

Securing User-Centric Mashup Applications

January 2008

Sigbjorn Finne
Isaac Potoczny-Jones
Eric Mertens
Galois, Inc
Beaverton, Oregon

Abstract

Having the ability to easily combine together information from a number of disparate input sources into a greater whole is a touted benefit of 'Web2.0' mashup applications. They have great promise as flexible, and user-tailored ways to both disseminate and collaborate on information on the web, but with today's web technology, face a number of security risks when being asked to also aggregate restricted information sources. This paper introduces the domain and what these risks are, along with suggested mashup application architectures that are more secure.

Introduction

Sharing and combining information from a number of disparate web resources is a common task for today's web applications. We introduce and define the key concepts in this area, focusing on how we can take advantage of the opportunities for collaboration and sharing of information that this affords without compromising or giving up on security.

Terminology

Mashups: (From Wikipedia) In technology, a **mashup** is a web application that combines data from more than one source into a single integrated tool; an example is the use of cartographic data from Google Maps to add location information to real-estate data from Craigslist, thereby creating a new and distinct web service that was not originally provided by either source.ⁱ

OAuth: An emerging authentication&authorization protocol for letting a user grant access rights to his/her Protected Resources held by a Service Provider to a third party, a web application (the Consumer.) This is done *without* the User having to hand over authentication credentials to the Consumer.ⁱⁱ

Public Resource: A piece of public data that can be offered by a Service Provider to a mashup. These are resources that are available to everyone on a particular network.

Protected Resource: A piece of "private" data that is not publicly available. This might take the form of information, text, video, or articles. It could be personal information specific to a user, or data

available to a set of users based on membership in a group.

Public Mashups: A mashup containing only public resources. No further authentication needs to be done. There are no "protected resources".

Private Mashups: A mashup containing any protected resource.

Personal Mashups: A mashup containing a protected resource that relates to a specific user, where a user should have control over sharing of that data.

Service Providers: The web site or web service controlling the data being used as input to a mashup application. Some or all of the hosted data may be Protected Resources.

Consumers: The hybrid application itself; the mashup application. The party that is aggregating the service provider resources on behalf a User or Community.

User: The end-user(s) of a mashup application. In the case of private or personal mashups, some of the resources being mashed up may be hosted by Service Providers on the User's behalf. We include a community/group of users under the User label, as well.

Web applications: common vulnerabilities

When venturing into the terrain of web applications, all issues surrounding the security of such applications

need to be carefully considered and explicitly designed for. Mashup applications are definitely no exception, quite the opposite. Confidentiality, integrity, availability and authenticity all need to be addressed.

Many kinds of attacks against both web application services and the user's browser are possible and known, cataloged by MITRE in their CVE database amongst others.ⁱⁱⁱ Some classes of vulnerabilities are more common than others, the current 'top 5' (in 2007) kinds are:^{iv}

- **Cross-site scripting (XSS).**^v User provided content is passed along to other clients' browser for malicious rendering/execution. The client-side is impacted.
- **Injection flaws.** User-supplied input is sent to an interpreter on the server, causing it to be included in commands or queries. A server-side issue.
- **Malicious File Execution.** User-supplied input is treated as valid input file content on the server.
- **Insecure Direct Object Reference.** References to sensitive information is directly included in content served out to a web browser. Attacker may harvest that info directly or use it to craft separate attacks. Both a server- and client-side issue.
- **Cross-Site Request Forgery (CSRF)**^{vi}. Attack that forces a logged-on client to send pre-authenticated requests to a vulnerable web application. Both a server- and client-side issue.

Of these, XSS, Direct Object Reference and CSRF are of particular concern in a "Web 2.0" setting where larger parts of an application are executing on the user's web browser. What are the implications for mashups with respect to these vulnerabilities?

Mashups: Web 2.0 and Cross-Domain Security

It is common for current mashup applications to take advantage of JavaScript to provide a rich and dynamic user interface; implementing what is termed an AJAX application. Combined with this, it is increasingly popular to use the JavaScript Object Notation (JSON)^{vii} format for transporting data from other web applications, including the information sources being mashed up. JSON is simply a serialized representation of JavaScript values, including arrays and dictionaries.

One immediate hurdle faced by an AJAX-based *mashup application* is one of the current foundations for web browser scripting security, the so-called Same Origin Policy^{viii}. It disallows JavaScript hosted in an HTML page from accessing web resources other than ones coming from the same domain as the server that offered up the hosting HTML page. A JavaScript-based mashup would naturally want to consult resources from a number of disparate sources.^{ix} A couple of workarounds exist to this policy, with one in particular being prevalent and compatible with JSON usage: introduce external JavaScript code (as JSON values) via the addition of extra HTML "<script>" elements to the that cause JSON values to be fetched and passed on to a client-defined callback function^x -- loading external scripts are excepted from the Same-Origin Policy. While flexible and achieving a useful goal, this use of JSON leaves applications even more susceptible to the common vulnerabilities in the previous section.^{xi}

An example of this architecture is shown in Illustration 1. In step 1, the consumer would generate JavaScript which, when executed, would instruct the user's browser to visit each "service provider." In step 2, the browser requests content from each service provider, supplying the user's session cookies. Each service provider would be enabled with a JSON (or similar) API which is the component that implements the callback. (From now on, we'll refer to this as the JSONP^{xii}).

The aggregation would occur entirely on the client side (in the browser), with JSONP requesting the page contents with the user's session cookie, no central component would need to be trusted to handle the data.

Unfortunately, we believe that configuring the Service Providers to allow such mashup access leads to a CSRF problem.

The concern is that this approach allows the Service Providers to return scripts containing private resources (via JSONP). That is passed to a callback, which circumvents the browser's cross-domain scripting

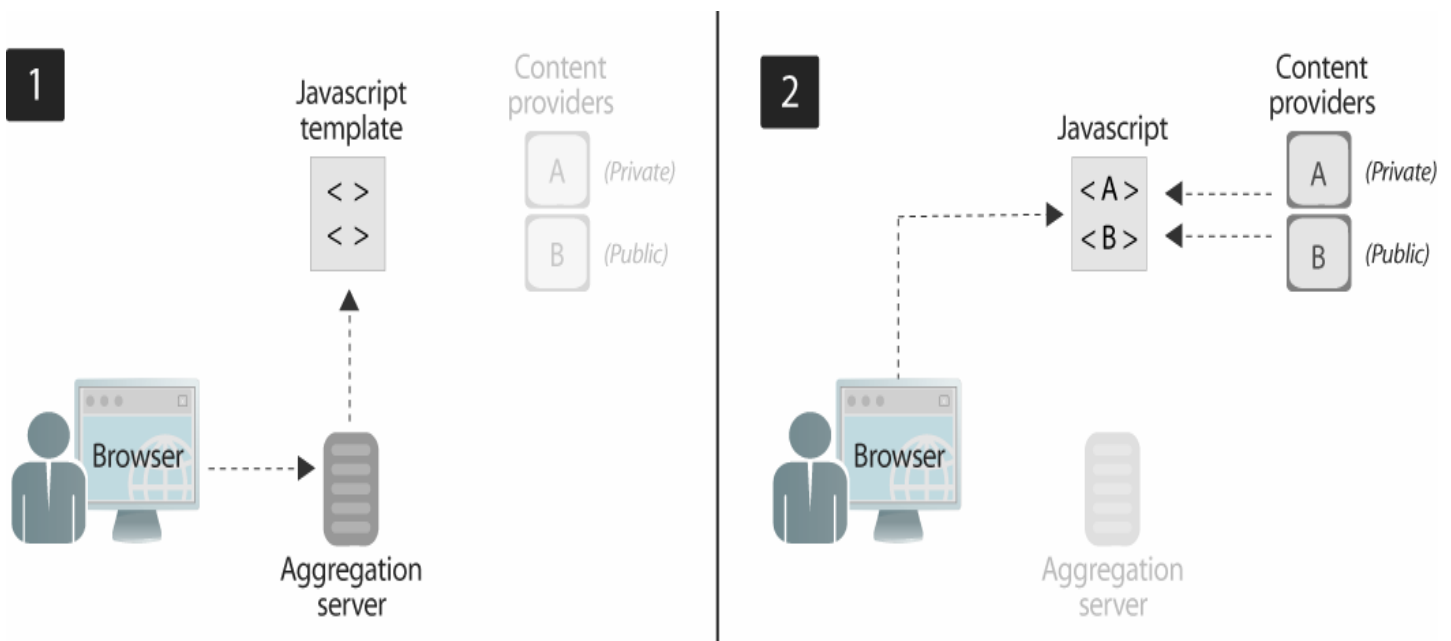
restrictions.

If the Consumer is able to instruct the user to load these scripts, then any site on the network that the user visits would be able to do the same. A malicious or poorly written page could easily instruct the user to request restricted content from the JSONP and send it to any server it might choose.

This is the case because the JavaScript callback allows a 3rd party site to proxy the user's credentials to connect to an otherwise secure service provider (thus forging the request). In this way, the 3rd party site can gain access to any data that the user has access to, just by the user loading that site, and without the user's knowledge. There is a CERT advisory about this and similar vulnerabilities^{xiii}.

There are ways to mitigate and protect against CSRF attacks by making the Service Provider much less trusting of requests coming from the User - that way weeding out the genuine mashup requests from CSRF attacks that piggybacks on a current browser session. However, the fact that a mashup application is aggregating content widens the attack surface a bit

Figure 1: Client-side aggregation with JavaScript



further, allowing cross-domain attacks that don't require issuing requests back to the server:

- **Cross-source XSS/CSRF.** If an attacker knows that it is being mashed up with other sources, it could gain access to these mashup input resources. These resources could be private to the user, so if they're mashed up along with public resources that aren't resistant to XSS attacks, the attacker may gain access to the private resource by way of the mashup application. The data may not be private per se either, but only accessible on an intranet.
- **Session snooping via CSRF.** One end-user recommended strategy for protecting yourself against CSRF attacks is not to remain logged in to web sites holding personal data (e.g., your online banking accounts, web mail) for longer than necessary, preventing those sessions from being hijacked by CSRF vulnerabilities in other web applications. Mashup applications will tend to leave sessions open to all web resources being combined, making CSRF attacks against any of those easier.

The above two aren't relevant if we consider Public Mashups of resources all on the same network, but CSRF attacks against the mashup application can still be used to determine what kind of resources a user is mashing up and exploit that extra knowledge somehow.

Still, this JavaScript + JSON approach to implementing mashups on a client is popular; it's a relatively simple, flexible and convenient way of creating public mashups. It's possible to accommodate personal mashups this way also, but requires tackling the issue of how the User authenticates and authorizes access to the Protected Resources being included in the mashup. A couple of approaches here are:

- Trust the mashup application with your login credentials (e.g., view your Gmail in the mashup that is Facebook by trusting Facebook with your Gmail password.)
- Have the user manually log in and establish a session for the mashup to use.
- Use an open-standard, distributed authentication protocol like OAuth (or vendor-specific solutions, like Yahoo's OpenAuth, Google's AuthSub etc.), which lets the user remain in control of his/her Protected Resources, but provides a way of granting access to them to a mashup application without sharing or compromising his/her login credentials.

All of these *can* be implemented with JavaScript+JSON, but many mashups either insist on being trusted with login credentials and implement the first option, or work under the umbrella of a bigger vendor's authentication solution and only support the mashing up of Protected Resources from that vendor next to public resources. A good example of this is third-party application developers registering with Google and thereby gaining mashup access to Google-hosted resources/services for a particular user through the use of the Google AuthSub protocol.

Irrespective of approach, providing a secure and trusted non-public mashup application requires careful design and even more careful JavaScript programming to achieve using *current* best practices. So, with the adoption and use of something like OAuth, it is not impossible to come up with a secure mashup application that's mainly running on a user's browser. Notice though that as this is done via browser JavaScript, all the logic and steps used to secure the application is available for any attacker to see and, potentially, circumvent.

Analysis: Although it may appear that we are trusting fewer components with our protected resource (since the consumer never touches the data), it's unclear if this is actually enforceable. The authorized consumer would be trusted not to use its privilege to look at private resources itself, but would have less liability to have to carefully handle secret data.

Reconsidering Mashups: Centralized Aggregation

So, if implementing mashup applications securely via client-side JavaScript+JSON is proving to be harder than first appearances, stressing as it does the language and browser to the point of breaking, what are the alternatives, if any? A more centralized aggregation approach to mashing up suggests itself. The mashup being a separate web application and centralized server, which performs the *main* aggregation/mashing up function.

As it is a separate web application, it is freed from running in a web browser, and can hence request resources from any network-accessible web service without having to use or rely on JSON combined with insertion of '<script>' elements (aka JSONP.xii) For private mashups, it also fits in relatively well with a protocol like OAuth, which lets the User grant access to the centralized mashup application (the Consumer) to his/her Protected Resources that are hosted by one or more Service Providers. The User remains in full control of what is shared and the credentials required

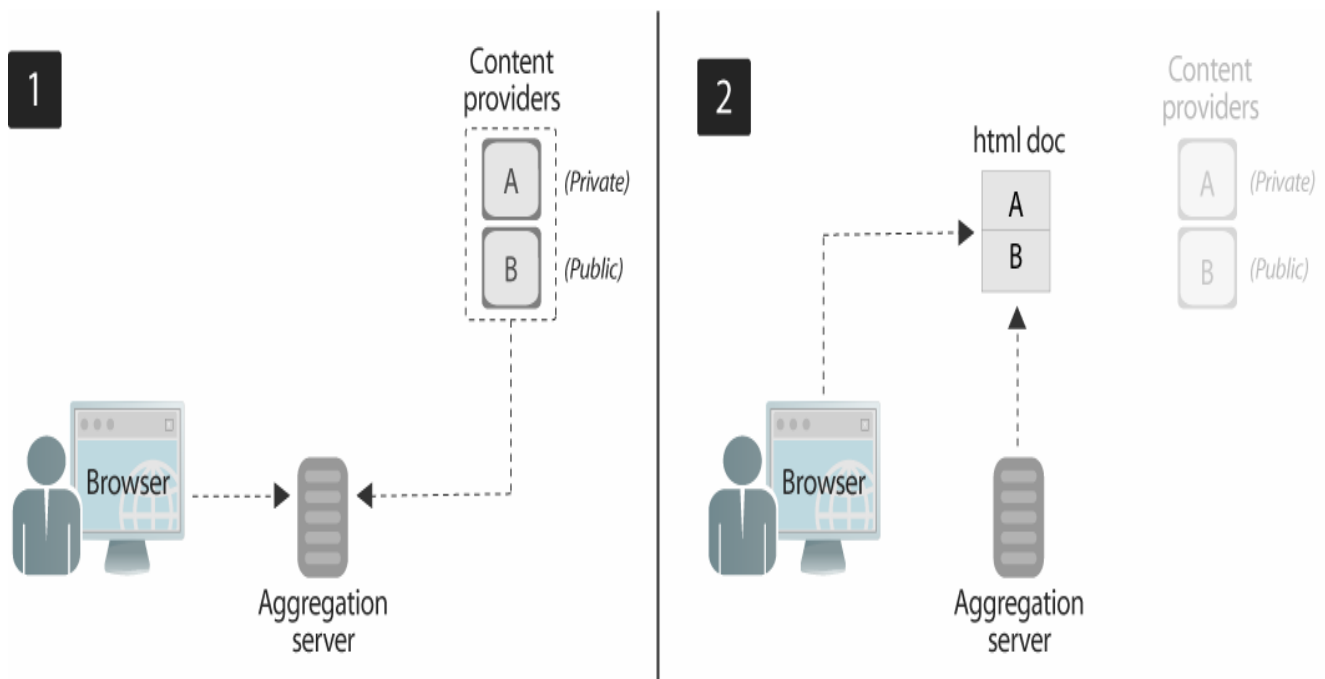
to authenticate with the Service Providers.

This approach is shown in Illustration 2. In this approach, the user authorizes the central mashup server (the Consumer) to gather protected resources on its behalf. In step 1, the user visits the mashup, and the mashup contacts the content providers to gather the resources. In step 2, the central server performs the aggregation and presents the aggregated content to the user.

When combined with the use of OAuth, the first time the user visits the mashup application, the user is redirected to the service providers to grant access to the mashup so that it can include some of his/her protected resources in the aggregated, mashed-up output. (A timeout can be set so that the grant can expire in an hour, a month, or a year, and the Consumer will no longer have access to that site.)

Analysis: A downside of this approach is that the user must now trust the Consumer (the centralized mashup component) and the Service Providers with their

Figure 2: Server-side aggregation with trusted server.



Protected Resources, since the Consumer now also has access to them. Rather than building defense-in-depth, we are simply adding another link to the chain of security; another point of failure. We believe that this is actually true of the JavaScript approach as well, although it may not be readily apparent.

Interestingly, this centralized architecture can be seen as a more general version of the other solution that AJAX applications use to get around the Same Origin Policy; having the mashup application proxy all outgoing requests back to the server.^{ix} In addition to performing the communication, the centralized component also performs the aggregation before communicating the result to the client. Notice that a centralized server does not preclude the use of AJAX-style user interfaces on the client.

Related and ongoing work

Addressing the need for more secure ways of writing mashup applications, is an ongoing concern in the web browser and applications community. A number of different efforts are underway to come up with a more secure platform for writing AJAX mashup applications, in particular:

- WAFWG's cross-site access control proposal.^{xiv}
- XMLHttpRequest object for enabling more secure, cross-domain JSON data transfers.^{xv}
- 'Sandboxing' the components of a mashup, allowing careful control over their ability to interact with other components.^{xvi}
- And others.

Common to all of them is the need to either extend the browsers (or add to their current installations) or have them become widely adopted by AJAX applications and the frameworks they're often built on top of.

Conclusions

This paper has presented a couple of different architectures for implementing Web2.0 mashup applications that aggregate content from a number of

disparate sources, with some of these sources possibly containing user sensitive content. Doing this securely, by provably fulfilling a security policy perhaps, without compromising the user's data is an absolute requirement. Possible security risks and the extended attack surface of mashup applications were introduced, highlighting in particular the added vulnerabilities faced by flexible, client-side AJAX-style mashup applications.^{xvii}

As of today (January 2008), there is no comprehensive and solid solution on offer for AJAX mashup developers, causing us to strongly suggest that another, less client-demanding, mashup application architecture based on a centralized mashup server should be given serious consideration instead. Especially if you want to make it a tractable problem to trust and possibly even prove that your mashup application's aggregated output cannot introduce covert channels that maliciously leak information about the content being mashed-up.

Complementary to the efforts of securing AJAX applications, the OAuth protocol was introduced and shown how it can effectively be used in a mashup application context to solve the issue of allowing mashup access to a user's data without handing over the credentials required to do so. That's an important piece, and will be directly applicable with either architecture.

i [http://en.wikipedia.org/wiki/Mashup_\(web_application_hybrid\)](http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid))

ii <http://oauth.net/about>

iii <http://cve.mitre.org/>

iv Copied from OWASP's Top 10 list for 2007, http://www.owasp.org/index.php/Top_10_2007

v http://en.wikipedia.org/wiki/Cross-site_scripting

-
- vi http://en.wikipedia.org/wiki/Cross-site_request_forgery
- vii <http://en.wikipedia.org/wiki/JSON>
- viii <http://www.mozilla.org/projects/security/components/same-origin.html>
- ix Assuming you don't proxy the outgoing requests in your AJAX application instead, http://ajaxpatterns.org/Cross-Domain_Proxy
- x For an explanation of this approach, see <http://www.mindsack.com/uxe/dynodes/>
- xi JavaScript Hijacking: http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf
- xii <http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/>
- xiii See "Attacker May Access Restricted Web Sites from the Client", <http://www.cert.org/advisories/CA-2000-02.html>
- xiv <http://www.w3.org/TR/access-controls>
- xv <http://json.org/>
- xvi <http://www.openajax.org/blogs/?p=36>, <http://ajaxian.com/archives/gears-and-the-mashup-problem>, and <http://www2007.org/papers/paper801.pdf>
- xvii For a second opinion and assessment of the security risks of AJAX-style mashups, see <http://www.openajax.org/whitepapers/Ajax%20Mashup%20Security.html>