this section, we discuss other alternatives for implementing the secure component model without browser modifications, and motivate the choice we made. We consider three alternatives (1) Subspace [14] (proxied `<iframe>s` with client-side communication), (2) proxied `<iframe>s` with server-side communication, and (3) JavaScript rewriting [15]-[17].

5.1 Subspace (Proxied `<iframe>s` with client-side communication)

Subspace [14] is a technique for secure cross-domain client-server communication. While the goal of this work was not secure cross-domain component communication within the browser the underlying technique can be used to address this problem. Subspace makes extensive use of `<iframe>s` to realize a hierarchy of documents, and relaxing the `domain` attribute of these documents to enable cross-domain exchange of JavaScript objects. It also relies on the fact that JavaScript objects can be passed into other `<iframe> s` when they are part of the same origin and remain accessible if this is no longer the case.

Subspace allows cross-domain communication while still preserving the safety introduced by loading different origins into different `<iframe>s`. Consider the scenario with a mashup application in `www.mashup.com` and an untrusted service that is to be loaded through `service.mashup.com`. First, the mashup application creates a mediator `<iframe>` in the domain `www.mashup.com`. As the two `<iframe>s` are in the same origin, it is possible to share a JavaScript object between them: the mashup application passes a callback object from its own `<iframe>` (1) to the mediator `<iframe>` (2). Next, the mediator `<iframe>` relaxes its origin to `mashup.com` and loads the untrusted `<iframe>` from `service.mashup.com which` also relaxes its origin to `mashup.com`. The mediator `<iframe>` can now pass the callback object from its own context (2) into the untrusted context (3). At this point a link between the mashup application and the untrusted `<iframe>` is created. The untrusted `<iframe>` can now load potentially malicious content from another origin, using a `<script>` tag, and safely communicate the result back to the mashup application.

In order to support the secure component model, isolation of components in Subspace relies on the use of `<iframe>s` from different origins. Due to the design of Subspace, it is not possible to directly load components from their origin, and therefore one needs to use a proxy. Insofar, the component provider would also have to completely trust the mashup provider whereas in our approach the component code and data can be protected even from the mashup application with appropriately secured component loading. Another significant drawback of this approach, in our context, is that the mashup application cannot dynamically create components. This is because the origin relaxation technique, to pass the callback object, is secure only if all callbacks (for all components) are setup before *any* potentially malicious component content is loaded into the browser.

5.2 Proxied `<iframe>` s with server-side communication

An alternative to the Subspace approach is to do server-side communication between client-side components, that are represented as `<iframe>s`. 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 circumvents cross-domain communication restrictions in the browser. A server proxy has to serve components that are in different trust domains from different DNS domains. The communication objects for all components would be located at the proxy. This

solution requires asynchronous server to client communication, which could be implemented using emerging techniques like the Bayeux protocol [18] and Cometd [19]. We did not choose this approach since it needs a proxy, and all component-component communication in the client needs to go through the proxy server, which is not efficient.

5.3 JavaScript Rewriting

An alternative to browser isolation mechanisms is to use language analysis and rewriting. This approach is challenging due to the self-modifying nature of JavaScript, which makes a pure static analysis solution ineffective. A combination of static analysis and dynamic code rewriting could be used, and there is some work on this in the literature [15]-[17]. These works does not necessarily address component isolation, but could be extended or specialized for this purpose. Some drawbacks of this approach are (1) the complexity of rewriting self-modifying JavaScript, which can make it hard to verify that the isolation achieved is complete, (2) the need for a trusted rewriting proxy, that rewrites all the component code, and (3) performance of rewritten JavaScript (some microbenchmarks in [15] show a slowdown of over 300x compared to the original JavaScript code).

6. Related Work

In this section, we discuss three proposals for browser modifications with goals similar to our secure component model (1) the `<module>` tag proposal [20], (2) cross-document messaging in the HTML 5 working document of the Web Hypertext Application Technology Working Group (WHATWG) [21], (3) the FRIV element proposal [22], and (4) a proposal [23]. These proposals all require HTML and-or browser modifications and hence there will be a considerable time lag before they are adopted by a standards committee implemented in browsers, and be widely adopted by users. Finally, we discuss an alternative finer-grained security model for mashups that involves access control on shared state. The security abstractions described in these proposals can be used to bootstrap our secure component model, by layering a JavaScript library over these abstractions.

6.1 The <module> tag

The `<module>` tag [20] is a proposal for a new tag in HTML, which defines a module with isolation similar to <iframes> but with communication support for parent child communication. The communication interface between the `<module>` and its parent document is a very simple messaging-based API, over which JSON [10] formatted messages are sent. At the parent's side, there is an object which allows sending and registering a callback for receiving; at the `<module>` side there is a similar mechanism.

Modules are similar to components in our model. The key difference from a programming model perspective is that the `<module>` tag implements one port for each module, which multiplexes all communication, while we support multiple ports (and consequently multiple channels per component). In addition, we support channels directly between components, and not just between a component and the parent document/mashup application. This allows us to define fine-grained access control policies on channels, which cannot be defined with the `<module>` tag. On the other hand, different `<module>s` are isolated even when they are loaded from the same origin. This simplifies administration of servers hosting multiple components as not each of them has to be assigned a separate DNS sub-domain.

6.2 Cross-document Messaging in WHATWG Proposal

WHATWG (Web Hypertext Application Technology Working Group) has recently introduced a new proposal for HTML 5 [21], which includes support for cross-document messaging, where documents could be from different origins. Conceptually, the proposal is similar to the `<module>`

tag proposal. Instead of proposing a new isolation abstraction like the <module>, this proposal leverages the current abstraction of a document, where documents can be arranged in a hierarchy. The HTML 5 proposal allows a document to post a message to another document, regardless of document origin. The receiving document has to decide whether it trusts messages from the origin of the sender.

Unlike our secure component model, this proposal does not define fine-grained channels between components, and requires each component to perform its own access control on received messages. Furthermore, unlike the <module> tag, DOM and JavaScript resources are shared based on the same origin policy. Hence, this proposal would still require a separate DNS domain per component.

6.3 FRIV element

This is a proposal to add a new element to HTML, FRIV [22], which combines the isolation properties of an <iframe> with explicit support for secure communication between the FRIV and its parent. Like an <iframe>, FRIV elements can be nested. The FRIV element offers some usability benefits over <iframe>, by allowing the FRIV to read the size of its display region. The proposal seems to share a lot of similarity to HTML 5. However, due to the difference in level and style of specification, it is hard to judge the exact differences of the two proposals.

6.4 DOM Level Access Control (DOMLAC)

Unlike the previous proposals, which support a message-passing interaction model, the DOMLAC [23] proposal supports a shared-state model. It provides fine-grained access control on read, write and traverse actions of the DOM tree of a web application. Policies can be associated with parts of the DOM tree inside a web page, which could be used to safely isolate the DOM sub-tree of each component. And therefore prevent potentially malicious parts of the DOM tree to access other parts.

Secure communication links between a component and the event hub could be achieved by creating communication zones in the DOM tree and associating policies with them such that only the component and the event hub can access and modify that zone. DOMLAC is implemented as a browser plugin.

DOMLAC does not consider separation of the execution context of JavaScript in the different zones (zones are analogous to trust domains). This makes it possible to steal information stored in variables in the component's execution context. DOMLAC also does not support loading of components directly from their original host (the same proxy must be used for all components used in a mashup application).

7. Conclusion

In this paper, we considered the problem of mediated communication between components which are co-resident on the end-users browser while coming from different trust domains. This problem is increasingly becoming important due to the prevalence of mashup applications. The current browser same-origin security policy is not directly amenable to supporting this scenario. To address this issue, we described a secure component model which consisted of 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 of components.

We have described a prototype which implements this model on current browsers and hence can be used right away in building secure mashup applications. While we think our approach is the best generic solution working with unmodified browsers, it is worthwhile pointing out that our programming model is intentionally general enough that other implementation techniques mentioned in Section 5 and 6 could be used by servers on a mix and match basis to optimize

requirements for special cases. For example, a mashup application server could use code-rewriting techniques for co-located components to save itself from having to manage a separate DNS domain for each component or exploit browser extensions supported by a particular client for increased performance.

We evaluated the security of the prototype by studying various attacks such as message forging, component phishing and Denial of Service attacks. To the best of our knowledge, this is the first work that achieves the goal of secure mashup components without any modifications to current browsers.

Due to the increasing importance of this problem, there are a number of proposals for extensions of the browser security model to support secure mediated communication. Our solution can be used directly in today's browsers or can be complemented with the potential browser extensions to leverage the enhanced security they could provide.

8. References

[1] Brett McLaughlin, "Mastering Ajax," http://www-128.ibm.com/developerworks/views/web/libraryview.jsp?search_by=Mastering+Ajax

[2] Jesse Ruderman, "The Same Origin Policy," http://www.mozilla.org/projects/security/components/same-origin.html

[3] World Wide Web Consortium, "HTML 4.01 Specification - Inline frames: the IFRAME element," http://www.w3.org/TR/html401/present/frames.html#h-16.5

[4] Bob Ippolito, "Remote JSON – JSONP," http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/

[5] Web Application Security Consortium, "Web Security Threat Classification," http://www.webappsec.org/projects/threat/

[6] Open Web Application Security Project, "Attacks," http://www.owasp.org/index.php/Category:Attack

[7] SecurityFocus, "Bugtraq mailing list," http://www.securityfocus.com/archive/1

[8] Kevin Spett, "Cross-Site Scripting - Are your web applications vulnerable?," http://www.spidynamics.com/whitepapers/SPIcross-sitescripting.pdf

[9] IBM, "QEDWiki," http://services.alphaworks.ibm.com/qedwiki/

[10] D. Crockford, RFC 4627 The application/json Media Type for JavaScript Object Notation (JSON)

[11] World Wide Web Consortium, "Document Object Model," http://www.w3.org/DOM/

[12] James Burke , "Cross Domain Frame Communication with Fragment Identifiers," http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html

[13] Dojo Foundation, "Dojo JavaScript toolkit," http://www.dojotoolkit.org/

[14] C. Jackson, H. J. Wang, "Subspace: Secure Cross-Domain Communication for Web Mashups," WWW 2007, pp. 611-619, Canada, May 2007.

[15] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir "BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML" Usenix OSDI, Seattle, WA December 2006

[16] M. Steiner and K. Vikram, "Mashup Component Isolation via Server-Side Analysis and Instrumention," W2SP2007: Web 2.0 Security and Privacy 2007, May 2007.

[17] D. Yu, A. Chander, N. Islam, I. Serikov, "JavaScript Instrumentation for Browser Security," Proceedings of the 2007 POPL Conference, pp. 237 – 249, January 2007.

[18] Dojo Foundation, "Bayeux Protocol," http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html

[19] Teknikill, Shadowcat Systems, and SitePen, Inc, "The Scalable Comet Framework," http://www.cometd.com/

[20] D. Crockford, "The <module> tag," http://www.json.org/module.html

[21] Web Hypertext Application Technology Working Group, "HTML 5 - Cross-document messaging," http://www.whatwg.org/specs/web-apps/current-work/#crossDocumentMessages

[22] J. Howell, C. Jackson, H. J. Wang, and X. Fan, "MashupOS: Operating System Abstractions for Client Mashups," 11th Workshop on Hot Topics in Operating Systems, May 2007.

[23] N. Uramoto, S. Yoshihama, F. De Keukelaere, "OpenAjax Security Work Session," http://www.openajax.org/member/wiki/images/0/0c/2007_March_Members_Meeting_Ajax_Security_Threats.pdf