OMash: Enabling Secure Web Mashups via Object Abstractions

Steven Crites
University of California, Davis
crites@cs.ucdavis.edu

Francis Hsu
University of California, Davis
fhsu@cs.ucdavis.edu

Hao Chen
University of California, Davis
hchen@cs.ucdavis.edu

ABSTRACT

The current security model used by web browsers, the Same Origin Policy (SOP), does not support secure cross-domain communication desired by web mashup developers. The developers have to choose between *no trust*, where no communication is allowed, and *full trust*, where third-party content runs with the full privilege of the integrator. Furthermore, the SOP has its own set of security vulnerabilities and pitfalls, including *Cross-Site Request Forgery*, *DNS rebinding* and *dynamic pharming*. To overcome the unfortunate tradeoff between security and functionality forced upon today's mashup developers, we propose OMash, a simple abstraction that treats web pages as objects and allows objects to communicate only via their declared public interfaces. Since OMash does not rely on the SOP for controlling DOM access or cross-domain data exchange, it does not suffer from the SOP's vulnerabilities. We show that OMash satisfies the trust relationships desired by mashup authors and may be configured to be backward compatible with the SOP. We implemented a prototype of OMash using Mozilla Firefox 2.0 and demonstrated several proof-of-concept applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Languages, Security.

Keywords

Web, browser, mashup, same origin policy, communication, protection, security model, object abstraction.

1. INTRODUCTION

Web browsers are an integral part of everyday computing, with their uses ranging from accessing simple web pages to accessing web applications such as online retail, banking, webmail, wikis, blogs, and discussion boards. The rise of AJAX (**A**synchronous **J**avaScript **a**nd **X**ML) style web applications has given the web dynamic, interactive content such as Google Maps [2]. In the case of Google Maps, Google has supplied a public API [3] so that web developers can use Google's service to create hybrid websites, such as HousingMaps [4] which takes data from Craigslist [1] and presents it using Google Maps. This type of hybrid web application is known as a *mashup*: a website that combines content from one or more websites.

Even before the rise of AJAX and mashups, at any given time a web browser likely contained pages from different domains, whether they were loaded in different browser windows, tabs, or even frames within another page. As such, security is extremely important: without security it would be possible for a script from one web page to steal or modify critical information from another page. The solution, first presented in Netscape Navigator 2.0, is known as the Same Origin Policy (SOP) [24]. It is now the de facto security model used by web browsers.

1.1 Problems with Same Origin Policy

The Same Origin Policy "prevents document or script loaded from one origin from getting or setting properties of a document from a different origin" [24]. Furthermore, "[T]wo pages are considered to have the same origin if the protocol, port (if given), and host are the same for both pages." A "page" refers to a browser window, `<frame>`, or `<iframe>` (as well as `<object>`), as each can contain a document. Note that a script's origin is considered to be the origin of the document that contains it rather than where the script is hosted. This has important security implications, as a malicious third-party script will run in the context of the site that included it via a `<script src="url_of_script">` tag.

Under this policy, a Domain Name System (DNS) domain is a *principal*, much like a user or group is a principal in a multi-user operating system. While this suffices for mutually distrusting web pages, it is insufficient for creating web applications with content from different domains: the only two trust relationships available between principals are *no trust*, where third-party content is isolated within a different window or frame, and *full trust*, where the third-party content is included as a library via a `<script>` tag (this is the case with Google Maps) and thus has full access to the page that included it. Faced with this coarse-grained security model, developers are often forced to make a tradeoff between security and functionality.

In addition, the SOP suffers from various vulnerabilities due to either its design limitations or its reliance on insecure services. For example, it relies on the security of the Domain Name System, and so where DNS deployment is insecure a *dynamic pharming* attack could subvert the SOP [20], and a *DNS rebinding* attack could leak information [18]. Furthermore, even though the SOP prevents certain undesirable cross-domain communication, it is unable to prevent Cross-Site Request Forgery (CSRF) attacks.

MashupOS: To offer a more fine-grained access control model for mashup applications, MashupOS proposes abstractions for expressing various trust relationships between subjects [26]. Our work is inspired by MashupOS, but we intend to resolve two problems with MashupOS. First, MashupOS requires different abstractions for expressing different types of trust relationships. It proposes `<ServiceInstance>` and `CommRequest` for *access-controlled content*, and `<Sandbox>` and `<OpenSandbox>` for *unauthorized content*. By contrast, we propose a single abstraction for expressing all trust relationships. Second, MashupOS still relies on the SOP for controlling Document Object Model (DOM) accesses, which has various vulnerabilities as mentioned. By contrast, our abstraction does not rely on the SOP for controlling DOM accesses or cross-domain data exchange, and therefore avoids SOP's vulnerabilities. While our model resolves these two problems with MashupOS, our model can express all the trust relationships described by MashupOS (Section 3.1).

1.2 OMash

We propose OMash, a new abstraction and access control model for writing secure yet flexible mashup applications. We draw an analogue to objects in object-oriented programming languages — such as Java — where an object represents a principal and objects can communicate with each other only via public methods. Our model treats each web page as a principal. By default, all the contents of a web page are private to the page; in other words, private content is accessible only within the same page. To enable inter-page communication, a page may declare a public interface, which all pages can access.

A web page's private data includes all of the content on the page, such as DOM objects and JavaScript objects and functions. In addition, it should also include authentication tokens, such as cookies, acquired by the page; in other words, only content on that page can access the authentication tokens, and the tokens will be sent out only for HTTP connections originating from that page. This mechanism reliably defeats CSRF attacks, since a malicious page cannot access or send authentication tokens held by another page.

OMash does not rely on the SOP. In fact, we advocate abolishing the SOP for controlling DOM accesses and cross-domain data exchange, since it is inflexible, prone to abuse (CSRF attacks), and vulnerable to DNS attacks. OMash, analogous to the Java object model, is simple and likely familiar to programmers. We will show that our simple model can express all the trust relationships discussed in MashupOS (Section 3.1) and can be configured to be backward compatible with the SOP (Section 3.3).

We implemented a prototype of our model using Mozilla Firefox 2.0. We did not need to modify the JavaScript engine. To implement the public interface, we only need to set a few preferences for Mozilla's Configurable Security Policies (CAPS) system. To privatize authentication tokens, we used Firefox 2's Session store API. A web page wishing to use our security model to provide a public interface can simply define a JavaScript function `getPublicInterface` that returns the page's public interface. We will show examples in Sections 3 and 4.

2. THE SAME ORIGIN POLICY

The Same Origin Policy is applied to protect three browser resources: documents, cookies, and access to remote services. To protect documents, sites from one origin cannot access documents from another origin via the Document Object Model. To protect cookies, sites can only set their own cookies and cookies are only sent to their originating site in HTTP requests. Remote services can be accessed via the `XMLHttpRequest` (XHR) object, which allows a script to issue an asynchronous HTTP request to a remote server. The SOP only permits XHR to issue requests to the origin of the containing document.

One exception to the SOP permits a script to set its domain to a suffix of the current domain, and use that newer, shorter domain for future SOP checks. For example, a script in a document originating from `foo.a.com` can perform the assignment `document.domain = "a.com"`. Thus, its origin is now `a.com`. Note that a site cannot set `document.domain` to a top-level domain (TLD) such as `.com`.

2.1 Problems

2.1.1 DOM Access

The SOP enforces a single unchangeable security policy on every site. While in many cases this is fine, since `a.com` likely does not want its DOM accessible from a page from `b.com` also running in the browser, `a.com` has no say in the matter; it cannot specify what resources other sites are allowed to access and let the browser enforce this policy.

This coarse-grained policy may result in undesired accesses. For example, documents at `http://a.com/foo` and `http://a.com/bar` are allowed to access each other, even when this is not desired. Another problem is determining the "public suffix" — previously known as the effective top-level domain (TLD) — for a given URL. While a TLD is the last dot-portion of the name (like `.com` and `.org`), many domains are effectively TLDs, such as `.co.uk`. This cannot be inferred programmatically, and instead must be determined via a list [13]. Getting this wrong could allow for a same origin violation on sites that set their `document.domain` to `.co.uk` or allow a site to get or set a cookie for the entire `.co.uk` domain.

Another problem with the SOP is that it relies on the security of the Domain Name System - a system that was not designed with security in mind. Karlof et al. demonstrated that when an attacker controls the domain name mapping it is possible to subvert the SOP using an attack technique called *dynamic pharming* [20]. Using this attack, it is possible to bypass all authentication schemes by first mapping the target domain to an attacker's web page that contains malicious JavaScript and (for example) an `<iframe>`. The attacker then re-maps the target domain to the actual server and loads the real web page in the `<iframe>` where authentication takes place as normal. This results in the attacker having full access to the user's session in the `<iframe>`, as the origin of the two pages appears to be the same from the viewpoint of the SOP. In the presence of SSL, the attack relies on the user accepting the attacker's self-signed certificate in the first stage, but as Karlof et al. discuss it is likely that the user will do so.

2.1.2 Authentication Credentials

One reason that `XMLHttpRequest` is restricted to communicating only with its originating page is because of the handling of authentication credentials in HTTP. When a request is made, cookies matching the destination domain are added to the request, as well as any other form of HTTP Authentication (e.g. Basic, Digest, NTLM) information for the domain. This is done regardless of what page caused the browser to initiate this request. Thus, the web browser can become a confused deputy [16], and this type of attack is known as Cross-Site Request Forgery (CSRF). CSRF does not rely on `XMLHttpRequest` to work, and can be performed in a variety of ways, e.g. enticing a user to click a URL, using an `<img>` tag, using a `<script>` tag, etc. However, in these cases the attacker can only cause a request that carries the user's credentials to be initiated and is unable to view the response (except possibly for the `<script>` tag approach if the targeted URL returns a script). If

XMLHttpRequest was not subject to the same-origin check, viewing the response would be possible, and would allow an attacker to both read and write data on a site for which the user possesses authentication credentials.

Another reason for restricting XMLHttpRequest is that, even in the absence of authentication credentials, it could still be used to read information from an organization's internal web site that sits behind a firewall and then communicate this information back out. This could occur if the internal web site has no authentication mechanism because it relies on the firewall to keep outsiders from accessing it, and assumes that no process inside the firewall can leak this information back out. However, this leaking of internal information is already possible using an attack technique called *DNS rebinding* [18], which makes up a portion of the *dynamic pharming* attack mentioned above. As described by Jackson et al., an attacker controlled website (e.g. attacker.com) can change its DNS mapping in order to read these unprotected internal documents.

2.2 Trust Levels

Wang et al. in MashupOS [26] enumerate all the possible trust levels available between integrators and providers in a mashup. These levels are summarized in Table 1. They identify four types of content that should be supported: (1) *isolated content* that should be isolated from other domains, (2) *access-controlled content* that should be isolated but allows for mediated access via, e.g. message passing, (3) (and (5)) *open content* that any domain can access and integrate into itself [1], and (4) *unauthorized content* that has no privileges of any domain. Isolated content is already possible via <frame> elements when each document comes from a different domain, and open content (3 and 5) is possible via the <script> tag. Note that, due to the coarse-grained nature of the SOP, access-controlled content and unauthorized content currently have no existing abstractions. This leaves mashup developers with the choice of either *no trust* using isolated content or *full trust* using open content; note that, in order to use an open script library such as Google Maps, the integrator is forced to trust the provider as the provider's library has full access to the domain of the page that includes it.

3. DESIGN

OMash can provide mashup developers with the ability to allow safe, controlled communication and interaction between web sites, and allow for the various trust models they desire.

3.1 Mediate DOM Access

We treat each web page as an object that declares public and private data and methods. A web page can only access its own content and the public content of another page. By content, we mean DOM objects (document, etc.) and JavaScript objects and functions. Thus, we no longer use the Same Origin Policy for determining whether or not an access on another page is allowed. A web page in the browser can thus be thought of as analogous to an OOP language object that has a well-defined public interface.

To achieve this, each page declares a JavaScript function named getPublicInterface. The name itself is unimportant — all that matters is that it be a valid JavaScript identifier name and that its use be standardized. Any page can access the getPublicInterface function of any other page but cannot modify it; a page can set only its own getPublicInterface function. Unless content is made accessible via the object returned by getPublicInterface, it cannot be accessed by another page.

[1] Although in the case of (3), the provider may not wish the integrator to directly access some of its private content, even though it wishes to provide some public access methods; for example, an e-mail widget.

```
var privateVar;

function getPublicInterface()
{
  function Interface()
  {
    this.getHeight = function ()
    {
      return document.body.clientHeight;
    }
    this.setVar = function (value)
    {
       privateVar = value;
    }
    this.anotherMethod = function (...)
    {
      ...
    }
    ...
  }

  return new Interface();
}
```

(a) **inner.html**, the provider, declares its public interface.

```
<iframe id="inner" src="inner.html">
...
// Broken into two steps for clarity
var win =
    document.getElementById("inner").contentWindow;
var innerInterface = win.getPublicInterface();
var innerHeight = innerInterface.getHeight();
innerInterface.setVar(10);
...
innerInterface.anotherMethod(...);
...
```

(b) **outer.html**, the integrator, calls the public interface declared in Figure 1(a)

Figure 1: Provider and integrator communicate via the public interface.

An example of its usage is as follows: let a page outer.html contain an <iframe> containing inner.html with an id of inner. The script in inner.html declares its public interface in Figure 1(a), and the script in outer.html calls this public interface in Figure 1(b). Note that the functions getHeight() and setVar() in Figure 1(a) are closures. A closure is "an expression (typically a function) that can have free variables together with an environment that binds those variables (that 'closes' the expression)." [10]. Using closures, pages can safely get and set information on other pages in a controlled manner, as closures allow for the creation of private members [11]. It might be preferable to only allow basic types like string and number to be passed around (performing checking via the typeof operator) for safety reasons.

By using the getPublicInterface function, a page's creator can specify what they want other pages to be able to access. Using this approach, the mashup developer can model a variety of trust

	P trusts *I* to access *P*'s content	*I* trusts *P* to access *I*'s resources	Content type	Existing abstraction	Run-as Principal
1	No	No	Isolated	`<frame>`	Provider
2	No	No	Access-controlled	None	Provider
3	No	Yes	Open	`<script>` (bad practice)	Integrator
4	Yes	No	Unauthorized	None	None
5	Yes	Yes	Open	`<script>`	Integrator

Table 1: The Trust Model on the Web for a provider *P* and an integrator *I* as defined in MashupOS

relationships. We will show how to model each of the trust relationships listed in Table 1 as proposed by MashupOS:

- *Isolated content:* Declare no `getPublicInterface` function (or have it return nothing). The page cannot be accessed by other pages unless it chooses to hand out, for example, callbacks for other pages to use.

- *Access-controlled content:* Provide methods for the returned interface that only allow access to a site's content based on the caller's credentials. For example, the provider could return data only if presented with a valid username and password that it verifies with an asynchronous communication with its originating server. In MashupOS, access-controlled content is provided by the `<ServiceInstance>` abstraction.

- *Open content:* In the case that the integrator trusts the provider, the provider can be placed in a separate page with an appropriate interface, and the integrator can expose whatever interface to it that it sees fit. However, in the case that the provider does not trust the integrator despite the reverse being true, the provider can demand that it be run from a page on its source domain that provides an interface to access its functionality.

- *Unauthorized content:* As defined in MashupOS (and provided by the `<Sandbox>` and `<OpenSandbox>` abstractions), unauthorized content should run without the privileges of either the integrator or the provider, and matches the trust relationship where the provider trusts the integrator but the integrator does not trust the provider (e.g. the provider has a script library for open use). As with the above case of open content, the integrator can isolate the provided content within another page on their site, or the provider can provide a page on their site. However, in the former case, this does not address the requirement that this type of content run with the privileges of neither the integrator nor the provider. Even if, for example, a script library was isolated (in terms of DOM access) inside another page on the integrator's web site, it would still be able to access the resources of the domain running it, i.e. cookies and the remote store. Since it would be able to perform actions with the authority of the domain running it, allowing the library this authority would allow it to steal the site's authentication credentials or issue its own requests with them[2]. To allow for this kind of trust relationship, we need to change the way authentication information is handled by the browser as discussed in the next section.

[2]Another concern is that the library could alter the page's interface, although this could be mitigated by enforcing a "set-once" property on `getPublicInterface` and defining it before including the untrusted script.

3.2 Mediate Authentication Credentials

To allow unauthorized content to run as neither the integrator nor the provider and to combat CSRF, we continue our model of viewing a web page as an object with public and private data. We thus propose that authentication credentials, be it HTTP authentication or cookies, be considered part of a page's private data.

However, this raises an important concern: how can pages transfer this information to another page on the same site? This is an important consideration because when, for example, after the user clicks a link, the page ceases to exist. If the link leads to another page on the site, the user would still want to remain logged in. Currently, this works because, in the case of cookies, authentication information is sent for `a.com` regardless of the request originator.

Therefore, our proposal for handling authentication information is as follows: When authentication information (HTTP or a cookie) comes in, the browser associates this information with the page that receives it, page *P*. This authentication information is passed on to other pages that are loaded via an action on *P* (for example, clicking a link), but only if the new page's domain matches that of the cookie (i.e. cookies for `a.com` are only sent to `a.com`). This is somewhat analogous to a `fork`ed process inheriting its parent's file descriptors.

While this is a natural change for HTTP authentication, it becomes trickier when dealing with cookies; the only cookies we want to treat this way are those that are used for authentication. Cookies that simply store preferences may be safely shared among pages as is currently possible.

Thus, we propose an extra attribute for cookies used for authentication named `Authentication`. Cookies that are marked with this attribute will thus be handled using the above policy. This works well for session cookies, as they are associated with a page when they are set. For persistent cookies (which normally should not be used for authentication), they can be associated with the first user-opened page. We also envision a browser setting being used to treat all or certain cookies as authentication cookies, even if they do not contain the `Authentication` attribute.

One interesting and useful consequence of this change is that it is now possible to log into two different accounts on the same website at the same time. For example, if a user has two email accounts Alice and Bob at a website, the user can log in as Alice in one window and as Bob in another window of the same browser. As a more important consequence of this change, we can lift the same-origin restriction on `XMLHttpRequest`, as malicious sites can no longer leverage CSRF to steal or modify data using XHR.[3] By lifting this

[3]If an organization relies on its firewall to protect data on its internal websites that have no access control, the same-origin restriction on `XMLHttpRequest` can prevent malicious web pages running on internal computers from stealing information on internal websites and then sending them to the external network. By contrast, our proposed access restriction on authentication information cannot prevent this attack. However, we argue that an internal website

restriction, we can accomplish safe cross-domain data exchange as the proposed JSONRequest [12, 17] does (which passes messages in the JavaScript Object Notation (JSON) [5] format and which does not send any HTTP authentication or cookies). In contrast to JSONRequest, our approach works with existing web authentication mechanisms and requires no server modification.

3.3 Backward Compatibility with the Same Origin Policy

An important consideration with our proposal is how to deal with legacy web applications that rely on the SOP. While our approach should not affect applications that use only single frames or incorporate frames from different domains that cannot interact in either the SOP or our model (by default, at least), applications that use multiple frames pose a problem. From a functionality standpoint, the solution is simple: return any functions in the target frame needed by other frames in the application via the target frame's `getPublicInterface` function.[4] From a security standpoint, however, the solution can be more complicated.

Since in our model any page can access the public interface of another page, a security-conscious application will need to ensure that the public interface provides services only to authorized callers. If the application wishes to use the Same Origin Policy for access control, it should ensure that the caller is from the same domain. The solution is straightforward: An application designer can, as part of the process of generating a user's page, embed a secret key that is shared among the pages generated for that user. Figure 2 presents a code example of this. Figure 2(a) shows the code of the provider, whose function `foo` authenticates the caller by checking the secret provided by the caller against the secret embedded in the provider. Figure 2(b) shows the code of the integrator, which passes the shared secret as a parameter to the call to the provider for authentication.

4. USAGE EXAMPLES

Section 3 showed the basic usage of `getPublicInterface`. In this section we will show how it can be used to construct more interesting applications.

Unauthorized content.

Figure 3 shows an example of Unauthorized Content. Here, we isolate an untrusted script library (in this example, Google's map service [3]).

Access-controlled content.

Figure 4 shows an example of Access-controlled Content. The integrator `a.com` authenticates itself to the resource `b.com` using a username and password, which the resource then verifies.

Service Integration.

It would also be possible to create services in which sibling resources communicate with each other, and the integrator merely connects them by passing callbacks. Figure 5 shows an example where the integrator connects resources from two sites, `b.com` and `c.com`. While this is simple enough for the case where the data being passed between services is innocuous, if the data passing between the sibling resources is sensitive, they must be able to protect themselves from a malicious integrator performing a man-in-the-middle attack. For such applications, they can draw on the body of knowledge already available to combat such problems; in this case, a client side mutual authentication library would be needed that operates in the same spirit as TLS/SSL. It should be noted, however, that the private keys for this process would still need to reside on the server, and thus the client code would in turn need to communicate data back to its origin server, likely over SSL as well.

```
// Secret for this user generated by the server
var secret = 12345;

// In the old application, would have just
// declared this here
// function foo() { ... }
function getPublicInterface()
{
  function Interface()
  {
    this.foo = function (providedSecret)
    {
      if (providedSecret != secret)
      {
        return;
      }
      // else perform requested action
    }
  }
  return new Interface();
}
```

(a) **inner.html**. The function `foo` authenticates the caller by checking the parameter `providedSecret` against the embedded global variable `secret`.

```
// Secret for this user generated by the server
var secret = 12345;
<iframe id="target" src="inner.html">
..
var targetWindow =
  document.getElementById("target").contentWindow;
var targetInterface =
    targetWindow.getPublicInterface();
..
function targetFoo()
{
  // In the old application, would have just done
  // targetWindow.foo();
  targetInterface.foo(secret);
}
```

(b) **outer.html**. It authenticates by providing the argument `secret` in the call to the provider.

Figure 2: Backward compatibility with the same origin policy

5. IMPLEMENTATION

We implemented OMash as an extension to Mozilla Firefox version 2.0. Our current implementation also requires a small change (changes to a handful of arguments to functions) to work.

5.1 Mediating DOM Access

To allow the cross-domain access to the `getPublicInterface`

should enforce its own access control to protect its valuable data rather than relying on the firewall for protection.

[4]This may get slightly more complicated if the application employs poor information hiding in terms of software engineering, but it should still be feasible.

```
function getPublicInterface()
{
  function Interface()
  {
    this.setMapCenter =
      function (latitude, longitude)
      {
        if (!map)
        {
          return;
        }
        map.setCenter(
          new GLatLng(latitude, longitude), 13);
      }
  }
  return new Interface();
}
```

(a) **map.html**, which provides a map service.

```
<iframe id="map" src="map.html">
...
var mapWindow =
    document.getElementById("map").contentWindow;
var mapInterface = mapWindow.getPublicInterface();
...
// Called on a button click, for example
function changeMapCenter()
{
  // Values in a textbox
  var latitude =
      document.getElementById("latitude").value;
  var longitude =
      document.getElementById("longitude").value;

  mapInterface.setMapCenter(latitude, longitude);
}
```

(b) **outer.html**, which uses the service provided by the code in Figure 3(a)

Figure 3: Unauthorized content example

function, we used Mozilla's Configurable Security Policies (CAPS) system [25].

Although we implemented this using an extension, what it amounts to is setting the two preferences in Table 2.

Preference Name	Value
`capability.policy.default.Window.getPublicInterface.get`	`"allAccess"`
`capability.policy.default.Window.getPublicInterface.set`	`"sameOrigin"`

Table 2: Setting preferences in Mozilla's CAPS to allow cross-domain access to the `getPublicInterface` function

Note that the other possible setting for one of these preferences is `"noAccess"`, meaning that no page can access this property, not even the originating page. There is no built-in policy that restricts access only to the same document. Thus, our current implementation does not remove the Same Origin Policy, but instead makes an

```
function getPublicInterface() {
  function Interface()
  {
    this.authenticate = function (username, password)
    {
      // Verify username and password, e.g. via an
      //  XMLHttpRequest to the server.
      ...
      // Assuming authentication succeeds, give the
      // caller a token to present for each operation.
      // This is analogous to a file descriptor
      // (although this kind of token should be
      // cryptographically random to prevent guessing).
      rememberToken(token);
      return token;
    }

    this.doSomething = function (token, ...)
    {
      // check if the presented token is valid
      if (!verifyToken(token))
      {
        return;
      }
      // else do something
      ...
    }
  }
  return new Interface();
}
```

(a) **http://b.com/resource.html**, which authenticates the caller using a username and password.

```
<iframe id="resource"
        src="http://b.com/resource.html">
...
var resourceWindow =
    document.getElementById("resource").contentWindow;
var resourceInterface =
    resourceWindow.getPublicInterface();
...
var token;
...
function authenticateToResource()
{
  token =
    resourceInterface.authenticate(username, password);
}

function doSomethingToResource()
{
  resourceInterface.doSomething(token, ...);
}
```

(b) **http://a.com/integrator.html**, which calls the code in Figure 4(a)

Figure 4: Access-controlled content example: Integrator `a.com` authenticates itself to the resource `b.com`

exception to it. An ideal implementation would only allow a document to access its own contents and the `getPublicInterface`

```
function getPublicInterface()
{
  function Interface()
  {
    this.registerC = function (referenceToC)
    {
      var cInterface =
          referenceToC.getPublicInterface();
      // Verify C really is who it claims to be by
      // calling its methods to get information,
      // e.g. certificates
      ...

    }
  }
  return new Interface();
}
```

(a) **http://b.com/resourceB.html**, provider for resource B. (resource C is similarly declared)

```
<iframe id="b" src="http://b.com/resourceB.html">
<iframe id="c" src="http://c.com/resourceC.html">
...
var bWindow =
    document.getElementById("b").contentWindow;
var bInterface =
    resourceWindow.getPublicInterface();
var cWindow =
    document.getElementById("c").contentWindow;
...

function hookUpBWithC()
{
  bInterface.registerC(cWindow);
}
```

(b) **http://a.com/integrator.html**, the integrator that connects resources B and C

Figure 5: Sibling resources example: Integrator connects `b.com` with `c.com`

function of other pages. We reserve modifying Firefox's security manager in such a way for future work.

5.2 Mediating Authentication Credentials

Cross-domain `XMLHttpRequest` can be allowed by setting the preference in Table 3.

Preference Name	Value
`capability.policy.default.XMLHttpRequest.open`	`"allAccess"`

Table 3: Setting preference in Mozilla's CAPS to allow cross-domain `XMLHttpRequest`

Setting some other preferences is also required to allow the response to be read without security violations. [22]

To make HTTP authentication and authentication cookies (cookies with our proposed `Authentication` attribute) private data, we associate these items with the browser tab that contains the page using Firefox 2's Session store API [7]. When such information is set by a site, it is placed in the data store for the tab that received the information. Since Firefox makes it possible to view and modify HTTP headers it is possible to capture such information. However, since it is possible for a web site to set a cookie in a script (i.e. by setting `document.cookie`), we had to make a small change to the source code in order to capture all cookie-related events. To allow for multiple independent sessions to exist at once, we also augment the stored authentication information with a unique identifier corresponding to the tab that received the information, as well as the location of the source window in the window hierarchy (to allow for nested `<iframe>`s). Thus, a user is able to log into the same site more than once in a different tab or window and the authentication information will be kept separately.

For each outbound request that would send authentication information under the current browser policy (i.e. with no regard for who initiates the request), we only send the information if it is found in the data store of the tab that initiated the request. This data is copied to new tabs and windows created as the result of actions on the current page, such as a link click. The data for the tab is discarded if the new page does not match the domain of the stored authentication.

6. POTENTIAL COMPLICATIONS

Since our model does not rely on the Same Origin Policy, it might cause the following complications for websites using our model.

6.1 Named Windows and Frames

Each `window` object has a property `name` that can be used as a target for links, form submissions, and for opening a new window. This name can be introduced via an explicit assignment to `window.name`, via `window.open("url", "name")`, or by setting the `name` attribute when creating a `<frame>`, or `<iframe>`. An example of its use is when a user clicks on the link `<a href="aURL" target="someName">text</a>` the URL `aURL` will load in the window named `someName`, or in a new window if no window with that name currently exists. In the case of links and form submissions, this has the potential to navigate an existing window to the given URL, disrupting the user's browsing experience. If cross-domain communication is taking place via fragment identifiers [9], this could disrupt their communication. Still, these are relatively minor problems. However, a script can gain a reference to an open window with name `windowName` via `var win = window.open("", "windowName")` (if no window with that name exists, an empty window will be created). In our testing, we found that Opera did not consider frame names when searching for a matching window name, only top-level window names, while Internet Explorer and Mozilla Firefox considered both. In current browsers, the Same Origin Policy determines whether or not access to elements inside this window (e.g. `win.document`) is allowed. In our proposal, access is permitted via the public interface of the page. Assuming the page's interface is tailored to the security level of its content, being accessible in this manner should not be a problem; simply allow access based on valid credentials. However, it is possible for the application developer to introduce a named `<frame>` or `<iframe>` that a malicious website could obtain a reference to in order to disrupt the user's browsing experience.

`<frame>`s and `<iframe>`s are often constructed with the `name` attribute set, to facilitate easy access via the `window.frames` property, e.g. `window.frames["frameName"]`. A named frame can also be accessed via `window.frameName` and `window.frames[x]` (where x is the index corresponding to the frame, e.g. 0, 1, etc.). However, dropping the `name` attribute for the aforementioned reasons and instead using `window.frames[x]` can be awkward in the presence of multiple frames. As an alternative, use the frame's `id` attribute in the same manner as in Section 3:

```
var innerWindow =
    document.getElementById("frameID").contentWindow;

var innerInterface =
    innerWindow.getPublicInterface();
```

6.2 Sibling Frames

As described by Jackson and Wang in Subspace [19], Firefox (along with Safari, Internet Explorer 7, and some configurations of Internet Explorer 6) allows the frame structure of a page to be navigated regardless of the domains involved (Opera restricts access to `frames`). Thus, a sibling frame with the name attribute set to `bar` can reference its sibling frame with name `foo` as discussed in the previous section via `parent.frames["foo"]` or `parent.foo` or `parent[x]`. If the sibling frame `foo` has a `getPublicInterface` function, `bar` can call it. Again, for the same reasons discussed in the previous section, this should not be a problem given careful design of the public interface of the page. However, in the event the developer wants to simply ensure the frames cannot reach each other, this is not currently possible, at least to our knowledge. For instance, in Firefox `frames = null;` will work, but `frames` can be restored via `delete frames;`, even if the caller is child frame[5]. Thus, even without the `name` attributes set, sibling frames are still reachable via `frames[x]`. However, it turns out that it is possible to make Firefox restrict access to the `frames` attribute from another domain by setting `"capability.policy.default.Window.frames.get"` to `"sameOrigin"` (but setting `".set"` to `"sameOrigin"` has no apparent effect).

6.3 Inheritance of Authentication Credentials

Since OMash considers authentication credentials as part of a page's private data, web authentication under OMash exhibits different semantics than under the Same Origin Policy. Under the Same Origin Policy, all pages from the same origin share authentication credentials. OMash tries to simulate these semantics by passing authentication credentials associated with a page *P* to other pages that are loaded via an action on *P* (e.g., clicking a link). We are currently working on techniques that can simulate these semantics when the user clicks the "Back" button or selects a bookmark.

7. RELATED WORK

SMash [21] proposes a model where different trust domains can create isolated components of content and code, and interact via publish and subscribe messages. It isolates components using the `iframe` tag, and URL fragment identifiers allow the frames to establish communication links. The implementation as part of OpenAjax [6] provides a JavaScript library and API to run on unmodified browsers. SMash addresses earlier problems of using fragment identifiers caused by browsers allowing complete navigation of other frames, even of different origins. It extends the fragment messaging protocol with a shared secret to ensure link integrity and prevents frame-phishing with a combination of event handlers and messages during frame unloads. However, the messaging protocol in version 1.1 of OpenAjax was vulnerable to an attack discovered by Barth et al. [8], allowing the attacker to impersonate messages between components. Since our abstraction allows for direct function calls for code communication, we avoid use of the fragment identifier messaging, which was not designed for use in this manner and lacks desirable security properties for secure messaging.

As mentioned in section 2.2, we talk in terms of MashupOS's names for the various trust levels that are possible in a mashup. MashupOS proposes its own abstractions for the missing trust levels: for *access-controlled content*, `<ServiceInstance>` and `CommRequest` and for *unauthorized content*, `<Sandbox>` and `<OpenSandbox>`. While their abstractions cover all the trust levels, they still rely on the Same Origin Policy for enforcement, as well as requiring browser writers and application developers to support and use several different abstractions. Our approach can support all the different trust levels with a single simple yet flexible abstraction, and do away with the Same Origin Policy and its problems at the same time.

We also build on the technique described by Jackson and Wang in Subspace [19] for allowing safe cross-subdomain communication (which is, from the point of view of the SOP, communication between different origins) by adopting their idea of passing JavaScript closures between different pages.

A recent project by Google name Caja [14], also allows web applications of different trust domains to directly communicate with JavaScript function calls and reference passing. With the realization that a subset of JavaScript is an object-capability language, it is possible to translate scripts to this enforced subset and to grant these scripts only the privileges they require. It is therefore possible to isolate scripts from each other and from the global execution environment (i.e. the browser window) to the degree needed. As it is merely an enforced subset of JavaScript, Caja has the advantage of requiring no changes to any web standards.

Even though our handling of authentication information could allow the same-origin restriction on `XMLHttpRequest` to be lifted, we believe that the proposed JSONRequest [12] is a good candidate for safe cross-domain data exchange since JSONRequest does not send any HTTP authentication or cookies. A prototype implementation exists as an extension for Mozilla Firefox [17]. JSON itself enjoys widespread use as a data interchange format, and is being used by, for example, Yahoo! [27] and Google [15].

Reis et al [23] propose a set of abstractions for a new browser to allow web applications to run safely. They propose that proper identification of the components that make up a web programs be used to delineate boundaries rather than the Same Origin Policy. We provide a mechanism, the shared secret key, by which different components of a web application from the same origin or different origins can authenticate each other from the `getPublicInterface` function.

8. CONCLUSION

We have presented OMash, a new security model based on object abstractions to allow web pages from different domains to interact in a safe, controlled manner. Our simple model, based on the familiar notion of public interfaces, allows mashup integrators to define various trust relationships between the integrators and providers. OMash does not rely on the Same Origin Policy for controlling DOM access or cross-domain data exchange and therefore avoids all its pitfalls and vulnerabilities. To support legacy web applications, OMash can be configured to be backward compatible with the Same Origin Policy. We have implemented a prototype of OMash as an extension to Mozilla Firefox 2.0[6] and showcased sample applications.

[5]Surprisingly, `delete parent.frames;` works even if their domains are different; this is likely a bug.

[6]We also had to modify three lines of Firefox source code.

Acknowledgments

This research is partially supported by NSF CNS award 0644450 and by an AFOSR MURI award. The authors wish to thank Helen Wang, Liang Cai, Yuan Niu, and Matthew Van Gundy for their helpful feedback.

9. REFERENCES

[1] Craigslist. http://www.craigslist.org/, 2008, (accessed August 10, 2008).

[2] Google Maps. http://maps.google.com/, 2008, (accessed August 10, 2008).

[3] Google Maps API. http://www.google.com/apis/maps/, 2008, (accessed August 10, 2008).

[4] HousingMaps. http://www.housingmaps.com/, 2008, (accessed August 10, 2008).

[5] JSON. http://www.json.org/, 2008, (accessed August 10, 2008).

[6] OpenAjax Alliance. http://www.openajax.org/, 2008, (accessed August 10, 2008).

[7] Session store API. http://developer.mozilla.org/en/docs/Session_store_API, January 2008, (accessed August 10, 2008).

[8] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Usenix Security Symposium*, 2008.

[9] J. Burke. Cross Domain Frame Communication with Fragment Identifiers. http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html, June 2006, (accessed August 10, 2008).

[10] R. Cornford. Javascript Closures. http://www.jibbering.com/faq/faq_notes/closures.html, March 2004, (accessed August 10, 2008).

[11] D. Crockford. Private Members in JavaScript. http://www.crockford.com/javascript/private.html, 2001, (accessed October 31, 2007).

[12] D. Crockford. JSONRequest. http://www.json.org/JSONRequest.html, 2006, (accessed August 10, 2008).

[13] M. Foundation. Public Suffix List: Learn more about the Public Suffix List. http://publicsuffix.org/learn/, 2008, (accessed August 10, 2008).

[14] Google. google-caja. http://code.google.com/p/google-caja/, 2008, (accessed August 10, 2008).

[15] Google. Using JSON with Google Data APIs. http://code.google.com/apis/gdata/json.html, 2008, (accessed August 10, 2008).

[16] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Reviews*, 22(4):36–38, 1988.

[17] C. Jackson. JSONRequest Extension for Firefox. http://crypto.stanford.edu/jsonrequest/, 2007, (accessed August 10, 2008).

[18] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 421–431, New York, NY, USA, 2007. ACM.

[19] C. Jackson and H. J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proceedings of the 16th International World Wide Web Conference (WWW2007)*, pages 611–620, New York, NY, USA, May 2007. ACM.

[20] C. Karlof, U. Shankar, J. Tygar, and D. Wagner. Dynamic Pharming Attacks and Locked Same-origin Policies for Web Browsers. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 58–71, New York, NY, USA, 2007. ACM.

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

[22] Z. Leatherman. Cross Domain XHR with Firefox. http://www.zachleat.com/web/2007/08/30/cross-domain-xhr-with-firefox/, August 2007, (accessed August 10, 2008).

[23] C. Reis, S. D. Gribble, and H. M. Levy. Architectural principles for safe web programs. In *Sixth Workshop on Hot Topics in Networks*, 2007.

[24] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html, August 2001, (accessed August 10, 2008).

[25] J. Ruderman. Configurable Security Policies (CAPS). http://www.mozilla.org/projects/security/components/ConfigPolicy.html, April 2006, (accessed August 10, 2008).

[26] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 1–16, New York, NY, USA, October 2007. ACM.

[27] Yahoo! Using JSON with Yahoo! Web Services. http://developer.yahoo.com/common/json.html, 2008, (accessed August 10, 2008).