Security of Web Mashups: a Survey

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

IBBT-DistriNet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract Web mashups, a new web application development paradigm, combine content and services from multiple origins into a new service. Web mashups heavily depend on interaction between content from multiple origins and communication with different origins. Contradictory, mashup security relies on separation for protecting code and data. Traditional HTML techniques fail to address both the interaction/communication needs and the separation needs. This paper proposes concrete requirements for building secure mashups, divided in four categories: separation, interaction, communication and advanced behavior control. For the first three categories, all currently available techniques are discussed in light of the proposed requirements. For the last category, we present three relevant academic research results with high potential. We conclude the paper by highlighting the most applicable techniques for building secure mashups, because of functionality and standardization. We also discuss opportunities for future improvements and developments.

1 Introduction

The evolution within web 2.0 has led to a new application type, called a web mashup – simply *mashup* from now on. A mashup is a composed application, using elements from different sources. The most simple form of mashups are web pages incorporating advertisements, which come from an external origin. More complex examples combine content from multiple sources into a new service. The classical example case is *HousingMaps*, which collects listings of real estate from *Craigslist* and visualizes their location on *Google Maps*. There are numerous mainstream mashup examples, of which *iGoogle* and *Facebook* are widely known. Mashups have also found their way into enterprise scenarios, where they can be used to create quick views on data coming from multiple sources within and outside the enterprise. Development tools for mashup scenarios have been included in the portfolio of IT application and service providers [12,16,17,18].

A mashup can be defined as "a web application that combines content or services from more than one origin to create a new service". By combining multiple separate services into a new application, a mashup generates added value, which is one of the most important incentives behind building mashups. Mashups also

succeed in maximizing content reuse, even from services that never intended to produce reusable data. Additionally, mashups are flexible and lightweight applications, since they merely gather and combine information, thus do not need complex application logic. These three advantages have driven the growth of mashups, which has led to the need of support for strong security requirements.

The discussion of the security requirements will become more concrete if applied to an example application: a financial mashup, which provides integrated access to your financial and stock information. The mashup contains a component from your bank, an advising component from a brokerage firm and an advertising component. The bank and brokerage component need to interact, to provide relevant advice regarding your stock portfolio and interests; the brokerage and banking component provide the advertising component with keywords about your financial habits, so that you receive targeted advertisements. The bank component and brokerage component need to communicate with the servers of their firm, to retrieve the most recent information. The advertising component needs to communicate with servers from multiple advertising firms, to retrieve relevant advertisements.

A first contribution of this paper is the concrete definition of the security requirements for mashup applications, which can be used to examine existing security mechanisms. Second, we contribute a detailed overview of the current state-of-practice and adopted state-of-the-art concerning mashup security techniques. Third, we highlight a few important academic results, as well as discuss potential future improvements and developments to enhance support for the mashup security requirements.

In the remainder of this paper, we will specify the security requirements for mashups (Section 2), followed by a detailed overview of the currently available techniques (Section 3, 4 and 5). We also discuss a few promising state-of-the-art techniques, which can contribute to the future of mashup security (Section 6). We conclude the paper in Section 7 with an overview of the presented techniques and their capabilities, as well as a detailed discussion of potential future improvements or evolutions of mashup security mechanisms.

2 Problems with Mashup Security

Examining the security requirements for mashups has led to the specification of four specific categories, of which the security-specific requirements have been determined. The following overview discusses these categories and requirements, which will be used to discuss existing security mechanisms.

C1. Separation Components need to be separated from each other, to ensure the following security properties:

- **a. DOM**: ensures that the component's part of the DOM tree is separated from other components.
- **b. Script**: ensures that the component's scripts can not be influenced by other components.

   c. **Applicable in same domain**: ensures that the separation techniques can also be applied to different components belonging to the same domain.

C2. Interaction Regardless of their separation, a component requires interaction with other components and the host page. This interaction is subject to the following requirements:

   a. **Confidentiality**: ensures that sensitive information can not be stolen from interactions between components.
   b. **Integrity**: ensures that the contents of an interaction can not be modified without the knowledge of the interacting components.
   c. **Mutual authentication**: ensures that the interacting components can establish who they are interacting with.

C3. Communication Components need to be able to communicate with the mashup provider, as well as with other parties. This requires the following properties:

   a. **Cross-domain**: components should be able to communicate with other origins than the origin to which they belong.
   b. **Authentication**: a service receiving messages should be able to identify the origin of the message.

C4. Behavior Control Control over specific behavior of components is needed to selectively allow or disallow specific functionality. This category is currently state-of-the-art and too broad to grasp in a few categories.

Currently, mashup security is based on the de facto security policy of the web: the Same Origin Policy (SOP) [34]. The SOP 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. Additionally to the SOP, browsers also apply a frame navigation policy, which restricts the navigation of frames to its descendants [3].

The security provided by the traditional mechanisms for building mashups relies on the application of these browser security policies. Loading components from different origins in Iframes causes them to be separated by the SOP. Using script inclusion causes the script to be loaded in the protection domain of the including page, which is a straightforward way to achieve interaction between components. Communication with the origin of the page containing the script can be achieved using the XHR object of the JavaScript language.

These traditional mechanisms have led to two different approaches for building mashups: server-side composition and client-side composition (Figure 1). The former combines the entire mashup at the server side and serves it as a whole to the client, while the latter provides a template to the client, which retrieves all pieces separately and composes the mashup at the client side, conform to the provided template. The difference between both approaches is fading as hybrid models are being used, where separate components and pre-composed content are combined. In either model, there are no significant technical challenges. The responsibility for security always lies with the mashup integrator, taking into account the security requirements of the different components and their stakeholders.

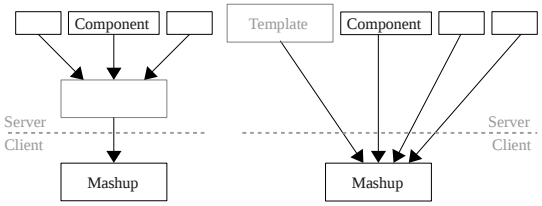

Figure 1. Server-side mashup (left) and client-side mashup (right).

Examining the traditional techniques in the light of the previously proposed security requirements yields some interesting results. Iframes offer full separation between different origins, but not within the same origin, and provide no interaction between components. Script inclusion offers no separation at all, but provides full interaction. This interaction is not authenticated, nor can confidentiality or integrity be ensured. As far as communication is concerned, XHR does not offer any cross-domain communication. These results show the pressing need for secure techniques to enable separation while still allowing secure interaction, as well as secure communication. Additionally, providing behavior control for components will only strengthen the security of mashups.

In the following sections, we provide a detailed discussion of both state-of-practice and state-of-the-art in mashup security. Section 3 focuses on specific techniques enabling separation and providing interaction. Section 4 presents techniques that enable the isolation of JavaScript modules within the same execution environment. Section 5 discusses techniques which help to achieve communication with remote parties. In Section 6, we discuss state-of-the-art academic research that supports fine-grained control over specific security-related aspects.

3 Separation and Interaction

The security requirements demand stronger separation guarantees, but also require the possibility of interaction between separated components. In this section we discuss several techniques which approach this problem on a document basis. Script-based solutions are discussed in the next section.

The solutions proposed here use three different points of view to address the needed security requirements: (i) leverage existing separation mechanisms and provide controlled interaction (Subspace, Fragment Identifier Messaging and postMessage), (ii) strengthen the existing separation mechanisms, while preserving interaction (module tag and sandbox attribute), and (iii) start from scratch, while honoring the already existing legacy by ensuring some form of backwards compatibility (MashupOS and OMash).

3.1 Subspace

Subspace [19] enables interaction across the boundaries of an iframe, using a shared JavaScript object and relying on domain relaxation. In a nutshell (Fig-

ure 2), a JavaScript object is created by frame A and shared with a nested intermediate iframe of the same domain (B). This intermediate iframe has a nested frame belonging to the component (C), which needs to obtain the JavaScript object to enable interaction. This is achieved by having both frames B and C relax their domain, so the JavaScript object can be shared. Interaction is now possible using the shared JavaScript object. More complex scenarios, involving multiple components and origins, are also supported.

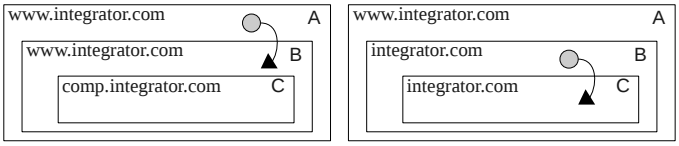

Figure 2. Subspace: initial setup (left) and after domain relaxation (right) (Source: [19]).

Subspace effectively enables interaction between frames, even with the restrictions imposed by the SOP. albeit with a few disadvantages. Apart from the fairly expensive setup phase, the burden of subdomain management for each component is another disadvantage of Subspace.

The security requirements for separation are addressed by the use of iframes. As for the security requirements regarding interaction, Subspace achieves confidentiality and integrity, as long as the shared objects are protected. Mutual authentication is inherent to the owners of the shared object, which are determined during the setup phase.

3.2 Fragment Identifier Messaging

Fragment Identifier Messaging (FIM) [3], also known as Iframe Cross-Domain Communication [10,31], builds a communication channel based on frame navigation. If the URL of a frame is set, but only the fragment[1] changes, the page is not reloaded. This allows JavaScript within the page to read this fragment, thus providing a one-way channel. Two-way interaction can be achieved using nested frames.

Even though FIM enables interaction without violating the browser's security policies, it is not a designed interaction channel. This brings a few disadvantages, such as a restricted message length, the lack of a notification system for new messages or the fact that messages can easily be overwritten.

Compared to the proposed security requirements, FIM is dependent on the use of iframes for separation. In terms of the security requirements for interaction, FIM does achieve confidentiality, since the browser's security policies prevent the frame location to be read by other origins. Integrity is also preserved,

[1] The part of a URL after the # symbol, used to navigate to an anchor within the page.

since the frame's location can only be overwritten as a whole, so no fragment can be partially modified. Mutual authentication is not available, since the sender of a message is not known, but an authentication mechanism can be implemented.

The issues with FIM can be addressed, as is shown by component framework SMash [7], the OpenAjax Hub [28], OMOS [35] and the Microsoft API for using FIM [31].

3.3 PostMessage

PostMessage is an extension of the browser API, providing a designed interaction channel between frames [14]. The specification introduces a new DOM event, `message`, which is fired if messages are received, as well as an API function that can be used to send messages to a frame, `postMessage()`. When sending a message, the destination origin has to be specified, which is validated by the browser upon message delivery [3]. For received messages, the browser provides the origin of the sender as part of the message object.

PostMessage is an improved version of FIM and addresses specific issues. Similar to FIM, the separation requirements are met by the underlying use of iframes. When compared to the security requirements for enabling interaction, postMessage does achieve confidentiality and integrity. Mutual authentication is also supported on the level of domains: the browser checks the destination when sending a message and the receiver can check the origin of a message.

PostMessage is part of the HTML5 standard, which is currently still a draft [13]. Nonetheless, postMessage is already supported by major browsers. It can also be used to replace FIM, as will be done in SMash [7] and the OpenAjax Hub [28].

3.4 Module tag

The module tag allows content separation in modules, which are only accessible through a message-passing interface for sending and receiving messages [5]. This message-passing interface is restricted to the JSON format, to prevent security issues through the leaking of JavaScript objects. Additionally, the module tag assigns a unique origin to each module, thus effectively enabling separation between multiple components from the same origin.

Compared to the security requirements for separation and interaction, the module tag effectively separates components from each other. Separation is enforced within the same domain, both for scripts as DOM elements. As for interaction between modules, confidentiality and integrity are achieved by the separation of internal state. Mutual authentication is not achieved, since there is no authentication of the sender, but can be implemented.

The module tag is not implemented by major browser vendors and is, as far as we know, not used in practice. It does however provide valuable insights and inspiration for the design of other standardized solutions, such as the sandbox attribute, discussed next.

3.5 Sandbox attribute

The sandbox attribute [15] is an extension of the iframe tag and augments the origin-based separation of iframes. The sandbox attribute imposes a set of restrictions, such as assigning a unique origin to the content, preventing scripts or browser plugins to run or preventing forms from being submitted. These restrictions, except for running plugins, can be relaxed by specifically allowing them when specifying the attribute.

Within the separation category, the sandbox attribute achieves all three security requirements. The interaction requirements are achieved by the chosen interaction technique. This can be any interaction technique available for iframes, but the standardized postMessage is a favorite, with one caveat: if a component is assigned a unique origin, the postMessage-origin is set to a globally unique identifier for outgoing messages. This may be problematic to achieve mutual authentication with sandboxed components.

The sandbox attribute is part of HTML5, which is currently a draft [13]. Major browsers are starting to support the sandbox attribute though, with Chromium/Chrome taking the lead.

3.6 MashupOS

MashupOS [33] arguments the need for additional trust levels within a mashup. Next to the "no trust" provided by iframes, known as isolated content, and "full trust" provided by script inclusion, known as open content, they propose access-controlled content, which provides separation with the possibility of message-passing across domains, and unauthorized content, which can not assume any privileges associated with a domain, such as authentication credentials or origins.

Technically, these levels of trust are achieved by introducing new HTML tags. These tags do not only provide separation and interaction, but also enable the separation of physical resources, which is out of scope here. MashupOS also provides a way for modules to expose a specific API.

Mapping MashupOS to the proposed security requirements is not easy, because there are multiple levels of trust. Using the different levels of trust, MashupOS is able to provide strong separation for both DOM elements and scripts. Separation within the same origin is dependent on the technique used (e.g. unauthorized content is not associated with a domain). As for interaction, confidentiality and integrity can be ensured using the provided API specification mechanism, but no support for mutual authentication is provided. This can however be implemented on top of the provided interaction mechanism.

MashupOS is not implemented in a major browser, but the four trust levels can be simulated using iframes and postMessage. MashupOS also serves a valuable role in the research on mashups.

3.7 OMash

A totally different approach is taken by OMash [4], where web pages are represented as objects, which have public interfaces for interaction. Such an object

encapsulates the internal state of a web page, including associated resources such as cookies or authentication credentials. By separating pages, using an object representation, OMash eliminates the need for the SOP. Resource sharing is done by passing the needed resources between objects, but only if they can be safely shared (e.g. session cookies are shared when a link within a site is followed).

OMash satisfies the separation requirements, since DOM objects and scripts belong to an object's private data. Since all objects are separated, OMash also supports separation within the same origin. Interaction is possible using the exposed interfaces, which provide confidentiality and integrity. Mutual authentication is not inherently present, but can be implemented using shared secrets.

OMash is not adopted by any major browser vendor, but is available as a prototype implementation.

4 Script Isolation

Script isolation techniques leverage the interaction possibilities present in a script environment, and try to introduce separation between different components. The general approach is restricting JavaScript to a subset, which adheres to the object-capability security model. This security model is based on the fact that separated objects have no capabilities and can only achieve capabilities on an object if they are handed a reference to that object. For example, if an object in the language has no reference to the Image object, it can not construct new images. By giving it a reference to the Image object, it obtains the capability to create images.

The three techniques presented here, i.e. ADsafe, Facebook JavaScript and Caja, follow this object-capability security model, thus achieving component separation, regardless of domain. Separation for DOM elements and built-in script objects is achieved using subset restrictions and run-time control over specific operations, such as DOM access. The isolated modules can interact using explicitly shared objects, which offer confidentiality and integrity. Mutual authentication can be implemented if desired.

4.1 ADsafe

The ADsafe subset [6] is aimed at putting guest code, such as advertisements, in a web page, without suffering security consequences. This is achieved by restricting scripts to a safe subset of JavaScript. Safe interaction with their environment, such as the DOM tree, is possible using a provided `ADSAFE` object.

ADsafe is not an active protection mechanism, but is enforced using a static code verification tool. This tool can determine whether a script adheres to the ADsafe subset or not, but will not actively rewrite code. Next to preventing access to the global object or well-known insecure language features, such as `eval` or `with`, ADsafe also prohibits the use of `this`, since it has subtle properties that can be used to obtain a reference to the global object.

In recent research on the security of JavaScript subsets, specific issues with ADsafe have been discovered [23]. These issues are minor design oversights, which do not break the fundamental model of the language. Continued formal verification is needed to prove that the ADsafe language fully adheres to the object-capability security model.

4.2 Facebook JavaScript

Facebook, the social networking site, supports an extension model based on applications, which are developed by external parties. To ensure the safe incorporation, Facebook provides Facebook JavaScript (FBJS) [11], which is a secure JavaSript subset. FBJS is an active protection mechanism, which applies a rewriting process to normal JavaScript. This rewriting process includes rewriting variable and function names to a unique namespace, as well as defining Facebook-specific DOM objects, which do not implement insecure features. Remote communication is available through an `Ajax` object, which uses a server-side proxy to retrieve cross-domain content. More importantly, this retrieved content is rewritten to FBJS, to ensure continuous protection.

The major advantage of the approach taken by Facebook is the active protection mechanism, which allows the dynamic addition of content. This is particularly useful in mashup applications. The disadvantage however is that every request needs to go through the Facebook servers, which might not be feasible for each integrator.

Recent research on the security of JavaScript subsets has also identified issues with FBJS [23]. These issues do not have an impact on the fundamental model of the language, and can be further eliminated using strong formal models.

4.3 Caja

Caja [27], a safe JavaScript subset designed by Google, takes a similar approach to FBJS. It analyzes JavaScript to detect subset violations and it rewrites the code to create isolated modules, as well as to mediate DOM access. Caja is a fairly flexible subset, since it allows the use of `this`, albeit in a limited way. Caja does more than subsetting JavaScript, it also introduces a new feature: frozen objects. Frozen objects can not be changed, which makes them ideal for information sharing between components. Objects in the default global environment are automatically frozen.

An advantage of the way Caja is introduced is that it is aimed at supporting existing scripts, with some exceptions such as `eval` or `with`. This allows a gradual transition towards the Caja subset. Underneath, a second subset is defined, named Cajita. Cajita can be considered "Caja without `this`", since `this` is considered a dangerous and unnecessary language feature. Cajita is meant to be the subset for writing new applications, while Caja is meant to be backwards compatible with current applications. Similar to FBJS, a server-side rewrite process ensures continuous protection of dynamic code.

Recent research on the security of JavaScript subsets has been able to prove that a subset based on Caja is capability safe [22]. This important result shows that a JavaScript subset can adhere to an object-capability security model, and can thus be used to achieve the proposed security requirements.

Caja is currently used by several OpenSocial gadget integrators, such as Yahoo! Application Platform, Shindig, iGoogle, Code Wiki and Orkut.

5 Communication

In this section, we discuss several techniques to achieve cross-domain communication. These techniques are mostly workarounds, to enable communication under the restrictions of the SOP. The last technique, i.e. cross-origin resource sharing, is designed to extend the SOP to allow safe, controlled cross-domain communication.

5.1 XMLHttpRequest Proxies

XHR does not allow cross-domain requests, a restriction that can be circumvented by providing a server-side proxy within the origin of the page initiating the request. The proxy receives a request for some content, retrieves it and sends it back to the requesting page. This solution is elegant in the sense that it allows the client-side implementation to use XHR, the standardized communication mechanism. The solution lacks elegance however in the fine details, such as the difficulty in handling authentication credentials of the remote site, where the information needs to be retrieved from. Another disadvantage is the fact that every component provider needs to provide a proxy. Furthermore, this proxy has to be fully trusted by the client, since it can manipulate both request and response.

When compared against the proposed security requirements, this solution does offer cross-domain communication, but offers no authentication. Even when an authentication mechanism is implemented on top of this communication channel, the proxy effectively acts as a man in the middle, which makes the authentication process untrustworthy.

This technique is currently used by Facebook JavaScript and iGoogle.

5.2 Script Communication

Scripts can be included from any origin, but their content is included in the protection domain of the page that includes it. Furthermore, the page does not get access to the contents of the received script file, which is executed immediately. This does not prevent the use of script inclusion as a communication channel: outgoing information is embedded using `GET` parameters and incoming information is encoded as JavaScript code. This code can be anything, but will most likely be JSON data.

This technique achieves cross-domain communication, but can not guarantee any authentication. Depending on the degree of separation between the components, an authentication mechanism may be implemented on top of this channel. A major issue with this technique however is the fact that the response has full privileges within the requesting page. This means that if an attacker can manipulate the response, the whole requesting page is vulnerable to attack.

This technique is used in practice, for instance in Google's mail service, Gmail.

5.3 Using Browser Plugins

By interacting with browser plugins, such as Flash or Java, cross-domain communication can be achieved. These plugins are not bound to the SOP of the browser and are free to implement their own policy. The implemented policies resemble the SOP of the browser, with some exceptions [34]. The origin to which the plugin is bound is typically the origin where it was downloaded from, not the origin of the including document. One noteworthy extension to the SOP of the browser is that Flash and Java, among others, use a cross-domain policy file (called `crossdomain.xml`) [1], which is used to selectively allow cross-domain requests. This policy file is created and served by the destination of a cross-domain request and identifies the origins where the request can come from. The plugin checks this policy file before executing a cross-domain request.

The use of browser plugins enables cross-domain communication and offers more fine-grained controls that other techniques do. Authentication can be achieved using cookies or HTTP authentication headers, but the browser plugin, which acts as a client-side proxy component, is still responsible for identifying the component behind the request. Disadvantageous to this technique is the need for browser plugin support, which can have an impact on the security of the browser platform, as shown by numerous vulnerabilities in both the Flash and Java plugin environment. Additional disadvantages are the potential lack of plugin support on mobile devices and the elevated resource consumption caused by the loaded browser plugin objects.

This technique is currently in use by Facebook JavaScript.

5.4 Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) is an extension of the HTTP protocol to support cross-domain requests [32]. CORS allows a remote server to indicate whether the given origin has access to its resources or not, a decision which is enforced by the browser. The server can formulate fine-grained decisions for particular resources, such as the HTTP methods that can be used or whether credentials (cookies, HTTP authentication) are allowed.

Technically, CORS adds request headers to provide the server additional information, such as the origin or the need for credentials, to which the server responds with response headers specifying the fine-grained decision that the browser needs to enforce. The specification preserves the protection of legacy

operations, which have no knowledge about CORS, using a deny-by-default approach.

This solution is a durable, long-term approach to enabling cross-domain communication. It even offers support for authentication, using cookies or HTTP authentication. A disadvantage with the specification is the domain-based identification of origins, which makes it hard for a remote server to distinguish requests coming from two different components from within the same origin. As experiments have shown, using CORS in conjunction with the unique origin of the `sandbox` attribute leads to a `null`-origin being associated with the request. This behavior can be attributed to the sandbox being a "privacy-sensitive" context [2].

The CORS specification is still a W3C working draft, but is already supported in major browsers. Since CORS only specifies an algorithm, browser vendors are free to implement it how they see fit. Firefox and Chrome have extended the traditional XHR communication mechanism with this additional functionality. Internet Explorer has implemented it as the new XDomainRequest API, due to previous security issues with the implementation of XHR [9].

6 Advanced Fine-Grained Control

In this section, we present three approaches which are aimed at providing fine-grained control over component behavior in a mashup. The first approach focuses on enforcing a policy on JavaScript code, either with or without specific browser-side support. A second approach mediates access to specific objects, thus enabling the enforcement of a security policy. A third approach is aimed at enabling information flow control for JavaScript.

6.1 Policy Enforcement Techniques for JavaScript

ConScript enables the specification and enforcement of fine-grained security policies for JavaScript in the browser [21]. Such policies can be used to control the script behavior, such as disallowing calls to certain functions (e.g. `eval`), or preventing the script from accessing cookies. To ease the task of writing policies, ConScript supports automatic policy generation trough static analysis of server-side code or run-time analysis of client-side code. Technically, ConScript supports the enforcement of security advice within the JavaScript engine. The advantage of this approach is its effectivity, since all indirections and ambiguities, such as different paths to the same function, are eliminated inside the JavaScript engine.

Self-protecting JavaScript [29] provides similar security features, but does not require specific support within the browser. Policy enforcement is achieved by wrapping security-sensitive JavaScript operations before normal script execution. As a consequence of not depending on browser-support, this technique faces several challenges, such as covering all access paths to a specific function or preventing wrapped operations to be restored by the malicious script. Several of these issues have been addressed in a follow-up paper [25], while others will be resolved in future research.

6.2 Mediating Access to Objects

Object views offer a fine grained control over shared objects in a JavaScript environment [26]. By creating and sharing a view of an object, instead of the full object, all calls to the object pass through the view, where a security policy can be enforced. An example application scenario is a document sharing policy, where the HTML document is a shared object. A view of this document can enforce the security policy, where a component can have read-only access to the entire DOM tree, and only gets write access to within its boundaries.

AdJail [30] offers a technique to mediate access to advertisements, which are embedded as a DOM object. Advertisements are executed separately in a sandboxed environment, where they can cause no harm. In order to preserve the user experience and to enable ad-specific services, such as compatibility with ad network targeting algorithms or billing operations, a mediation technique selectively forwards specific operations, such as visualizing content and forwarding of user interface events, between the sandbox and hosting page.

6.3 Information Flow Control for JavaScript

Applying information flow control (IFC) to mashup components on the client-side can prevent the leaking of sensitive data. A lattice-based approach to mashup composition [24] prevents unauthorized leaking between origins. Authorized sharing can be enabled by so-called *escape hatches*, which allow the declassification of specific content items. Related work is Mash-IF [20], which presents a client-side solution for enabling information flow control by means of a browser extension. The extension supports the identification of sensitive data and uses a reference monitor to prevent unauthorized disclosure within the mashup.

Additionally, secure multi-execution achieves non-interference between different levels in the security lattice, by executing a script for each security level, which results in only a limited run-time overhead on multi-core client machines [8].

7 Discussion

The overview in Figure 3 shows the compliance of the discussed solutions with the security requirements for separation and interaction. The table also indicates whether a technique is currently supported by mainstream browsers or not. From this table – and the earlier discussion – it can be concluded that the use of iframes combined with postMessage offers separation and interaction in a standardized way, without much overhead. Stronger separation can be achieved by using sandboxed iframes. For script separation within the same execution environment, Caja is most widely used and has the strongest formal background. The techniques to enable communication have not been summarized in a table, because there are too many differences between different techniques. The conclusion for this category is that the use of CORS is the recommended solution, since it is a soon-to-be-standardized approach, with very limited overhead.

	Separation			Interaction			
	DOM	**Script**	**Applicable in same domain**	**Confidentiality**	**Integrity**	**Mutual Authentication**	**Standardized/ Supported**
iframe	yes	yes	no	N/A	N/A	N/A	yes
script	no	no	no	no	no	no	yes
iframe + postmessage	yes	yes	no	yes	yes	yes	yes
sandbox + postmessage	yes	yes	yes	yes	yes	yes	yes
subspace	yes	yes	no	yes	yes	yes	yes
smash	yes	yes	no	yes	yes	yes	yes
module	yes	yes	yes	yes	yes	possible	no
mashupOS	yes	yes	yes	yes	yes	possible	no
Omash	yes	yes	yes	yes	yes	possible	no
Adsafe	yes	yes	yes	yes	yes	possible	yes
Facebook JavaScript	yes	yes	yes	yes	yes	possible	yes
Caja	yes	yes	yes	yes	yes	possible	yes

Figure 3. Overview: Separation/Isolation and Interaction. Note: mutual authentication is most of the time not available, but can be implemented (indicated by "possible").

If we revisit the running example from the beginning of the paper, we can use the following techniques to meet the security requirements: a client-side mashup composes the application by separating the components using iframes (all different domains, so no need to use sandboxes). Interaction between banking an brokerage component is enabled using postMessage, with access-control to ensure that the advertising component does not try to request private information. Both banking an brokerage component also expose an API to retrieve relevant keywords, which is publicly available, and can be used by the advertising component. The banking and brokerage component can communicate with their servers using traditional XHR. The advertising component can retrieve specific advertisements using CORS, where the remote server allows requests coming from the domain of the advertising component.

Opportunities for future work and developments for building secure mashups are available both within the currently existing techniques as in the evolution of mashups. One way currently existing techniques can be improved is by solving the remaining issues, such as the authentication problems with the use of unique-origin sandboxes [2]. Another important improvement is the support for web developers. The proposed security mechanisms, such as the postMessage API and CORS specification serve their purpose, but expose too many low level details to the developers. An abstraction on top of the postMessage API could allow developers to define a public interface in some form of interface definition language, which is then translated to the corresponding, low level message handler. Similarly, the CORS specification enables cross-domain communication, but the header injection at the server-side needs to be encapsulated by frameworks and management tools, to relieve the implementation and management burden.

A growing mashup popularity will lead to changing requirements, especially the need for fine-grained control techniques. The selective restrictions introduced by the sandbox attribute are a step in the right direction, but more fine-grained

control will be needed in the future, an evolution started by the techniques presented in Section 6. Providing secure, fine-grained policy enforcement techniques will enable developers and integrators to compose mashups, which respect specified policies. This is especially important for complex enterprise mashups, where regulations, service level agreements or contracts may need to be respected.

Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, the Research Fund K.U. Leuven and the EU-funded FP7-projects WebSand and NESSoS.

References

1. Adobe Systems Inc. Cross-domain policy file specification. `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`, January 2010.
2. A. Barth, C. Jackson, and I. Hickson. The web origin concept. `http://tools.ietf.org/html/draft-abarth-origin-07`, June 2010.
3. A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *In Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.
4. S. Crites, F. Hsu, and H. Chen. Omash: Enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108. ACM, 2008.
5. D. Crockford. The module tag. `http://www.json.org/module.html`, October 2006.
6. D. Crockford. Adsafe. `http://www.adsafe.org/`, December 2009.
7. F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: Secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th international conference on World Wide Web*, pages 535–544. ACM, 2008.
8. D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy Proceedings*, pages 109–124, 2010.
9. S. Dutta. Client-side cross-domain security. `http://msdn.microsoft.com/library/cc709423.aspx`, June 2008.
10. Facebook Developer Wiki. Cross domain communication. `http://wiki.developers.facebook.com/index.php/Cross_Domain_Communication`, January 2009.
11. Facebook Developer Wiki. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, August 2010.
12. Harmonia, Inc. Liquidapps. `http://www.liquidappsworld.com/`, 2010.
13. I. Hickson and D. Hyatt. Html 5 working draft. `http://www.w3.org/TR/html5/`, June 2010.
14. I. Hickson and D. Hyatt. Html 5 working draft - cross-document messaging. `http://www.w3.org/TR/html5/comms.html#crossDocumentMessages`, June 2010.
15. I. Hickson and D. Hyatt. Html 5 working draft - the sandbox attribute. `http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox`, June 2010.

16. IBM. IBM Mashup Center. http://www-01.ibm.com/software/info/mashup-center/, 2010.
17. Intel Corporation. Mash Maker. http://mashmaker.intel.com/web/, 2010.
18. JackBe Corporation. Presto: Powering the enterprise app store. http://www.jackbe.com/products/, 2010.
19. C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th international conference on World Wide Web*, page 620, 2007.
20. Z. Li, K. Zhang, and X. F. Wang. Mash-if: Practical information-flow control within client-side mashups. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 251–260, 2010.
21. B. Livshits and L. Meyerovich. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. Technical report, Microsoft Research, 2009.
22. S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of IEEE Security and Privacy'10*. IEEE, 2010.
23. S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *22nd IEEE Computer Security Foundations Symposium*, pages 77–91, 2009.
24. J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 15–23, 2010.
25. J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *15th Nordic Conference on Secure IT Systems*, 2010.
26. L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th international conference on World wide web*, pages 721–730, 2010.
27. M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf, January 2008.
28. OpenAjax Alliance. Openajax hub 2.0 specification. http://www.openajax.org/member/wiki/index.php?title=OpenAjax_Hub_2.0_Specification\&oldid=12174, July 2009.
29. P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, 2009.
30. M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium*, 2010.
31. D. Thorpe. Secure cross-domain communication in the browser. http://msdn.microsoft.com/en-us/library/bb735305.aspx, July 2007.
32. A. van Kesteren. Cross-origin resource sharing, 2009.
33. H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. *ACM SIGOPS Operating Systems Review*, 41(6):16, 2007.
34. M. Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Main, 2010.
35. S. Zarandioon, D. D. Yao, and V. Ganapathy. Omos: A framework for secure communication in mashup applications. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 355–364, 2008.