WebJail: Least-privilege Integration of Third-party Components in Web Mashups

Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, Wouter Joosen
IBBT-Distrinet, Katholieke Universiteit Leuven, 3001 Leuven, Belgium
Steven.VanAcker@cs.kuleuven.be

ABSTRACT

In the last decade, the Internet landscape has transformed from a mostly static world into Web 2.0, where the use of web applications and mashups has become a daily routine for many Internet users. Web mashups are web applications that combine data and functionality from several sources or components. Ideally, these components contain benign code from trusted sources. Unfortunately, the reality is very different. Web mashup components can misbehave and perform unwanted actions on behalf of the web mashup's user.

Current mashup integration techniques either impose no restrictions on the execution of a third-party component, or simply rely on the Same-Origin Policy. A least-privilege approach, in which a mashup integrator can restrict the functionality available to each component, can not be implemented using the current integration techniques, without ownership over the component's code.

We propose WebJail, a novel client-side security architecture to enable least-privilege integration of components into a web mashup, based on high-level policies that restrict the available functionality in each individual component. The policy language was synthesized from a study and categorization of sensitive operations in the upcoming HTML 5 JavaScript APIs, and full mediation is achieved via the use of deep aspects in the browser.

We have implemented a prototype of WebJail in Mozilla Firefox 4.0, and applied it successfully to mainstream platforms such as iGoogle and Facebook. In addition, micro-benchmarks registered a negligible performance penalty for page load-time (7ms), and the execution overhead in case of sensitive operations (0.1ms).

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

Keywords

Web Application Security, Web Mashups, Sandbox, Least-privilege integration

1. INTRODUCTION

The Internet has seen an explosion of dynamic websites in the last decade, not in the least because of the power of JavaScript. With JavaScript, web developers gain the ability to execute code on the client-side, providing for a richer and more interactive web experience. The popularity of JavaScript has increased even more since the advent of Web 2.0.

Web mashups are a prime example of Web 2.0. In a web mashup, data and functionality from multiple stakeholders are combined into a new flexible and lightweight client-side application. By doing so, a mashup generates added value, which is one of the most important incentives behind building mashups. Web mashups depend on collaboration and interaction between the different mashup components, but the trustworthiness of the service providers delivering components may strongly vary.

The two most wide-spread techniques to integrate third-party components into a mashup are via script inclusion and via (sandboxed) iframe integration, as will be discussed in more detail in Section 2. The script inclusion technique implies that the third-party component executes with the same rights as the integrator, whereas the latter technique restricts the execution of the third-party component according to the Same-Origin Policy. More fine-grained techniques (such as Caja [23] or FBJS [31]) require (some form of) ownership over the code to transform or restrict the component to a known safe subset before delivery to the browser. This makes these techniques less applicable to integrate third-party components directly from their service providers.

To enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components, web mashups should integrate components according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. Unfortunately, least-privilege integration of third-party mashup components can not be achieved with the current script-inclusion and frame-integration techniques. Moreover, the need for least-privilege integration becomes highly relevant, especially because of the augmented capabilities of the upcoming HTML5 JavaScript APIs [32] (such as access to local storage, geolocation, media capture and cross-domain communication).

In this paper, we propose WebJail, a novel client-side se-

curity architecture to enable the least-privilege integration of third-party components in web mashups. The security restrictions in place are configurable via a high-level composition policy under control of the mashup integrator, and allow the use of legacy mashup components, directly served by multiple service providers.

In summary, the contributions of this paper are:

1. a novel client-side security architecture, WebJail, that supports least-privilege composition of legacy third-party mashup-components

2. the design of a policy language for WebJail that is tuned to support the effective use of WebJail to limit access to the powerful upcoming HTML5 APIs

3. the implementation of WebJail and its policy language in Firefox, and evaluation and discussion of performance and usability

The rest of this paper is structured as follows. Section 2 sketches the necessary background, and Section 3 further elaborates the problem statement. In Section 4, the WebJail least-privilege integration architecture is presented and its three layers are discussed in more detail. Next, the prototype implementation in Firefox is described in Section 5, followed by an experimental evaluation in Section 6 and discussion in Section 7. Finally, Section 8 discusses related work, and Section 9 summarizes the contributions.

2. BACKGROUND

This section briefly summarizes the Same-Origin Policy. Next, Section 2.2 discusses how mashups are constructed and gives some insights in the state-of-practice on how third-party mashup components get integrated.

2.1 Same-Origin Policy

Currently, mashup security is based on the de facto security policy of the web: the Same-Origin Policy (SOP) [34]. An origin is a domain name-protocol-port triple, and 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. In addition to the SOP, browsers also apply a frame navigation policy, which restricts the navigation of frames to its descendants [1].

Among others, the Same-Origin Policy allows a per-origin separation of JavaScript execution contexts. Contexts are separated based on the origin of the window's document, possibly relaxed via the `document.domain` property to a right-hand, fully-qualified fragment of its current hostname. Within an execution context, the SOP does not impose any additional security restriction.

2.2 Integration of mashup components

The idea behind a web mashup is to integrate several web applications (components) and mash up their code, data and results. The result is a new web application that is more useful than the sum of its parts. Several publicly available web applications [25] provide APIs that allow them to be used as third-party components for web mashups.

To build a client-side mashup, an integrator selects the relevant in-house and third-party components, and provides the necessary glue code on an integrating web page to retrieve the third-party components from their respective service providers and let them interact and collaborate with each other.

As stated before, the two most-widespread techniques to integrate third-party components into a web mashup are through script inclusion or via (sandboxed) iframe-integration [4, 18].

Script inclusion. HTML script tags are used to execute JavaScript while a webpage is loading. This JavaScript code can be located on a different server than the webpage it is executing in. When executing, the browser will treat the code as if it originated from the same origin as the webpage itself, without any restrictions of the Same-Origin Policy.

The included code executes in the same JavaScript context, has access to the code of the integrating webpage and all of its datastructures. All sensitive JavaScript operations available to the integrating webpage are also available to the integrated component.

(Sandboxed) iframe integration. HTML iframe tags allow a web developer to include one document inside another. The integrated document is loaded in its own environment almost as if it were loaded in a separate browser window. The advantage of using an iframe in a mashup is that the integrated component from another origin is isolated from the integrating webpage via the Same-Origin Policy. However, the code running inside of the iframe still has access to all of the same sensitive JavaScript operations as the integrating webpage, albeit limited to its own execution context (i.e. origin). For instance, a third-party component can use local storage APIs, but only has access to the local storage of its own origin.

HTML 5 adds the "sandbox" attribute to the iframe element, allowing an integrator to disable all security-sensitive features through its "allow-scripts" keyword. Obviously, this very coarse-grained control has only a very limited applicability in a web mashup context.

3. PROBLEM STATEMENT

In this section, the attacker model is specified, as well as two typical attack vectors. Next, the increasing impact of insecure mashup composition is discussed in the context of the upcoming set of HTML5 specifications. Finally, the security assessment is concluded by identifying the requirements for secure mashup composition, namely the least-privilege integration of third-party mashup components.

3.1 Attacker model

Our attacker model is inspired by the definition of a *gadget attacker* in Barth *et al.* [1]. The term gadget in their definition should, in the context of this paper, be read as "third-party mashup component".

We describe the attacker in scope as follows:

Malicious third-party component provider The attacker is a malicious principal owning one or more machines on the network. The attacker is able to trick the integrator in embedding a third-party component under control of the attacker.

We assume a mashup that consists of multiple third-party components from several service providers, and an honest mashup consumer (i.e. end-user). A malicious third-party

component provider attempts to steal sensitive data outside its trust boundary (e.g. reading from origin-specific client-side storage), impersonate other third-party components or the integrator (e.g. requesting access to geolocation data on behalf of the integrator) or falsely operate on behalf of the end-user towards the integrator or other service providers (e.g. requesting cross-application content with XMLHttpRequest).

We have identified two possible ways in which an attacker could present himself as a malicious third-party component provider: he could offer a malicious third-party component towards mashup integrators (e.g. via a malicious advertisement, or via a malicious clone of a popular component), or he could hack into an existing third-party component of a service provider and abuse the prior existing trust relationship between the integrator and the service provider.

In this paper, we consider the mashup integrator as trusted by the mashup consumer (i.e. end-user), and an attacker has no control over the integrator, except for the attacker's ability to embed a third-party components of his choice. In addition, we assume that the attacker has no special network abilities (such as sniffing the network traffic between client and servers), browser abilities (e.g. extension under control of the attacker or client-side malware) and is constrained in the browser by the Same-Origin Policy.

3.2 Security-sensitive JavaScript operations

The impact of running arbitrary JavaScript code in an insecure mashup composition is equivalent to acquiring XSS capabilities, either in the context of the component's origin, or in the context of the integrator. For instance, a malicious third-party component provider can invoke typical security-sensitive operations such as the retrieval of cookies, navigation of the browser to another page, launch of external requests or access and updates to the Document Object Model (DOM).

However, with the emerging HTML5 specification and APIs, the impact of injecting and executing arbitrary JavaScript has massively increased. Recently, JavaScript APIs have been proposed to access geolocation information and system information (such as CPU load and ambient sensors), to capture audio and video, to store and retrieve data from a client-side datastore, to communicate between windows as well as with remote servers.

As a result, executing arbitrary JavaScript becomes much more attractive to attackers, even if the JavaScript execution is restricted to the origin of the component, or a unique origin in case of a sandbox.

3.3 Least-privilege integration

Taking into account the attack vectors present in current mashup composition, and the increasing impact of such attacks due to newly-added browser features, there is clearly a need to limit the power of third-party mashup components under control of the attacker.

Optimally, mashup components should be integrated according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. This would enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components.

Unfortunately, a least-privilege integration of third-party mashup components can not be achieved with the current script-inclusion and iframe-integration techniques. These techniques are too coarse-grained: either no restrictions (or only the Same-Origin Policy) are imposed on the execution of a third-party component, implicitly inviting abuse, or JavaScript is fully disabled, preventing any potential abuse but also fully killing desired functionality.

To make sure that attackers described in Section 3.1 do not exploit the insecure composition attack vectors and multiply their impact by using the security sensitive HTML5 APIs described in Section 3.2, the web platform needs a security architecture that supports least-privilege integration of web components. Since client-side mashups are composed in the browser, this architecture must necessarily be implemented in the browser. It should satisfy the following requirements:

R1 Full mediation. The security-sensitive operations need to be fully mediated. The attacker can not circumvent the security mechanisms in place.

R2 Remote component delivery. The security mechanism must allow the use of legacy third-party components and the direct delivery of components from the service provider to the browser environment.

R3 Secure composition policy. The secure composition policy must be configurable (and manageable) by the mashup integrator. The policy must allow fine-grained control over a single third-party component, with respect to the security-sensitive operations in the HTML5 APIs.

R4 Performance The security mechanism should only introduce a minimal performance penalty, unnoticeable to the end-user.

Existing technologies like e.g. Caja [23] and FBJS [31] require pre-processing of mashup components, while ConScript [21] does not work in a mashup context because it depends on the mashup component to load and enforce its own policy. A more thorough discussion of related work can be found in Section 8.

4. WEBJAIL ARCHITECTURE

To enable least-privilege integration of third-party mashup components, we propose WebJail, a novel client-side security architecture. WebJail allows a mashup integrator to apply the least-privilege principle on the individual components of the mashup, by letting the integrator express a secure composition policy and enforce the policy within the browser by building on top of the deep advice approach of ConScript [21].

The secure composition policy defines the set of security-sensitive operations that the component is allowed to invoke. Each particular operation can be allowed, disallowed, or restricted to a self-defined whitelist. Once loaded, the deep aspect layer will ensure that the policy is enforced on every accesspath to the security-sensitive operations, and that the policy can not be tampered with.

The WebJail architecture consists of three abstraction layers as shown in Figure 1. The upper layer, the *policy layer*, associates the secure composition policy with a mashup component, and triggers the underlying layers to enforce the

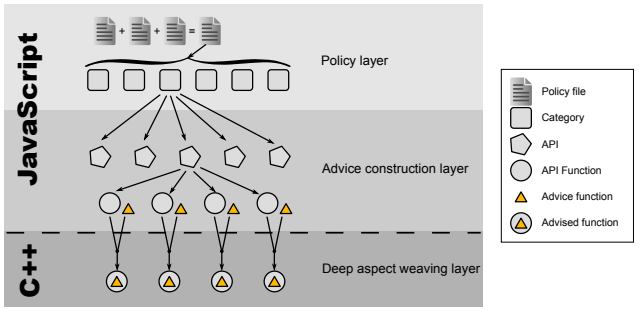


Figure 1: The WebJail architecture consists of three layers: The policy layer, the advice construction layer and the deep aspect weaving layer.

policy for the given component. The lower layer, the *deep aspect weaving layer*, enables the deep aspect support with the browser's JavaScript engine. The *advice construction layer* in between takes care of mapping the higher-level policy blocks onto the low-level security-sensitive operations via a 2-step policy refinement process.

In this section, the three layers of the WebJail will be described in more detail. Next, Section 5 will discuss a prototype implementation of this architecture in Mozilla Firefox.

4.1 Policy layer

The policy layer associates the secure composition policy with the respective mashup component. In this section, an analysis of security-sensitive operations in the HTML5 APIs is reported and discussed, as well as the secure composition policy itself.

4.1.1 Security-sensitive JavaScript operations

As part of this research, we have analyzed the emerging specifications and browser implementations, and have identified 86 security-sensitive operations, accessible via JavaScript APIs. We have synthesized the newly-added features of these specifications in Figure 2, and we will briefly summarize each of the components in the next paragraphs. Most of these features rely on (some form of) user-consent and/or have origin-restrictions in place.

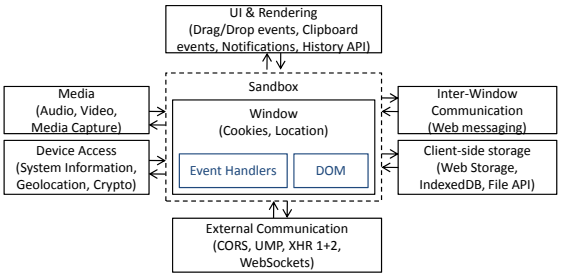


Figure 2: Synthesized model of the emerging HTML5 APIs

Central in the model is the *window* concept, containing the document. The window manifest itself as a browser window, a tab, a popup or a frame, and provides access to the location and history, event handlers, the document and its associated DOM tree. Event handlers allow to register for a specific event (e.g. being notified of mouse clicks), and access to the DOM enables a script to read or modify the document's structure on the fly. Additionally, a *sandbox* can impose coarse-grained restrictions on an iframe, as mentioned in Section 2.2.

Inter-frame communication allows sending messages between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (postMessage).

Client-side storage enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

External communication features such as CORS, UMP, XMLHttpRequest level 1 and 2, and websockets allow an application to communicate with remote websites, even in cross-origin settings.

Device access allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information and ambient sensors.

Media features enable a web application to play audio and video fragments, as well as capture audio and video via a microphone or webcam.

The UI and rendering features allow subscription to clipboard and drag-and-drop events, issuing desktop notifications and populating the history via the History API.

For a more thorough analysis of the HTML5 APIs, we would like to refer to an extensive security analysis we have carried out, commissioned by the European Network and Information Security Agency (ENISA) [7].

4.1.2 Secure composition policy

The policy layer associates the secure composition policy with a mashup component, and deploys the necessary security controls via the underlying layers. As composition granularity, we have chosen the iframe level; i.e. mashup components are each loaded in their separate iframe.

In particular, within WebJail the secure composition policy is expressed by the mashup integrator, and attached to a particular component via a newly-introduced *policy* attribute of the iframe element of the component to be loaded.

```
1 <iframe src="http://untrusted.com/compX/"
      policy="https://integrator.com/compX.policy"/>
```

We have grouped the identified security-sensitive operations in the HTML5 APIs in nine disjoint categories, based on their functionality: DOM access, Cookies, External communication, Inter-frame communication, Client-side storage, UI & Rendering, Media, Geolocation and Device access.

For a third-party component, each category can be fully disabled, fully enabled, or enabled only for a self-defined whitelist. The whitelists contain category-specific entries. For example, a whitelist for the category "DOM Access" contains the ids of the elements that might be read from or updated in the DOM. The nine security-sensitive categories are listed in Table 1, together with their underlying APIs, the amount of security-sensitive functions in each API, and their WebJail whitelist types.

The secure composition policy expresses the restrictions for each of the security-sensitive categories, and an example policy is shown below. Unspecified categories are disallowed by default, making the last line in the example policy obsolete.

```
1 { "framecomm" : "yes",
2   "extcomm" : [ "google.com", "youtube.com" ],
3   "device" : "no" }
```

Categories and APIs (# op.)	Whitelist
DOM Access	ElemReadSet, ElemWriteSet
DOM Core (17)	
Cookies	KeyReadSet, KeyWriteSet
cookies (2)	
External Communication	DestinationDomainSet
XHR, CORS, UMP (4)	
WebSockets (5)	
Server-sent events (2)	
Inter-frame Communication	DestinationDomainSet
Web Messaging (3)	
Client-side Storage	KeyReadSet, KeyWriteSet
Web Storage (5)	
IndexedDB (16)	
File API (4)	
File API: Dir. and Syst. (11)	
File API: Writer (3)	
UI and Rendering	
History API (4)	
Drag/Drop events (3)	
Media	
Media Capture API (3)	
Geolocation	
Geolocation API (2)	
Device Access	SensorReadSet
System Information API (2)	
Total number of security-sensitive operations: 86	

Table 1: Overview of the sensitive JavaScript operations from the HTML 5 APIs, divided in categories.

It is important to note that WebJails or regular frames can be used inside WebJails. In such a case, the functionality in the inner frame is determined by the policies imposed on enclosing frames, in addition to its own policy (if it has one, as is the case with a WebJail frame). Allowing sensible cascading of policies implies that "deeper" policies can only make the total policy more strict. If this were not the case, a WebJail with a less strict policy could be used to "break out" of the WebJail restrictions.

The semantics of a policy entry for a specific category can be thought of as a set. Let $\mathcal{V}$ be the set of all possible values that can be listed in a whitelist. The "allow all" policy would then be represented by the set $\mathcal{V}$ itself, a whitelist would be represented by a subset $w \subseteq \mathcal{V}$ and the "allow none" policy by the empty set ϕ. The relationship "x is at least as strict as y" can be represented as $x \subseteq y$. Using this notation, the combined policy p of 2 policies a and b is the intersection $p = a \cap b$, since $p \subseteq a$ and $p \subseteq b$.

After loading, parsing and combining all the policies applicable to the WebJail protected iframe, the policy is enforced via the underlying layers.

4.2 Advice construction layer

The task of the advice construction layer is to build advice functions based on the high-level policy received from the policy layer, and apply these advice functions on the low-level security-sensitive operations via deep aspect technology in the deep advice weaving layer.

To do so, the advice construction layer applies a 2-step refinement process. For each category of the secure composition policy, the set of relevant APIs is selected. Next for each API, the individual security-sensitive operations are processed. Consider for instance that a whitelist of type "KeyReadSet"[1] is specified for the client-side storage in the composition policy. This is first mapped to the various storage APIs in place (such as Web Storage and File API), and then advice is constructed for the security-sensitive operations in the API (e.g. for accessing the *localStorage* object).

The advice function decides, based on the policy, whether or not the associated API function will be called: if the policy for the API function is "allow all", or "allow some" and the whitelist matches, then the advice function allows the call. Otherwise, the call is blocked.

On successful completion of its job, the advice construction layer has advice functions for all the security-sensitive operations across the nine categories relevant for the specific policy. Next, the advices are applied on the original operations via the deep advice weaving layer.

4.3 Deep aspect weaving layer

The *(advice, operation)* pairs received from the advice construction layer are registered into the JavaScript engine as deep advice. The result of this weaving is that the original API function is replaced with the advice function, and that all accesspaths to the API function now go through the advice function. The advice function itself is the only place where a reference to the original API function exists, allowing it to make use of the original functionality when desired.

5. PROTOTYPE IMPLEMENTATION

To show the feasibility and test the effectiveness of WebJail, we implemented a prototype by modifying Mozilla Firefox 4.0b10pre.The modifications to the Mozilla code are localized and consist of ±800 lines of new code (±300 JavaScript, ±500 C++), spread over 3 main files. The prototype currently supports the security-sensitive categories external and inter-frame communication, client-side storage, UI and rendering (except for drag/drop events) and geolocation.

Each of the three layers of the implementation will be discussed now in more detail.

5.1 Policy layer

The processing of the secure composition policy via the *policy* attribute happens in the frame loader, which handles construction of and loading content into frames. The specified policy URL is registered as the policy URL for the frame to be loaded, and any content loaded into this frame will be subject to that WebJail policy, even if that content issues a refresh, submits a form or navigates to another URL.

When an iframe is enclosed in another iframe, and both specify a policy, the combinatory rules defined in Section 4 are applied on a per-category basis. To ease up parsing of a policy file, we have chosen to use the JavaScript Object Notation (JSON).

Once the combined policy for each category has been calculated, the list of APIs in that category is passed to the advice construction layer, along with the combined policy.

5.2 Advice construction layer

The advice construction layer builds advice functions for individual API functions. For each API, the advice construction layer knows what functions are essential to enforce the policy and builds a specific advice function that enforces it.

The advice function is a function that will be called instead of the real function. It will determine whether or not the real function will be called based on the policy and the arguments passed in the function call. Advice functions in WebJail are written in JavaScript and should expect 3 ar-

[1]Such a whitelist contains a set of keys that may be read

guments: a function object that can be used to access the original function, the object on which the function was invoked (i.e. the `this` object) and a list with the arguments passed to the function.

```
1  function makeAdvice(whitelist) {
2    var myWhitelist = whitelist;
3
4    return function(origf, obj, vp) {
5      if(myWhitelist.ROindexOf(vp[0])>=0) {
6        return origf.ROapply(obj, vp);
7      } else {
8        return false;
9      }
10   };
11 }
12
13 myAdvice = makeAdvice(['foo', 'bar']);
14 registerAdvice(myFunction, myAdvice);
15 disableAdviceRegistration();
```

Figure 3: Example advice function construction and weaving

The construction of a rather generic example advice function is shown in Figure 3. The listing shows a function `makeAdvice`, which returns an advice function as a closure containing the whitelist. Whenever the advice function is called for a function to which the first argument (`vp[0]`) is either 'foo' or 'bar', then the original function is executed. Otherwise, the advice function returns false.

Note that in the example, `ROindexOf` and `ROapply` are used. These functions were introduced to prevent prototype poisoning attacks against the WebJail infrastructure. They provide the same functionality as `indexOf` and `apply`, except that they have the `JSPROP_READONLY` and `JSPROP_PERMANENT` attributes set so they can not be modified or deleted.

Next, each *(advice, operation)* pair is passed on to the deep aspect weaving layer to achieve the deep aspect weaving.

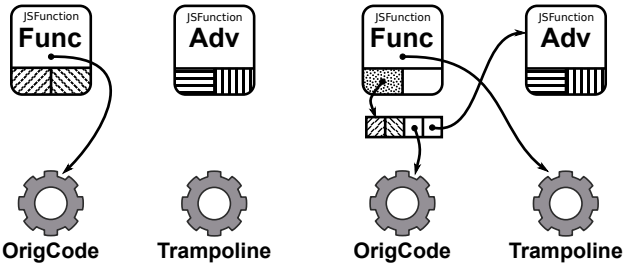
5.3 Deep aspect weaving layer

The deep aspect weaving layer makes sure that all codepaths to an advised function pass through its advice function. Although the code from WebJail is the first code to run in a WebJail iframe, we consider the scenario that there can be code or objects in place that already reference the function to be advised. It is necessary to maintain the existing references to a function, if they exist, so that advice weaving does not break code unintentionally.

The implementation of the deep aspect weaving layer is inspired by ConScript. To register deep advice, we introduce a new function called `registerAdvice`, which takes 2 arguments: the function to advise (also referred to as the 'original' function) and its advice function. Line 14 of Figure 3 illustrates the usage of the `registerAdvice` function.

In Spidermonkey, Mozilla's JavaScript engine, all JavaScript functions are represented by `JSFunction` objects. A `JSFunction` object can represent both a native function, as well as a JIT compiled JavaScript function. Because WebJail enforces policies on JavaScript APIs and all of these are implemented with native functions, our implementation only considers `JSFunction` objects which point to native code[2].

The process of registering advice for a function is schematically illustrated in Figure 4. Consider a native function `Func` and its advice function `Adv`. Before deep aspect weaving, the `JSFunction` object of `Func` contains a reference to a native C++ function `OrigCode`.



(a) Before weaving (b) After weaving

Figure 4: Schematic view of deep aspect weaving.

At weaving time, the value of the function pointer in `Func` (which points to `OrigCode`) and a reference to `Adv` are backed up inside the `Func` object. The function pointer inside `Func` is then directed towards the `Trampoline` function, which is an internal native C++ function provided by WebJail.

At function invocation time, the `Trampoline` function will be called as if it were the original function (`OrigCode`). This function can retrieve the values backed up in the weaving phase. From the backed up function pointer pointing to `OrigCode`, a new anonymous `JSFunction` object is created. This anonymous function, together with the current `this` object and the arguments to the `Trampoline` function are passed to the advice function `Adv`. Finally, the result from the advice function is returned to the calling code.

In reality, the `registerAdvice` function is slightly more complicated. In each `JSFunction` object, SpiderMonkey allocates 2 private values, known as "reserved slots", which can be used by Firefox to store opaque data. As shown in Figure 4, the reserved slots of `Func` (hatched diagonally) are backed up in the weaving phase together with the other values. During invocation time, these reserved slots are then restored into the anonymous function mentioned earlier.

Note that all code that referenced `Func` still works, although calls to this function will now pass through the advice function `Adv` first. Also note that no reference to the original code `OrigCode` is available. The only way to call this code is by making use of the advice function.

To prevent any other JavaScript code from having access to the `registerAdvice` function, it is disabled after all advice from the policy has been applied. For this purpose, WebJail provides the `disableAdviceRegistration` function, which disables the use of the `registerAdvice` function in the current JavaScript context.

6. EVALUATION

6.1 Performance

We performed micro-benchmarks on WebJail to evaluate its performance overhead with regard to page load-time and function execution. The prototype implementation is built on Mozilla Firefox 4.0b10pre, and compiled with the GNU C++ compiler v4.4.4-14ubuntu5. The benchmarks were performed on an Apple MacBook Pro 4.1, with an Intel Core 2 Duo T8300 CPU running at 2.40GHz and 4GB of memory, running Ubuntu 10.10 with Linux kernel version `2.6.35-28-generic`.

[2]Although WebJail could be implemented for non-native functions as well.

6.1.1 Page load-time overhead

To measure the page load-time overhead, we created a local webpage (`main.html`) that embeds another local page (`inner.html`) in an iframe with and without a local policy file. `inner.html` records a timestamp (`new Date().getTime()`)) when the page starts and stops loading (using the body `onload` event). WebJail was modified to record the starttime before anything else executes, so that policy retrieval, loading and application is taken into account. After the results are submitted, `main.html` reloads.

We averaged the results of 1000 page reloads. Without WebJail, the average load-time was 16.22ms ($\sigma = 3.74$ms). With WebJail, the average is 23.11ms ($\sigma = 2.76$ms).

6.1.2 Function execution overhead

Similarly, we used 2 local pages (`main.html` and `inner.html`) to measure function execution overhead. `inner.html` measures how long it takes for 10000 iterations of a piece of code to execute. We measured 2 scenarios: a typical XMLHttpRequest invocation (constructor, `open` and `send` functions) and a localStorage set and get (`setItem` and `getItem`). Besides measuring a baseline without WebJail policy, we measured each scenario when restricted by 3 different policies: "allow all", "allow none" and a whitelist with 5 values. The averages are summarized in Table 2.

	XMLHttpRequest	localStorage
Baseline	1.25 ms	0.37 ms
"Allow all"	1.25 ms (+ 0%)	0.37 ms (+ 0%)
"Allow none"	0.07 ms (- 94.4%)	0.04 ms (- 89.2 %)
Whitelist	1.33 ms (+ 6.4%)	0.47 ms (+ 27%)

Table 2: Function execution overhead

To conclude, we have registered a negligible performance penalty for our WebJail prototype: a page load-time of 7ms, and an execution overhead in case of sensitive operations about 0.1ms.

6.2 Security

As discussed in Subsection 5.3, the `registerAdvice` function disconnects an available function and makes it available only to the advice function. Because of the use of deep aspects, we can ensure that no other references to the original function are available in the JavaScript environment, even if such references already existed before `registerAdvice` was called. We have successfully verified this full mediation of the deep aspects using our prototype implementation.

Because advice functions are written in JavaScript and the advice function has the only reference to the original function, it would be tempting for an attacker to attack the WebJail infrastructure. The retrieval and application of a WebJail policy happens before any other code is executed in the JavaScript context. In addition, the `registerAdvice` function is disabled once the policy has been applied. The only remaining attack surface is the advice function during its execution. The advice functions constructed by the advice construction layer are functionally equivalent to the example advice function created in Figure 3. We know of 3 attack vectors: prototype poisoning of `Array.prototype.indexOf` and `Function.prototype.apply`, and `toString` redefinition on `vp[0]` (the first argument to the example advice function in Figure 3). By introducing the readonly copies `ROindexOf` and `ROapply` (See Subsection 5.2), we prevent an attacker from exploiting the first 2 attack vectors. The third vector, `toString` redefinition, was verified in our prototype implementation and is not an issue because `toString` is never called on the argument `vp[0]`.

6.3 Applicability

To test the applicability of the WebJail architecture, we have applied our prototype implementation to mainstream mashup platforms, including iGoogle and Facebook. As part of the setup, we have instrumented responses from these platforms to include secure composition policies, by automatically injecting a *policy* attribute in selected iframes. Next, we have applied both permissive composition policies as well as restricted composition policies and verified that security-sensitive operations for the third-party components were executed as usual in the first case, and blocked in the latter case. For instance, as part of the applicability tests, we applied WebJail to control Geolocation functionality in the Google Latitude[11] component integrated into iGoogle, as well as external communication functionality of the third-party Facebook application "Tweets To Pages"[14] integrated into our Facebook page.

7. DISCUSSION AND FUTURE WORK

In the previous sections, we have showed the feasibility of the WebJail architecture via a prototype implementation in Firefox, and evaluated the performance, security and applicability. By applying micro-benchmarks, we measured a negligible overhead, we discussed how the WebJail architecture achieves full mediation via deep aspect weaving, and we briefly illustrated the applicability of WebJail in mainstream mashup platforms.

In this section, we will discuss some points of attention in realizing least-privilege integration in web mashups and some opportunities for further improvements.

First, the granularity chosen for the secure composition policies for WebJail is primarily driven by the ease of configuration for the mashup integrator. We strongly believe that the category level of granularity increases the adoption potential by integrators and browsers, for instance compared to semantically rich and expressive security policies as is currently the case in wrapper approaches or ConScript. In fact, we chose to introduce this policy abstraction to let the integrator focus on the "what" rather than the "how". A next step could be to define policy templates per mashup component type (e.g. advertisement and geotagging components).

Nevertheless, more fine-grained policies could also be applied to achieve least-privilege integration, but one should be aware of the potential risk of creating an inverse sandbox. The goal of a least-privilege integration architecture, such as WebJail, is to limit the functionality available to a (possibly) malicious component. In case the policy language is too expressive, an attacker could use this technology to achieve the inverse. An attacker could integrate a legitimate component into his website and impose a malicious policy on it. The result is effectively a hardcoded XSS attack in the browser. For instance, the attacker could introduce an advice that leaks all sensitive information out of a legitimate component as part of its least-privilege composition policy without being stopped by the Same-Origin Policy.

One particular area where we see opportunities for more fine-grained enforcement are cross-domain interactions. Ongoing research on Cross-Site Request Forgery (CSRF) [5,

6, 28, 20] already differentiates between benign and potentially malicious cross-domain requests, and restricts the latter class as part of a browser extension. This line of research could be seen as complementary to the presented approach, and a combination of both would allow a more fine-grained enforcement for cross-domain interactions.

Second, a possible technique to escape a modified JavaScript execution context in an iframe, would be to open a new window and execute JavaScript in there. We have anticipated this attack by hardcoding policies for e.g. the `window.open` function. This is however not the best approach. The upcoming HTML 5 specs include the sandbox attribute for iframes. This specification states that a sandbox should prevent content from creating new auxiliary browsing contexts. Mozilla Firefox does not support the sandbox attribute yet. The hardcoded policy for `window.open` is a quick fix while we are working on our own full implementation of the sandbox attribute in Mozilla Firefox.

Another way to escape WebJail is to access the window object of the parent or a sibling frame and make use of the functions in that JavaScript context (e.g. `parent.navigator.geolocation.getCurrentPosition`). In such a scenario, accessing another JavaScript context falls under the Same-Origin Policy and will only be possible if both the caller and callee are in the same origin. To avoid this attack, the WebJail implementation must restrict access to sensitive operations in other execution contexts under the Same-Origin Policy.

Thirdly, the categories in the policy files of WebJail are a result of a study of the sensitive JavaScript operations in the new HTML5 APIs. Most of the HTML5 APIs are working drafts and might change in the future. The category list in WebJail is therefore an up-to-date snapshot, but might be subject to change in the future. Even after the specifications for HTML5 are officially released, the functionality in browsers might keep changing. To cope with this evolving landscape, WebJail can easily be extended to support additional categories and APIs as well.

Finally, the WebJail architecture is tailored to support least-privilege integration in mashups that are built via iframe-integration. An interesting future track is to investigate how to enable browsers to support least-privilege script-inclusion integration as well. Since in such a scenario, one can not build on the fact that a separate execution context is created, we expect this to be a challenging trajectory.

8. RELATED WORK

There is a broad set of related work that focuses on the integration of untrusted JavaScript code in web applications.

JavaScript subsets.

A common technique to prevent undesired behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed by the integrator.

ADSafe[3] and FBJS[31] requires third-party components to be written in a JavaScript subset that is known to be safe. The ADSafe subset removes several unsafe features from JavaScript (e.g. global variables, eval, ...) and provides safe alternatives through the `ADSAFE` object. Caja[23], Jacaranda[15] and Live Labs' Websandbox[22] take a different approach. Instead of heavily restricting the developer's language, they transform the JavaScript code into a safe version. The transformation process is based on both static analysis and rewriting to integrate runtime checks.

These techniques effectively support client-side least-privilege integration of mashup components. The main disadvantage is the tight coupling of the security features with the third-party component code. This requires control over the code, either at development or deployment time, which conflicts with legacy components and remote component delivery (R2), and reduces the applicability to mashup scenarios where the integrator delivers the components to the browser.

JavaScript instrumentation and access mediation.

Instead of restricting a third-party component to a JavaScript subset, access to specific security-sensitive operations can be mediated. Mediation can consist of blocking the call, or letting a policy decide whether or not to allow it.

BrowserShield[26] is a server-side rewriting technique, that rewrites certain JavaScript functions to use safe equivalents. These safe equivalents are implemented in the "bshield" object that is introduced through the BrowserShield JavaScript libraries that are injected into each page. BrowserShield makes use of a proxy to inject its code into a webpage.

Self-protecting JavaScript[24, 19] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring any browser modifications. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. The advice is *non-deep* advice, meaning that by protecting one operation, different access paths to the same operation are not automatically protected. The main challenge of this approach is to ensure full mediation (R1) without breaking the component's legitimate functionality (e.g. via removal of prototypes), since both policy and third-party component code live in the same JavaScript context.

Browser-Enforced Embedded Policies (BEEP)[16] injects a policy script at the server-side. The browser will call this policy script before loading another script, giving the policy the opportunity to vet the script about to be loaded. The loading process will only continue after the approval of the policy. This approach offers control over which scripts are loaded, but is too coarse grained to assign privileges to specific components.

ConScript[21] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript, except that ConScript uses *deep* advice, thus protects all access paths to a function. The price for using deep advice is the need for client-side support in the JavaScript engine. A limitation of ConScript is that policies are not composition policies: the policies are provided by and applied to the same webpage, which conflicts with remote component delivery (R2) and secure composition policy configurable by the integrator (R3).

In contrast to the techniques described above, WebJail offers the integrator the possibility to define a policy that restricts the behavior of a third-party component in an isolated way. Additionally, all of the techniques above use JavaScript as a policy language. This amount of freedom complicates the writing of secure policies: protection against all the emerging HTML5 APIs is fully up to policy writer and can be error-prone, a problem that the WebJail policy language is not susceptible to.

Web application code and data analysis.

A common protection technique against XSS vulnerabilities or attacks is server-side code or data analysis. Even though these techniques can only be used to check if a component matches certain security requirements and do not enforce a policy, we still discuss them here, since they are a server-side way to ensure that a component meets certain least-privilege integration requirements *out-of-the-box*.

Gatekeeper[12] is a mostly static [sic] enforcement mechanism designed to defend against possibly malicious JavaScript widgets on a hosting page. Gatekeeper analyzes the complete JavaScript code together with the hosting page. In addition, Gatekeeper uses runtime enforcement to disable dynamic JavaScript features.

XSS-Guard[2] aims to detect and remove scripts that are not intended to be present in a web application's output, thus effectively mitigating XSS attacks. XSS-Guard dynamically learns what set of scripts is used for an HTTP request. Using this knowledge, subsequent requests can be protected.

Recently, Mozilla proposed the Content Security Policy (CSP) [29], which allows the integrator to insert a security policy via response headers or meta tags. Unfortunately, CSP only supports restrictions on a subset of the security-sensitive operations discussed in this paper, namely operations potentially leading to content injection (e.g. script inclusion and XHR).

Information flow control.

Information flow control techniques can be used to detect unauthorized information sharing or leaking between origins or external parties. This is extremely useful for applications that are allowed to use sensitive data, such as a location, but are not allowed to share that data.

Both Magazinius et al.[18] and Li et al.[17] have proposed an information flow control technique that prevents unauthorized sharing of data. Additionally, both techniques support authorized sharing by means of declassification, where a certain piece of data is no longer considered sensitive.

Secure multi-execution[9] detects information leakage by simultaneously running the code for each security level. This approach is a robust way to detect information leakage, but does not support declassification.

Information flow control techniques themselves are not suited for enforcing least-privilege integration. Likewise, WebJail is not suited to enforce information flow control, since it would be difficult to cover all possible leaks. Both techniques are complementary and can be used together to ensure least-privilege integration without unauthorized information leaking.

Isolating content using specialized HTML.

Another approach to least-privilege integration is the isolation of untrusted content. By explicitly separating the untrusted code, it becomes easier to restrict its behavior, for example by preventing script execution.

The "untrusted" attribute[10] on a div element aims to allow the browser to make the difference between trusted and untrusted code. The idea is to enclose any untrusted content with such a div construct. This technique fails to defend against injecting closing tags, which would trivially circumvent the countermeasure.

The new "sandbox" attribute of the iframe element in HTML 5[13] provides a safer alternative, but is very coarse-grained. It only supports limited restrictions, and as far as JavaScript APIs are concerned, it only supports to completely enable or disable JavaScript.

ADJail[30] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to. Changes to the shadow page are replicated to the hosting page if those changes conform to the specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. ADJail limits DOM access and UI interaction with the component, but does not restrict the use of all other sensitive operations like WebJail can.

User-provided policies.

Mozilla offers *Configurable Security Policies*[27], a user-configurable policy that is part of the browser. The policy allows the user to explicitly enable or disable certain capabilities for specific internet sites. An example is the option to disallow a certain site to open a popup window. Some parts of this idea have also been implemented in the Security zones of Internet Explorer.

The policies and enforcement mechanism offered by this technique resemble WebJail. The major difference is that these policies are user-configurable, and thus not under control of the integrator. Additionally, the policies do not support a different set of rules for the same included content, in two different scenarios, whereas WebJail does.

9. CONCLUSION

In this paper we have presented WebJail, a novel client-side security architecture to enables least-privilege integration of third-party components in web mashups. The WebJail security architecture is compatible with legacy mashup components, and allows the direct delivery of components from the service providers to the browser.

We have designed a secure composition language for WebJail, based on a study of security-sensitive operations in HTML5 APIs, and achieved full mediation by applying deep aspect weaving within the browser.

We have implemented a prototype of WebJail in Mozilla Firefox 4.0, and applied it successfully to mainstream platforms such as iGoogle and Facebook. In addition, we have evaluated the performance of the WebJail implementation using micro-benchmarks, showing that both the page load-time overhead ($\pm$7ms) and the execution overhead of a function advised with a whitelist policy ($\pm$0.1ms) are negligible.

10. ACKNOWLEDGMENTS

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

The authors would also like to thank Maarten Decat and Willem De Groef for their contribution to early proof-of-concept implementations [8, 33] to test the feasibility of the presented research.

11. REFERENCES

[1] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*,

52:83–91, June 2009.

[2] P. Bisht and V. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *5th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assesment*, July 2008.

[3] D. Crockford. ADsafe – making JavaScript safe for advertising. `http://adsafe.org/`.

[4] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *15th Nordic Conference in Secure IT Systems (NordSec 2010)*. Springer, 2011.

[5] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Lecture Notes in Computer Science*, volume 5965, pages 18–34. Springer Berlin / Heidelberg, February 2010.

[6] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. In V. Atluri and C. Diaz, editors, *Computer Security - ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 100–116. Springer Berlin / Heidelberg, 2011.

[7] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A security analysis of next generation web standards. Technical report, G. Hogben and M. Dekker (Eds.), European Network and Information Security Agency (ENISA), July 2011.

[8] M. Decat. Ondersteuning voor veilige Web Mashups. Master's thesis, Katholieke Universiteit Leuven, 2010.

[9] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. *2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[10] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[11] Google. Google Latitude. `https://www.google.com/latitude/`.

[12] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[13] 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.

[14] Involver. Tweets To Pages. `http://www.facebook.com/TweetsApp`.

[15] Jacaranda. Jacaranda. `http://jacaranda.org`.

[16] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[17] Z. Li, K. Zhang, and X. 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, 28 2010-july 1 2010.

[18] 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*, ASIACCS '10, pages 15–23, New York, NY, USA, 2010. ACM.

[19] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *The 15th Nordic Conf. in Secure IT Systems. Springer Verlag*, 2010.

[20] G. Maone. Noscript 2.0.9.9. `http://noscript.net/`, 2011.

[21] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[22] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`.

[23] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[24] 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*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[25] Programmable Web. Keeping you up to date with APIs, mashups and the Web as platform. `http://www.programmableweb.com/`.

[26] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[27] J. Ruderman. Configurable Security Policies. `http://www.mozilla.org/projects/security/components/ConfigPolicy.html`.

[28] J. Samuel. Requestpolicy 0.5.20. `http://www.requestpolicy.com`, 2011.

[29] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[30] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium*, Aug. 2010.

[31] The FaceBook Team. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`.

[32] W3C. W3C Standards and drafts - Javascript APIs. `http://www.w3.org/TR/#tr_Javascript_APIs`.

[33] Willem De Groef. ConScript For Firefox. `http://cqrit.be/conscript/`.

[34] M. Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Main`, 2010.