Towards Building Secure Web Mashups

Maarten Decat, Philippe De Ryck,
Lieven Desmet, Frank Piessens, Wouter Joosen

IBBT-Distrinet, Katholieke Universiteit Leuven, 3001 Leuven, Belgium

Abstract. Web mashups combine components from multiple sources into a single, interactive application. This kind of setup typically requires both interaction between the components to achieve the necessary functionality, as well as component separation to achieve a secure execution. Unfortunately, the traditional web is not designed to easily fulfill both requirements, which can be seen in the restrictions imposed by traditional development techniques. This paper gives an overview of these traditional techniques and investigates new developments, specifically aimed at combining components in a secure manner. In addition, topics for further improvement are identified to ensure a wide adaptation of secure mashups.

1 Introduction

Web mashups are a relatively new development within the Web 2.0 application domain. Web mashups - simply *mashups* from now on - can be defined as "a web application that combines content (data and/or code) or services from more than one origin to create a new service". Combining multiple services typically means that some form of interaction between these services is desired.

One of the first mashups was `HousingMaps`, a site visualizing address data about houses for sale, available from `craigslist.org`, on a map, which comes from `Google Maps`. A totally different example is `iGoogle`, a user-customizable Google page. The gadgets which can be included on this page can be supplied by Google itself, or by any other developer. Another mashup example is Facebook, where users can add custom applications to their profile. A page including advertisements from Google Adsense, DoubleClick or other advertisement services, can also be seen as a mashup.

There are two major reasons for combining content from multiple origins in a mashup. First, the combined result is worth more than simply the sum of the components. `iGoogle` does this by providing the user with a single place for all information. `HousingMaps` creates its added value by visualizing address data on a map, very useful for someone looking to buy property. Enterprises can also leverage the power of mashups, by combining various data in so called *enterprise mashups*. Second, mashups can add a new dimension to content reusing, by providing a seamlessly integrated application. This both reduces effort and cost for the new application. A perfect example is the combination of web pages and content-relevant advertisement services.

Due to the nature of mashups, there are a lot of stakeholders involved. The *user* is merely interested in the added value of the mashup. This added value is created by the *composer*, who is responsible for combining multiple components into a mashup. The composer wants an elegant way to integrate the components into a mashup. The

components in their turn are created by a *developer* and made available by a *provider*. The developer wants to be able to implement the component with a minimal focus on technical interaction details, while providing limited interaction functionality to the composer. The provider is interested in easily providing mashup components, with a minimal supporting infrastructure. Finally, the mashup is offered to the users by an *integrator*. The integrator is concerned with the technical details of combining multiple components from multiple providers into the mashup created by the composer.

As an example, we can identify the stakeholders of the Facebook platform and its applications. A Facebook application is a mashup component, with the application developer being both the component's *developer* and *provider*. Users can add applications to their profile, which makes the user both the *user* and the *composer* of the mashup. Naturally, the mashup is offered by the Facebook platform, which is the *integrator* of the mashup.

Within a mashup, two main requirements can be identified: interaction and security. Component interaction has to be possible between the *principals* of a mashup, which are the integrator page and the different components. Remote interaction is needed between the components and the integrator and between the components and the provider. Typically, remote interaction will take place between different domains. An example of remote interaction can be found in the `GeoChallenge` Facebook application, which interacts with Facebook, the integrator, to retrieve a list of the user's friends. Interaction with the provider is used for saving high scores.

To ensure the security of the principals in a mashup, principals should be able to rely on the three basic security principles: confidentiality, integrity and availability. Imagine a web advertisement, embedded in an online email application, reading the mails a user is typing (confidentiality). If a component is able to modify the content of another gadget, it can both breach integrity and availability. Take for instance a malicious `iGoogle` gadget that modifies the contents of an email within the GMail gadget, which is an entirely new way of spamming (integrity). In the same way, the functionality of other components can be changed (availability). Security is also required for the principal interaction in a mashup. Two interacting principals should be able to rely on confidentiality and integrity. Additionally, they also want mutual authentication, to avoid unwanted interaction.

We can identify three general requirements for secure mashups:

R1. Separation: In order to guarantee the confidentiality, integrity and availability of the principals, they need to be separated from each other.

R2. Controlled and secure principal interaction: While respecting their separation, principals do want controlled and secure interaction. To be more precise, this requires confidentiality, integrity and mutual authentication.

R3. Cross-domain component-server interaction: Components want to be able to interact with their provider and integrator (referred to as remote interaction).

Section 2 gives an overview of traditional techniques for building mashups, as well as an analysis against the proposed requirements from above. Sections 3, 4 and 5 each discuss new proposals that focus on one of the requirements, respectively enabling interaction, enforcing separation or allowing remote interaction. Finally, section 6 discusses the most important proposals and identifies topics for further improvement. In section 7, we summarize the conclusions of this paper.

2 Traditional Techniques

Mashups, as most other web applications, will typically be run in a browser environment or a browser-based stand-alone client. Therefore, this section will study the traditional techniques to combine content and enable interaction in web pages: iframes, scripts and XMLHttpRequests. Before doing so, we will discuss the *Same Origin Policy*, the security cornerstone of all browsers. At the end of the section, we argue why an evolution in mashup techniques was required.

Same Origin Policy The *Same Origin Policy* (SOP) [16] states that scripts from one origin should not be able to access content from other origins. This prevents scripts from stealing data, cookies or login credentials from other sites. Next to the SOP implemented in browser, other technologies also have some form of SOP (e.g. Flash, Java, ...). In this paper, we will consider the SOP of the browsers, since this is our main execution environment. One important consideration is that nowadays web applications can relax the SOP to allow interaction with and between subdomains by mutually relaxing the `domain` parameter.

Iframes Using the HTML `iframe` tag, pages can embed documents from any origin. Iframes are also bound to the SOP, which means that the parent document can only access the iframe contents if it is from the same origin. The only exception is write access to the `location` property of an iframe, which is subject to the *frame navigation policy* of the browser[4]. This policy allows the parent to navigate a child-frame to a different site.

Compared to the requirements stated before, the `iframe` tag provides separation between origins, but prevents any interaction.

Script tags Using the HTML `script` tag, a page can include a script source file from any origin. The retrieved script is executed immediately within the protection domain of the including page. Since any content is executed immediately and can not be read, the `script` tag can not be used to fetch arbitrary content. Principal interaction can be achieved using ordinary function calls. Due to the inclusion in the page's protection domain, no separation is offered.

XMLHttpRequest `XMLHttpRequest` or *XHR* is a JavaScript API that is built specifically for remote interaction. Using XHR, a component is able to make a request and process the response in any way it likes. In order to maintain the separation of origins, XHR is not allowed to make cross-domain requests. This means that XHR can be used to partially meet the third requirement. The first two requirements do not apply to XHR, since it is only meant for remote interaction.

Evolving mashup techniques Table 2 shows a summary of the techniques from this section, together with the requirements from the previous section. We can conclude that iframes offer separation between components, but fail to offer principal interaction. They can be used to make remote requests, but real interaction is hard to achieve. Script tags offer principal interaction and remote interaction, but fail to offer separation between components. XHR offers remote interaction, but not in a cross-domain fashion and is not applicable for separation or principal interaction.

	Separation	Principal Interaction	Remote Interaction
Iframes	✓	✗	✓
Script tags	✗	✓	✓
XMLHttpRequest	-	-	✗

Table 1. Comparison of the different browser techniques.

We can conclude that the traditional techniques, combined with the SOP leave us with to major issues:

1. Principal separation with controlled and secure interaction is not possible.
2. Secure cross-domain component-server interaction is not possible.

The first issue can be approached in two ways. The first is by keeping the separation and adding interaction to it, to be more precise by extending iframes with principal interaction or by introducing new primitives with similar functionality. Techniques that achieve this are discussed in section 3. The second is by keeping the interaction and adding separation to it, by writing or rewriting scripts such that they are separated from the rest of the document. Techniques achieving this are discussed in section 4. Techniques that solve the second problem and provide cross-domain component-server interaction are discussed in section 5.

3 Principal interaction

This section discusses recent techniques, which achieve both separation and principal interaction, by building on the separation properties of the traditional techniques. One group of techniques extends the iframes: fragment identifier messaging, postMessage and Subspace. A second group introduces new HTML tags, which are evolutions of existing HTML tags. In this group, we discuss the `module` tag and MashupOS.

3.1 Fragment Identifier Messaging

Fragment Identifier Messaging (*FIM*)[4], also known as *Iframe Cross-Domain Communication* [13, 1], allows interaction between iframes, by using a backdoor in the SOP. FIM builds upon the fact that other domains can write to the `src` attribute of an iframe tag - also called *navigating* the iframe - although they cannot read it. Frame navigation is only restricted by the *frame navigation policy* [4], not by the SOP. Moreover, if only the fragment[1] of the new source differs from the previous one, the document inside the iframe will not be reloaded. Since the document can read its own source, a principal can interact with an iframe by encoding a message in the fragment of its source. Using nested iframes, two-way principal interaction can be achieved.

Messages exchanged using FIM are confidential, since documents from other origins are not permitted to read a frame's location. Integrity is ensured by the fact that the contents of the fragment identifier can only be overwritten completely, not partially updated. FIM does not ensure mutual authentication, since neither the origin of the sender or receiver are known. This allows for *frame phishing* attacks[4].

[1] The fragment of an URL is the part behind the # symbol.

Besides the lack of mutual authentication, FIM is also very impractical. It only offers limited message size and the receiving document has to actively monitor its fragment identifier to catch messages. Without careful timing, messages can easily get lost.

There are solutions for the problems of FIM: SMash [9] constructs a protocol stack on FIM, using shared secrets for authentication. It labels messages to detect timing errors, splits and assembles large messages automatically and takes measures against frame phishing. OMOS [17] uses a similar method. Microsoft also offers an API to use FIM in practice [13].

3.2 Subspace

Subspace [8] shares a JavaScript object across the boundaries of an iframe to achieve principal interaction. To achieve this, Subspace relies on subdomains and domain relaxation[2].

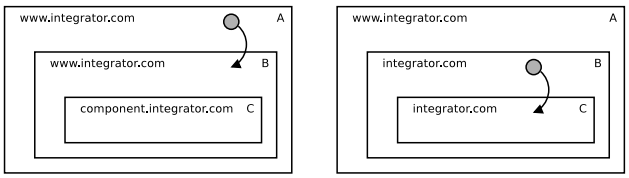

Fig. 1. Subspace: initial setup (left) and after domain relaxation (right).

An example scenario can be an integrator at `www.integrator.com` that wants to interact with a component from subdomain `component.integrator.com`. To accomplish this, it uses a setup as shown on the left in fig. 1. The integrator (A) now passes a JavaScript object to frame B, which is in the same domain. Frames B and C (the component) then use domain relaxation to enter the same domain, as shown on the right in fig. 1. Using the shared object, the integrator and the component can communicate. More complex scenarios, involving multiple components, are also supported by Subspace.

The principal interaction provided by Subspace is efficient and secure. The principals sharing references are clearly known to each other. Confidentiality and integrity can be assured by restricting access to the shared objects. A disadvantage of Subspace is the need for separate subdomains for each component, which needs to be handled by the integrator.

3.3 postMessage

postMessage is a browser API providing principal interaction [4]. It is part of the HTML5 specification [7] and has already been implemented by the major browsers. The API provides the `postMessage()` operation on a DOM `window` object, which also

[2] Documents can relax their domain to a higher level than the current domain, excluding the top level (e.g. `www.example.com` can be relaxed to `example.com`, but not `ample.com` or `com`)

applies to iframes. The receiving document can use the `onmessage` event to handle incoming messages.

postMessage can be considered an improved version of Fragment Identifier Messaging. Browsers offer native postMessage support and the `onmessage` event solves the polling issues. The origin of both sender and receiver are provided, which allows a security-aware developer to write checks to ensure mutual authentication and to prevent frame phishing attacks [4].

3.4 The `module` tag

The `module` tag is a new HTML tag trying to improve iframes on both content separation and principal interaction. Content is separated in a module, which is only accessible through a provided message passing interface. This way, the content is even separated from its origin domain. The message passing interface specifies a `send` and `receive` operation, which transfer messages in the JSON format, to prevent capability leakage by transferring JavaScript objects.

The `module` tag offers no authentication of the sender, but offers confidentiality and integrity, since the only accessible operations are `send` and `receive`. The `module` tag has not been implemented by browser vendors, but is available as an experimental addon for Firefox. Using iframes and postMessage, a similar interaction solution can be built, but internal data will be available to same origin documents. Additionally, using the new `sandbox` attribute, specified in HTML5 [7], the contents of an iframe can be forced to be from a unique origin, effectively prohibiting access to every other document.

3.5 MashupOS

MashupOS [15] takes a higher level approach. It provides four different trust levels, which all play their role in a mashup:

- *Isolated content*: separated from other domains.
- *Access-controlled content*: separated, but allows controlled interaction.
- *Open content*: accessible by all domains.
- *Unauthorized content*: readable by the containing document but not vice versa.

These four trust levels should be provided by the browser. Isolated content already exists as iframes and open content as script tags. To achieve access-controlled MashupOS defines a new `ServiceInstance` tag. A service instance is considered a unit of isolation, not only for content, but also for physical resources (memory, display, ...). A service can expose ports, which are used for controlled principal interaction. Unauthorized content, the fourth level, is provided by the proposed `Sandbox` and `OpenSandbox` tags. The `Sandbox` tag defines a private sandbox, which means that its contents are accessible for the enclosing page if they both belong to the same origin. A public sandbox, defined by the `OpenSandbox` tag, is always accessible for the enclosing page.

MashupOS offers confidentiality and integrity, but does not provide mutual authentication for principal interaction. MashupOS itself is not yet available in major browsers., but the four trust levels defined by MashupOS can be simulated in current browsers using iframes and the postMessage API.

4 Script isolation

This section discusses techniques providing separation on top of principal interaction, already available using traditional `script` tags. These techniques will limit the functionality of JavaScript, to guarantee certain security properties.

One approach will enable separation by limiting the JavaScript language to a specific subset, which complies with an *object-capability* security model. For example, if an object in this language has no reference to the Image object, it can not construct new images. By giving this object a reference to the Image object, it is given the *capability* to construct images. By using an object-capability security model, the effects of a subset on its environment can be closely monitored and regulated, thus achieving separation. Unfortunately, JavaScript provides numerous ways to obtain the global object, which a secure subset will have to take into account. Isolated principals can achieve interaction using explicitly shared objects. By doing so, the subsets offer confidentiality and integrity, but no mutual authentication. The latter can be implemented in the shared object, if desired. In this section, we will discuss three different subsets: ADsafe, Facebook JavaScript and Caja.

A second approach focuses on policy-based security at the client-side. By specifying fine-grained security policies, the separation between principals can be regulated and ensured. The security goals or the expressiveness of the policies are endless. In this section we discuss two techniques. The former, ConScript, is a fine-grained policy mechanism for access control and DOM interactions. The latter offers information flow control for JavaScript, which ensures that no sensitive data can leak to unauthorized parties. Both techniques offer confidentiality and integrity properties, but do not aim at mutual authentication.

4.1 ADsafe

Scripts written in the *ADsafe* subset, can be considered isolated and can be included in a web page without consequences. ADsafe scripts are restricted, but can still interact with their environment using the provided `ADSAFE` object. The `ADSAFE` object also provides operations to perform certain DOM operations. Note that ADsafe does not provide an automatic rewriting mechanism, but simply defines a subset and offers static code analysis to verify that scripts comply to this subset.

Concretely, ADsafe prevents dangerous JavaScript features, such as access to the `prototype` properties, use of `eval` and `with`, ... Furthermore, access to most global variables is prohibited, except for limited access to `Array`, `String`, `Math`, `Boolean` and `Number`. Since `this` can bind to the global object, its use is also prohibited.

4.2 Facebook JavaScript

The functionality of Facebook can be easily extended using Facebook applications. These applications are developed externally, but are incorporated in the main site. Security is provided by automatically rewriting an application, which can be written in full JavaScript, to Facebook JavaScript (FBJS). This rewriting process ensures that an application is restricted to its own scope, but also ensures that the application can not directly access its environment. This is done by replacing methods, such as

`document.getElementById()`, with a proxy function that controls DOM access. Similarly, XHR requests can be executed using the `AJAX` object, which proxies requests through the Facebook servers, ensuring that retrieved content is also rewritten to FBJS.

4.3 Caja

Caja is a safe subset of JavaScript, adhering to an object-capability security model. Scripts can automatically be translated to Caja modules, which are isolated from each other. During instantiation, a Caja module can be given imports, which enables the sharing of references to achieve principal interaction.

Caja deals with all unsafe JavaScript technicalities, such as `eval` or `with`. Remarkably, Caja does not prevent `this` from being used, but restricts its use. New to the language are *frozen objects*, which can not be changed. Objects in the default global environment are automatically frozen.

The Caja project also defines a second JavaScript subset, called *Cajita*. Cajita can be considered "Caja without `this`". The project developers consider the use of `this` dangerous and unnecessary, which is why they suggest writing new code directly in the Cajita subset. Caja only deals with the `this` keyword to achieve backward compatibility with currently existing code.

Caja is currently used by OpenSocial gadget integrators as Yahoo! Application Platform, Shindig, iGoogle, Code Wiki and Orkut [2, 12].

4.4 ConScript

ConScript is recently proposed by Livshits and Meyerovich to enable the specification and enforcement of fine-grained security policies for JavaScript in the browser [11]. ConScript provides an aspect-oriented rewriting approach to limit a script's capabilities to the intended policy on top of Internet Explorer 8. Among others, [11] proposes policies to achieve fine-grained access control on the communication, as well as restrictions on the DOM interactions.

To ease the policy writing, ConScript policies can be generated automatically trough static analysis of server-side code or runtime analysis of client-side code.

4.5 Information flow control for JavaScript

To achieve confidentiality and integrity of interacting mashup components, information flow control can be applied to the JavaScript components. In [10], a lattice-based approach for mashup compositions is proposed. Each origin is assigned a different level in the security lattice, and declassification allows controlled information release between components of different domains. In [6], a multi-execution approach is proposed to achieve non-interference between different levels in the security lattice, while only resulting in a limited runtime overhead on multi-core client machines.

Although implementations of information flow control for JavaScript are not yet available in the mainstream browsers, these techniques look promising to allow controlled information sharing without the need for strict component separation.

5 Component-server Interaction

This section discusses workarounds and proposals to achieve cross-domain component-server interaction. To achieve this, the barriers of the SOP need to be circumvented or loosened. By doing so, the solutions will have to ensure that no new attack vectors on legacy servers are introduced. We will discuss XMLHttpRequest proxies, script communication, cross-document interaction, browser plugins and cross-origin resource sharing, of which the latter is the latest development.

5.1 XMLHttpRequest Proxies

Due to restrictions on XHR requests, cross-domain traffic is not allowed. This can be circumvented using server-side proxies, that execute XHR requests on behalf of the client script [8]. This solution is technically not that elegant, since the browser can not include cookies or HTTP authentication headers with requests. The proxies may also want to pose restrictions on the components that can make requests, as well as on the type of requests. Another disadvantage of this approach is that in case of mashups, these proxies need to be provided by each integrator.

5.2 Script Communication

Script Communication can be used to establish cross-domain component-server communication. Using the `src` attribute of a `script` tag, a cross-domain request can be made, with information embedded in the GET parameters. The server can formulate the result as a response which is JavaScript code, that will be executed by the page embedding the script. The disadvantage of this technique is that the response of the server has full privileges, which means that the server needs to be trusted completely.

5.3 Leveraging cross-document interaction

With the introduction of postMessage, cross-document interaction has been made easy. Using postMessage, client-side proxies can be asked to make requests to a host within their origin domain. The proxies use simple XHR requests to execute the request and receive the response. The advantage here is that cookies and authentication headers can be included by the browser, contrary to server-side proxies. The restriction of server-side proxies, where a proxy per domain is needed, still applies here. For every domain, a client-side proxy needs to be available.

5.4 Using Browser Plugins

Browser plugins, such as Flash or Java, are typically not bound by the SOP of the browser. Most of them implement a customized version of the SOP [16]. Since most of these plugins offer JavaScript interaction, they can be used as a client-side proxy for making cross-domain requests.

This solution is particularly interesting if the SOP of the plugin environment allows cross-domain requests, such as the Flash plugin. Flash uses a cross-domain policy file, called `crossdomain.xml`, which is located at the target site and specifies origins that

are allowed to send cross-domain requests. This allows a specific web application to selectively accept cross-domain requests. Of course, the disadvantage here is that these plugin environments need to be available within the browser, which is not always the case. Concretely, modern devices such as the iPhone lack extensive plugin support.

5.5 Cross-Origin Resource Sharing

Cross-Origin Resource Sharing or *CORS* is a W3C Working Draft that extends the HTTP protocol to achieve controlled cross-domain client-server interaction [14]. The specification emphasizes the algorithm, which can be implemented by any API. Is is designed to not offer new attack vectors for legacy servers, to work with all data types, to allow multiple policies within one domain and to be compatible with current authentication methods. CORS implements some interesting ideas, which have been mentioned in JSONRequest, a proposal to allow unrestricted, cross-domain JSON communication [5].

CORS demands mutual agreement between client and server to retrieve cross-domain resources. A request from the client contains an origin, which can be validated by the server. The browser will only process the response, if the server indicates that the client origin is in fact trusted. This additional information can be added using the newly introduced `Origin` and `Access-Control-Allow-Origin` headers. In case of a sensitive operation, such as certain `POST` requests and all `PUT` or `DELETE` requests, the specification requires a preflight request. This preflight request is an `OPTIONS` request, to check if the server will accept the sensitive operation. This prevents the sensitive operation to be carried out, after which the browser detects that the server actually did not approve the request.

The disadvantage of CORS is that next to browser support, also servers or applications will have to be configured to include the additional header in their responses. Although CORS is still a working draft, both Firefox 3.5 and Safari 4 have extended the XHR API with this functionality. Internet Explorer 8 has implemented part of it into the new XDomainRequest API [3].

6 Discussion

In this section, we briefly summarize the three previous sections, which introduced new techniques for building mashups. In addition, we identify topics for further improvement to ensure a wide adoptation of secure mashups.

To be able to provide interaction between documents from different origins, postMessage is the way of the future. This becomes particularly clear by its inclusion in the HTML5 specification. As we have discussed earlier, postMessage improves Fragment Identifier Messaging, which is currently in use. Furthermore, postMessage also provides the needed interaction mechanism to implement other, more complex propositions, such as the `module` tag or MashupOS.

To isolate scripts, the application of the object-capability security model seems very promising. In practice however, JavaScript contains a lot of peculiarities that need to be dealt with. Due to the formal level of Caja and Cajita, together with its widespread adoption, this seems the most promising approach.

In the field of cross-domain component-server interaction, CORS is on the path of standardization. It is also an elegant solution, that provides answers to the issues of other cross-domain interaction techniques. Another advantage of CORS is the fact that it already has been implemented and is beginning to be used in real-life applications.

Principal interaction is possible using the postMessage API, but this mechanism is too low level for component developers. Ideally, an API built on top of postMessage provides an interface specification mechanism, which exposes certain operations of the component. These operations can be invoked by other components, as long as they adhere to the interface specification.

Script isolation using the object-capability security model is already provided by JavaScript subsets, such as Caja. What is lacking in this domain are ways to specify fine-grained security policies such as the ConScript access control policies and the lattice-based information flow policies, and techniques for developers and integrators to infer appropriate policies for a given component or mashup.

If an automatic rewrite or verification process for secure JavaScript subsets is available, it is executed server-side. If this technology could be provided client-side, it would relieve the integrator from this burden and at the same time allow clients to opt-in on secure, isolated mashup components.

As for the cross-domain component-server interaction, the proposed techniques provide sufficient freedom at the client-side. At the server-side however, they tend to create a management burden, especially with respect to a policy of which origins are allowed for which cross-domain interactions. Difficult-to-manage server policies lead to an "allow from all" policy, which is not desired. Especially with CORS entering in this field, techniques for automatic policy creation and management should be investigated. A possible technique would be if the component developer indicates possible entry points in the application, which can be used to generate a policy.

7 Conclusions

Mashups combine content (code and/or data) and services from multiple sources into a single application. This involves a lot of stakeholders, such as the user, the component developer, mashup composer and the integrator service. These numerous different stakeholders impose security and technical requirements on mashups. First of all, content from different providers needs to be separated, but secure and controlled interaction is desirable. The integrator also wants to combine these components easily, without having to provide a heavy communication infrastructure.

The traditional separation, interaction and communication techniques suffer from restrictions imposed by the Same Origin Policy. These limitations have driven the development of new techniques, which often start as workarounds and then evolve into standardized solutions. postMessage, which is part of the HTML5 specification has been standardized and can be used to replace or simulate the other techniques. Script isolation by means of creating a JavaScript subset is used in numerous solutions, of which Caja is the most widespread. Facebook JavaScript is also used intensively, albeit only on the Facebook platform.

Cross-domain communication, which used to be impossible, has slowly found its way into current web application development techniques, such as XMLHttpRequest,

the essence of AJAX. Cross-Origin Resource Sharing specifies a way to allow controlled cross-domain interactions and can be implemented in any API, such as XHR.

With the different solutions in mind, we still see room for further improvement, especially with respect to the usability and applicability of the proposed techniques. By building additional abstractions on top of the new technical bridges, building secure mashups will become much more easy and intuitive, ensuring the adoption and viability of web mashups.

Acknowledgment

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. Cross Domain Communication - Facebook Developer Wiki, January 2009.
2. Getting Started - Google Caja, December 2009.
3. XDomainRequest Object, December 2009.
4. Collin Jackson Adam Barth and John C. Mitchell. Securing frame communication in browsers. *Proceedings of the 17th USENIX Security Symposium*, May 2008.
5. Douglas Crockford. JSONRequest, April 2006.
6. Dominique Devriese and Frank Piessens. Non-interference through secure multi-execution. In *IEEE Symposium on Security and Privacy (Oakland), May 2010, to appear*, 2010.
7. Ian Hickson and David Hyatt. HTML 5, December 2009.
8. Collin Jackson and Helen J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. *Proceedings of the 16th international conference on World Wide Web*, 2007.
9. Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers. *Proceedings of the 17th International World Wide Web Conference (WWW)*, 2008.
10. Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 2010 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, to appear*, 2010.
11. Leo Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy (Oakland), May 2010, to appear*, 2010.
12. Sam Pullara. OpenSocial API Blog: Launched: Yahoo!'s First Implementation of OpenSocial Support, october 2008.
13. Danny Thorpe. Secure Cross-Domain Communication in the Browser, July 2007.
14. Anne van Kesteren. Cross-Origin Resource Sharing, March 2009.
15. Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. *ACM SIGOPS Operating Systems Review*, October 2007.
16. Michal Zalewski. Browser Security Handbook, December 2009.
17. Saman Zarandioon, Danfeng Yao, and Vinod Gagapathy. OMOS: A Framework for Secure Communication in Mashup Applications. *Proceedings of the 2008 Annual Computer Security Applications Conference*, 2008.