Crowdsourced Web Augmentation: A Security Model

Cristóbal Arellano, Oscar Díaz, and Jon Iturrioz

ONEKIN Research Group, University of the Basque Country,
San Sebastián, Spain
{cristobal.arellano,oscar.diaz,jon.iturrioz}@ehu.es
http://www.onekin.org/

Abstract. Web augmentation alters the rendering of *existing* Web applications at the back of these applications. Changing the layout, adding/removing content or providing additional hyperlinks/*widgets* are examples of Web augmentation that account for a more personalized user experience. *Crowdsourced* Web augmentation considers end users not only the beneficiaries but also the contributors of augmentation scripts. The fundamental problem with so augmented Web applications is that code from numerous and possibly untrusted users are placed into the same security domain, hence, raising security and integrity concerns. Current solutions either coexist with the danger (e.g. *Greasemonkey,* where scripts work on the same security domain that the hosting application) or limit augmentation possibilities (e.g. *virtual iframes* in *Google's Caja,* where the widget is prevented from accessing the application space). This work introduces *Modding Interface*s: application-specific interfaces that regulate inflow and outflow communication between the *hosting code* and the *user code*. The paper shows how the combined use of sandboxed *iframes* and "modding-interface" *HTML5 channels* ensures application integrity while permitting controlled augmentation on the hosting application.

Key words: Augmentation, Sandbox, JavaScript, Crowdsourcing, Web2.0

1 Introduction

The evolution of Web applications can be staged based on the degree of layman's involvement. Web 1.0 limits laymen activities to mainly reading and form filling. Next, Web 2.0 puts *content authoring* in the user's hand: blogging, tagging or wiki editing are nowadays common practices. This work is about the last frontier of layman participation: *application authoring.* The challenge is for end users to provide their own functionality on top of existing Web applications. Mashups [19] and Web augmentation [3] illustrate this approach. Mashup techniques are available for laymen to build *new* applications out of existing Web resources (e.g. APIs, RSS feeds). By contrast, Web augmentation does not create a new application. Rather, the rendering of the hosting application is augmented to change the user experience. An example is the *Skype* add-on, a plugin that turns any phone number found in a web page to a button that launches *Skype* to call that number. No new application is created. Rather, the hosting application is augmented with a *Skype* button.

This paper focuses on *crowdsourced* Web augmentation whereby end users are not only the beneficiaries but also the contributors of augmentation scripts. *Greasemonkey* scripts illustrate this approach that permits end users change the content/rendering/layout of the current page on the fly [1]. With over 32,000,000 downloads, *Greasemonkey* evidences the success of this approach.

Crowdsourced augmentation implies user-provided code (i.e. the scripts) to co-exist with hosting code (e.g. the *HTML page*). This is risky. Placing different resources for numerous and possibly untrusted or malicious sources into the same security domain raises security and integrity concerns. Threats include: creation of/redirection to phishing pages, stealing history information (or sensitive data stored on either pages or cookies), or port scanning upon the user's local network [10]. Traditionally, the solution is to place content from multiple untrusted sources in different security domain. If scripts are not in the same domain, *JavaScript*'s Same Origin Policy prohibits them from communicating with each other. This ensures application integrity but at the expense of limiting the communication between the script and the hosting document. This is a severe limitation for augmentation where the added value comes from the amendment being seamlessly and interspersedly intermingled with the hosting code. **The challenge is then, balancing security *versus* augmentation expressiveness.**

In a previous work [6], we introduced the notion of *"Modding Interface" (MI)* as a means to isolate user scripts from changes in the hosting application. The aim was that changes in the rendering of the hosting application (a certainty in the always-evolving Web world) do not make scripts stop working. From the script perspective, *MI* provides a stable base on which to set the script. However, from the application viewpoint, interfaces also prevent scripts from peering at the hosting code. Hence, even if the application never changes, there is still a case for *MI* as a means to protect the hosting code.

This paper introduces an MI-based architecture for safe co-existence of hosting code and augmentation scripts. Set and managed by the hosting application, this interface regulates both the outflow (i.e. what "hosting data" can flow to augmentation scripts) and the inflow (i.e. what "hosting rendering aspects" can be subject to augmentation). Akin to the *JavaScript* programming model, this interface is event-based. Traditionally, scripts have open access to hosting rendering through *DOM events*. Now, scripts are "sandboxed" so that interaction can only be through *MI events*. Notice however, that no change is needed in how scripts are programmed. Rather than subscribing to *DOM events*, augmentation scripts subscribe to *MI events*. Through a running example, the paper introduces the notion of *MI*, the architecture, and discusses on the sought balance between security and augmentation expressiveness. Next section introduces the sample case.

2 Crowdsourced Augmentation: A Sample Case

Conferences addressing Web issues can tap on their attendees to enhance the conference site itself. The vision is to regard the conference site as a platform for attendees to enhance. As an example, consider the conference website for *ICWE'09*. Figure 1 (left side) depicts a screenshot for the page on accepted submissions, located at

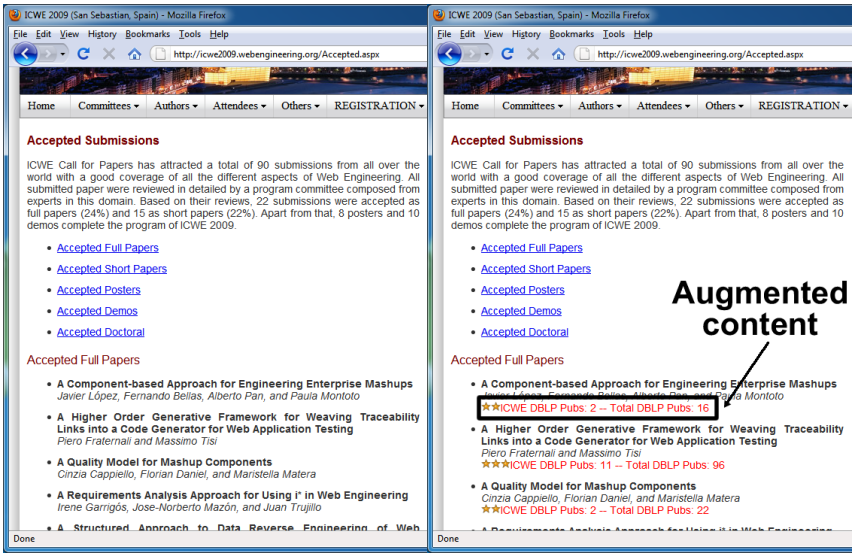

Fig. 1. Raw page (left side) vs. Augmented page (right side).

http://icwe2009.webengineering.org/Accepted.aspx. On deciding which presentations to attend, an attendee can augment this content with data obtained from Michael Ley's DBLP site[1] so that each accepted paper is augmented with data about previous publications from the paper's authors. To this end, the attendee writes the *dblpFigures* script[2] (see Figure 2, left side). The outcome (see Figure 1, right side) shows how *"host markup"* is intermingled with *"augmented markup"* produced by the script.

So far, the blend of host markup and augmented markup is conducted at the client through a *weaver*. The *weaver* is realized as a browser plugin[3]. The *weaver* places both markups in the same domain so that user scripts can react to events raised by the hosting application. The process is broadly illustrated for the script in Figure 2, left side:

- interacting with a page triggers UI events (e.g. *load*),
- the script *reacts* to this event by triggering a handle (e.g. *"init"*, line 19). An event is bound to a handler through the *addEventListener()* function (line 38),
- a handler can access any node of the page (using *DOM functions* such as *"document.evaluate()"* in lines 21-23 and 26-28). It can also create *HTML* markup (e.g. *dblpFiguresPanel* in lines 10-17),
- a handler can also *change* the *DOM* structure at wish by injecting *HTML* markups. In the example, a *dblpFiguresPanel* markup is injected at the point identified by an *XPath* expression on the *DOM* structure (i.e. the injection point). *DOM functions*

[1] http://dblp.uni-trier.de/db/index.html

[2] Available at *http://userscripts.org/scripts/source/76472.user.js.*

[3] Script *weavers* are available for Firefox (e.g. *Greasemonkey*), Internet Explorer (e.g. *IE7Pro* or *Turnabout*), Safari (e.g. *SIMBL+GreaseKit*). In Opera and Google Chrome, it is supported natively.

Fig. 2. Two versions of the *dblpFigures* script: using *DOM Events* (left side) vs. using *Conceptual Events* (rigth side).

are used for this purpose (e.g. *"papers[i].appendChild(dblpFiguresPanel)"* in line 34).

This process takes place at the client. The hosting application is completely unaware of this process: no responsibility is taken on certifying or disseminating augmentation scripts among its users. Script safety is not validated, hence, script users are exposed to malware.

This is certainly bad news for users but so is it for Web application owners. Although augmentation can threat the business models of some sites (e.g. by removing banners), in other cases, augmentation accounts for honest enhancements that serve a small set of users the application cannot afford to support their requirements, but leaves external users to fill the gap. After all, popular sites such as *Facebook*, encourage their users to build and share *Facebook applications* on the certitude that this increases the stickiness and usefulness of the site [9,12]. Customer loyalty, engagement and satisfaction are among the rewards. The vision is to create an open ecosystem between the hosting application and the augmentation contributors. However, this vision is undermined by the lack of an architecture where the hosting code can safely coexist with user-provided scripts. Next section looks at related work on this issue, mainly related with *widgets*.

3 Related Work

Certification Approach. *Widget* code is certified. External parties need to get validated before their widgets being made accessible to customers. This approach is exhibited by component-based development [16]. However, it imposes an important burden to third parties. This contrasts to the openness and freewill that characterise crowdsourcing. As a Web master, it is in your own benefit to reduce the hurdles for offering contributions, more to the point, if no financial incentives exist. Reducing the contribution hurdles calls for run-time, automated solutions that permit uploading user contribution while preserving the integrity of the hosting application. That is, *the application architecture itself should ensure this property*.

Iframe-jail Architecture. *Widget* code is isolated. The *widget* is loaded inside of its own *iframe* tag [8]. The *src* attribute of each iframe is a randomly generated subdomain of the hosting application. Being in different domains, *JavaScript*'s Same Origin Policy ensures that each code runs in complete isolation. Isolation brings security but also severely limits the interconnection of the *widget* with the hosting application.

Sanitization Architecture. *Widget* code is sanitized. *Widgets* run in the very same domain that the hosting code. This imposes *widgets* should first go sanitized. That is, *widgets* are compiled into "safe *widgets*" by removing and restricting some *JavaScript* functions. Because the code of the compiled *widget* is not pure *JavaScript*, an interpreter is needed to run it. Once sanitized, the host functionality and the *widget* are executed in the same domain. As an additional precaution, the *DOM API* is re-written (i.e. *tamed API*) to prevent *widget* output to expand over the assigned area (the "virtual iframe" in *Google*'s parlance). This approach is followed by *MS' Web Sandbox* [4] and *Google's Caja* [10].

Validation Architecture. *Widget* code is validated, i.e. code is checked for unsafe instructions. A list of unsafe instructions can be found at *Dojo*'s home page [7]. If valid, *widget* markup is constrained to a "virtual iframe" using a *tamed API* ("sandbox" in *Dojo*'s terminology).

Figure 3 depicts an architecture along the lines of the aforementioned approaches. *Widgets* are kept at the server. *Google/Microsoft* first sanitizes the *widget*'s code, and then, makes it available for the application to load at runtime. By contrast, *Dojo* moves the validation process at the client: the *widget* is loaded, and then, validated. If passed then, the *widget* is ready for enactment.

However, augmentation scripts pose more stringent demands than *widgets*. According to Wikipedia, a *widget* "is a portable chunk of code that can be installed and executed within any separate *HTML*-based web page by an end user without requiring additional compilation". *Widgets* are Web components (i.e. potentially reusable in different applications) and "single-markuped" (i.e. only one piece of markup is delivered). By contrast, the scope of augmentation scripts can be a single application, and scripts can deliver a set of markups. Additionally, *widgets* are thought to be rendered in a pre-set cell. This contrast with the dynamic binding that characterise the selection of the place where the script output is placed. This puts aside the *Caja* approach, where *widgets* can only manipulate a bounded portion (virtual iframe). Additionally, *Caja* widgets are coded in *Valija* (a subset of *JavaScript*) but debugged in *Cajita* (the compiled code in which Valija is compiled to). Since bugs are noted in Cajita, the

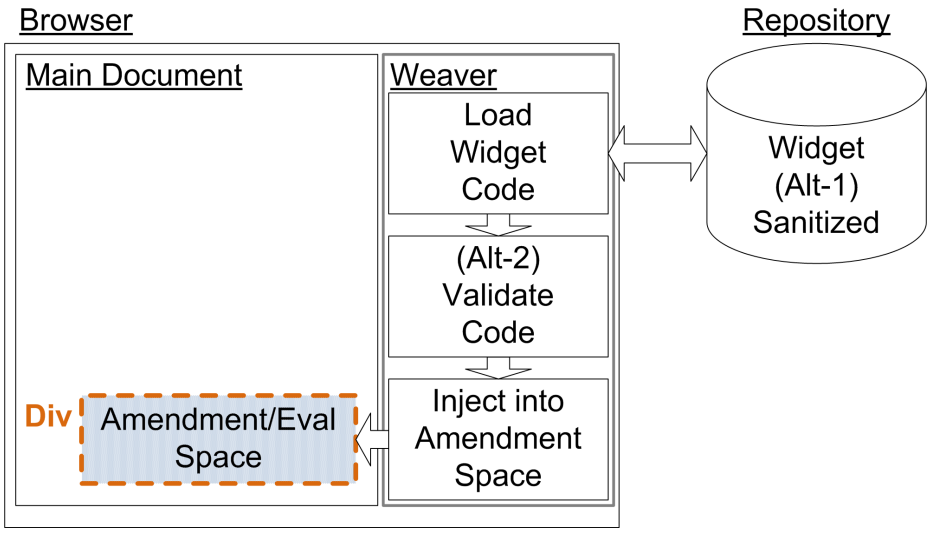

Fig. 3. An architecture for *widget* secure execution: sanitization approach ("Alt-1") and validation approach ("Alt-2").

programmer is forced to move back and forth between *Valija* and *Cajita* to debug the code. Finally, *Valija* and its *tamed API* are built on top of *HTML* and *JavaScript* (*EcmaScript5*). Upgrades on these standards can require new amendments to *Valija* and its *tamed API*. Our approach departs from creating a new framework for safely executing third-party code but to use the *HTML5* security characteristics. This is the *MI Architecture*[4].

4 The Modding-Interface Architecture

Figure 4 outlines the runtime evolution of a document with embedded scripts. On loading, the document becomes a *DOM tree*. Initially, *DOM nodes* stand for the raw content of the page. Additionally, some nodes contain "cells" (denoted as doted-lined rectangles in Figure 4). A cell is realized as either an *HMTLDivElement* (i.e. <*div*> *HTML tag*) or an *HTMLIFrameElement* (i.e. <*iframe*> *HTML tag*) element that holds the script. Enacting the script can result on augmenting the *DOM tree* (denoted as a dot-filled circle in Figure 4). This figure illustrates the existence of two spaces: "the eval space" where the script is enacted (doted-lined rectangles), and "the amendment space" where the script markup is placed. In widget-oriented architectures both spaces coincides.

[4] The term "modding" is borrowed from the video-game industry where mods are introduced by vendors for players to tune the appearance, weapons, and even the strategy of the game [14]. Similar terminology is used for cars with a similar meaning: product personalization by the customers themselves.

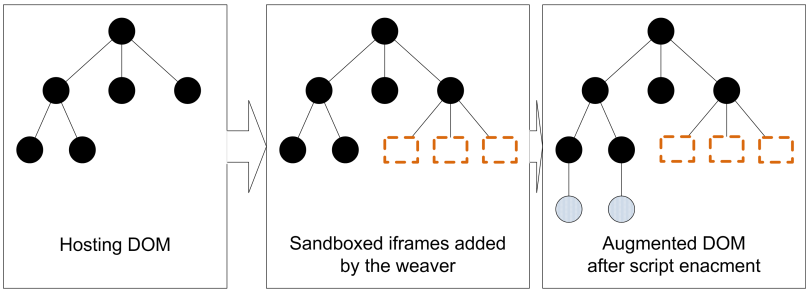

Fig. 4. Augmentation at run-time: *DOM tree* evolution.

The *Modding-Interface Architecture (MI Architecture)* (see Figure 5) clearly distinguishes between the amendment space and the eval space. The amendment space is contained within the hosting document. The eval space is placed within an *"iframe jail"*. The eval space is sandboxed from the hosting document so that access is not permitted. The novelty comes from the communication model. A publish/subscribe communication model regulates the interaction between the eval space and the amendment space. A *Modding Interface* describes the messages permitted between these two spaces. Being event-driven, a *weaver* regulates publish/subscribe messages. However, and unlike *Greasemonkey*-like approaches, now the *weaver* is part of the hosting application itself. No browser plugin is required.

Therefore, engineering a Web application for augmentation requires (1) a *Modding Interface* as a means to preserve application integrity and, (2) a generic *weaver* that mediates between the amendment space and the eval space. Next sections address these topics.

5 Modding Interface Specification

This interface encapsulates the hosting code. It regulates both (1) what data can be obtained and (2), what amendments are permitted on the hosting code. Interfaces are commonly specified in terms of operations defined upon data types. However, *JavaScript* favours event-based programming, i.e. handlers are associated to UI events. Unlike operations, handlers are not explicitly called but *triggered* when the associated event occurs. Akin to the *JavaScript* approach, *Modding Interfaces* are to be described in terms of events rather than operations, but they will act upon *concepts* (e.g. *Paper*) rather than *DOM nodes*. In this way, scripts can subscribe to the event *loadPaper* (rather than the *DOM event*, *load*) and obtain *Paper* data as event payload rather than scraping the *DOM tree*. Scripts can also publish the event *appendChildPaper* to add an *HTML fragment* as a child of a *Paper* (rather than using an *XPath expression*).

The right side of Figure 2 shows the *dblpFigures* augmentation script but now using *Conceptual Events*. The augmentation logic is the same (lines 10-17). Differences rest on (1) *HTML* scraping being substituted by event parameter recovering (lines 21-23) and, (2) amendment spaces described by the point where *Conceptual Events* occur (lines 28-32) rather than *XPath expressions*.

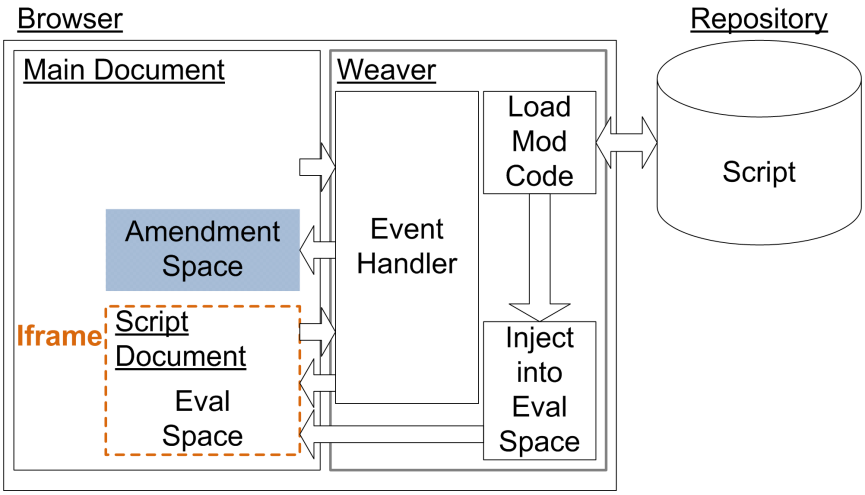

Fig. 5. The *Modding-Interface* Architecture.

Therefore, a *Modding Interface* encapsulates a Web application in terms of its concepts, and provides a set of services to "read" and to "write" these concepts. The "read" part realizes the required interface as the set of events the interface just signals but leaves to the scripts the event processing (a.k.a. **Publishing Events**). As for the "write" part, it identifies the amendment space in terms of concept occurrences rather than through *DOM nodes*. Rather than using *XPath* on *DOM trees*, the amendment space is identified by the target of **Processing Events**. For instance, the event *appendChildPaper* indicates that *Paper* denotes an amendment point. Scripts can now raise *appendChildPaper* to inject its *HTML markup* into this amendment point. *Processing Events* then realize the provided interface[5].

The *Modding Interface* is described as an *OWL* document [15]. *OWL* permits to describe concepts, properties related to these concepts and associations between concepts. The aim of *OWL* is to provide a way to exchange information between applications with a specific semantic. Such aim aligns with our purposes. Next paragraphs describe the *"Concepts", "PublishingEvents"* and *"ProcessingEvents"*[6] for our sample case (see Figure 6).

Concepts. The *ICWE* Website is seen as renderer of a set of concepts: *ConferenceEvent*, *Paper*, *Person*, etc. (lines 14-16). The <*Ontology*> element contains the description for these concepts. Concept description includes <*DataTypeProperty*> and <*ObjectProperty*> (i.e. title and author (lines 17-24)). It is possible to import the ontology. For conference description, an external ontology is available at [13].

[5] The terminology of "processing events" and "publishing events" is widely used for event-based components such as portlets [11].

[6] A schema for defining *Modding Interfaces* is available at *http://userscripts.org/scripts/source/61129.user.js*.

```
<!DOCTYPE rdf:RDF [
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
<!ENTITY owl "http://www.w3.org/2002/07/owl#">
<!ENTITY swc "http://data.semanticweb.org/ns/swc/ontology#">
<!ENTITY swrc "http://swrc.ontoware.org/ontology#">
<!ENTITY mod "http://userscripts.org/scripts/source/61129.user.js#">
<!ENTITY icwe "http://icwe2009.webengineering.org/">
]>
<rdf:RDF xmlns:base="&icwe;" xmlns:owl="&owl;" xmlns:rdf="&rdf;"
xmlns:rdfs="&rdfs;" xmlns:mod="&mod;">
<owl:Ontology>
 <owl:Class rdf:about="&swc;ConferenceEvent"/>
 <owl:Class rdf:about="&swc;Paper"/>
 <owl:Class rdf:about="&swrc;Person"/>
 <owl:DatatypeProperty rdf:about="&swrc;title">
  <rdfs:domain rdf:resource="&swc;Paper"/>
  <rdfs:range rdf:resource="&xsd;string"/>
 </owl:DatatypeProperty>
 <owl:ObjectProperty rdf:about="&swrc;author"/>
  <rdfs:domain rdf:resource="&swc;Paper"/>
  <rdfs:range rdf:resource="&swrc;Person"/>
 </owl:ObjectProperty>
</owl:Ontology>

 <mod:PublishingEvent rdf:ID="loadPaper">
  <mod:payloadType rdf:resource="&swc;Paper"/>
  <mod:uiEventType rdf:resource="&mod;load"/>
  <mod:cancelable>false</mod:cancelable>
 </mod:PublishingEvent>

 <mod:ProcessingEvent rdf:ID="appendChildPaper">
  <mod:payloadType rdf:resource="&mod;HTMLSpanElement"/>
  <mod:operationType rdf:resource="&mod;appendChild"/>
  <mod:targetConcept rdf:resource="&swc;Paper"/>
 </mod:ProcessingEvent>
</rdf:RDF>
```

Fig. 6. *ICWE Website Modding Interface*.

Publishing Events. These events notify "concepts" being delivered by the Web application. In other words, the payload of a *Publishing Event* is a concept of the Web application at hand. But events are happenings of interest, i.e. they are instants of time. An event cannot be described just by its associated concept but needs to include what happens to this concept, e.g. loading, selecting, de-selecting the concept, etc. Therefore, *Publishing Events* are described by both the event payload (*"**payloadType**"* property), and the time when the event arises (*"**uiEventType**"* property). The values for *"uiEventType"* are taken from the *W3C's DOM Level 2 Events* specification [17]. Additionally, a *"**cancelable**"* property is added that mimics the namesake property available for *JavaScript* events whereby an event is liable to be called off by a handler so that the occurrence is no longer propagated to other handlers. As specified in Figure 6 (lines 27-31), *loadPaper* is introduced as a *Publishing Event* to occur every time a *Paper* is loaded.

Processing Events. A website determines *what* can be augmented but leaves to the scripter to decide *when* and *how* is to be augmented. The *what* refers to the concept that denotes the amendment space ("***targetConcept***" property). But being a layout issue, the concept alone is not enough. We need to indicate the position w.r.t. the concept ("***operationType***" property) through a reference to the *W3C's DOM Level 2 Core* operations [17]. Figure 6 shows an example where the concept *Paper* is used to pinpoint the amendment space. The *"operationType"* indicates that augmented content is to be rendered as children of the *Paper* at hand (i.e. *appendChildPaper* (lines 33-37)).

As for the *how*, traditional scripts can inject any *HTML fragment* on the premise that the disclosure of the page implementation makes them acknowledgeable about what would be the right fragment code. This approach may work for simple pages but is hardly scalable as pages become more complex. We cannot rely on end users peering on *HTML code* to ascertain what would be a wrong fragment to be injected. We resort to *HTML types* [17]. The augmentation markup should be compliant to an *HTML type* ("***payloadType***" property). This type restricts how rendering can be augmented. For instance, augmenting a *"Paper"* is set to be of type *HTMLSpanElement,* meaning that augmentation markup on *Papers* need to be compliant with this type. This introduces a type-like mechanism for regulating augmentation to existing Web application. The *weaver* can then check whether this *payloadType* is fulfilled, and if not so, ignores the script markup but still renders the rest of the page. This is akin to browser practices where wrong *HTML tags* do not prevent the browser from rendering the page.

6 A Weaver for Augmentation Scripts

The *weaver* mediates between the main document and the script document (see Figure 5). Specifically, the *weaver*'s duties include (1) loading the augmentation scripts for the current user, and (2), managing *Conceptual Events*. This section outlines the implementation of these functions. The code has been tested for *Google's Chrome*, using extensively *HTLM5* new features [18].

Loading Augmentation Scripts. Customers of the Web application have previously registered their interests in some augmentation scripts. These preferences are kept locally through a *localStorage* variable at the browser: *augmentationConfiguration*. Scripts are kept at the server. Figure 7 lists the *weaver*'s code that loads the scripts.

On loading the Web application, the *weaver*'s first duty is to load the script identifiers kept at *augmentationConfiguration* (lines 2-3). For each script, the *weaver* creates an *iframe* (line 7-11). An *iframe* holds a generic document (*src* attribute) that is parameterized with the identifier of the script at hand. This document has no rendering counterpart (i.e. *"display:none"*). *Iframes* are sandboxed. When the *iframe* is added to the page (line 12), the script is downloaded and evaluated. Being sandboxed, the script cannot access the hosting page (i.e. the script cannot subscribe to UI events from the main document). Interactions are restricted to occur through a channel (line 14-17). A message channel is an *HTML5* object that enables the direct communication of independent pieces of code (e.g. running in different browsing contexts). This interaction follows a publish/subscribe pattern based on *Conceptual Events*.

```
// Load configuration
var installedScripts=localStorage.getItem("augmentationConfiguration");
installedScripts=installedScripts?JSON.parse(installedScripts):[];
// Load scripts
for(script in installedScripts){
 // Create of sandbox
 var iframe=document.createElement("iframe");
 iframe.src="http://icwe2009.webengineering.org/mod_document.html?"+
            installedScripts[script].id;
 iframe.style.setProperty("display","none","important");
 iframe.sandbox="allow-scripts";
 document.body.appendChild(iframe);
 // Initialize communication
 var channel = new MessageChannel();
 iframe.addEventListener("load",function(){
    iframe.contentWindow.postMessage("initChannel",[channel.port2],
    "http://icwe2009.webengineering.org/"); },true);
  ...
```

Fig. 7. *Weaver*'s code: loading augmentation scripts.

Managing Conceptual Events. When the *iframe* space is initialized, the main document and the script document are ready for exchanging *Conceptual Events*. However, these *Conceptual Events* are to be produced/handled by the *weaver*. The *weaver* has two main duties: raising *Publishing Events* in the main document, and handling *Processing Events* as signalled by *script* documents.

The process goes as follows. The UI event (e.g. loading a page) is first notified to the *weaver*. The *weaver* constructs and raises the *Conceptual Event* (e.g. *loadPaper*) along the indications of the *Modding Interface*. *Conceptual Events* are captured by the *script* that recovers the event payload, constructs an *HTML fragment*, and dispatches the appropriate *Processing Event* (e.g. *appendChildPaper*). This *Processing Event* is then de-constructed in terms of UI operations by the *weaver* according to the indications of the *Modding Interface*. These UI operations causes the main document (the page you see) to be augmented.

7 Discussion

7.1 Impact on security

The work's first motivation was safe coexistence of heterogeneous-sourced code. Both redirection to phishing pages or stealing sensitive data are avoided by running the script inside an *"iframe jail"*. On the other side, we can prevent port scanning and history sniffing by using the same approach as *Google Caja*: a *monkey patch* [2]. *Monkey patch* is a way to extent/modify runtime code in dynamic languages. This technique can be applied to dynamically replace/extend script functions liable to content malware with others that block such malware. Finally, browser blocking can be alleviated as in *MS' Web Sandbox*, i.e. using a QoS Layer [5]: a wrapper-like mechanism that imposes some limits on the consumption of shared resources. Exceeding these thresholds (e.g. CPU consumption) makes the script be blocked.

```
<head>
 <link rel="moddingInterface" href="/modding_interface.owl" />
 <link rel="transformation" href="/extractor.xsl" />
 <script type="text/javascript" src="/weaver.js"></script>
 ...
```

Fig. 8. Augmentation-enabled page: meta-data about the *Modding Interface*.

7.2 Impact on users

In a crowdsourcing setting, the viability of an approach heavily rests on causing minimal disturbance to the involved parties: Web application programmers and script programmers. As for the former, the *MI Architecture* imposes almost no disruption. Augmentation-enabled HTML pages differ from traditional pages in that they keep three links: two to the *MI* files, another to the *weaver* script (see Figure 8). Apart from that, these pages do not differ from "traditional pages".

From a script-programmer perspective, *MI* implies notification/publication to be based on *Conceptual Events* rather than *DOM events*. Otherwise, native *JavaScript* mechanisms are used to handle *Conceptual Events* with no variations w.r.t. traditional script development. From the start of this work, we have been very conscious about reducing the hurdles for offering contributions. Next paragraphs provide evidence that programming on top of a *Modding Interface,* causes minimal deviation from traditional practices.

Notification of Processing Events. *JavaScript* follows an event-based approach where handlers can be associated with *DOM-based events*. Operations are available for creation of event occurrences (e.g. *createEvent("MouseEvents")*), assigning the payload to an occurrence (e.g. *initMouseEvent("eventInstance", "eventParameters")*), or raising the event manually (e.g. *dispatchEvent(eventOccurrence)*). Raising of *Conceptual Events* uses these standard *JavaScript* operations. Back to our running example, a *dblpFiguresPanel* (i.e. an *HTML fragment*) is to be injected as a child of a *Paper*. Figure 2 (right side) show the code along the following pattern: *createEvent* (lines 28-29), obtain concept (line 21), *initProcessingEvents* (line 30-31), *dispatchEvent* on this concept (line 32). This is standard *JavaScript* code. The only difference with traditional scripting is that now the injection point is not a *DOM node* but the current concept. This current concept is to be obtained through a *Publishing Event*.

Subscription to Publishing Events. *JavaScript* achieves event subscription by registering a handler through the *addEventListener* method. Subscription to *Conceptual Events* is accomplished in the very same way: associating a handler. For instance, instruction (line 38 in Figure 2 (right side)) *"doc.addEventListener("loadPaper", init, true)"* adds a handler to the *loadPaper* event, i.e. occurrences of *loadPaper* will trigger the *init()* function. The difference rests on handlers being associated to the whole document (i.e. variable *doc*) rather than to *DOM nodes* (e.g. a *checkbox*). This highlights the fact of events being risen by acting on *Papers* rather than on *DOM nodes* (i.e. the circumstantial representation of these *Papers*).

7.3 Impact on performance

All our measurements are realized in Windows 7 x64 running on Intel Core2 Duo 2.20 GHz CPU with 4GB of memory. The experiments have been realized with a domestic 6Mbps WIFI LAN bandwidth.

Loading time. The *Greasemonkey* architecture keeps scripts at the client. So, no loading penalty at the time the script is enacted. By contrast, our approach makes scripts a valuable asset of the Web application which becomes a partner on disseminating these resources among its user base. Therefore, the *MI Architecture* maintains scripts at the server as site resources. When application pages are loaded, so are the appropriate scripts (as any other page resource such as associated images). Compared with *Greasemonkey*, this certainly imposes an overhead. However, script files tend not to be very large, and its cost is similar to loading a *"jpg"* thumbnail file. Additionally, the *weaver* and *Modding Interface* file are loaded on accessing the first page. The size of the *weaver* file is 3.8kb (no obfuscated) which approximately accounts for a 100 millisecond delay (less if the *weaver* is cached by the browser). The size of the *Modding Interface* for a given page is similar to a script. On the upside, this approach frees users from installing any plugin (as it is the case for *Greasemonkey*).

Enactment time. Script enactment takes place at the client (no server impact). *Greasemonkey* scripts act upon *DOM events*. By contrast, interface-aware scripts rest on *Conceptual Events*. This imposes an indirection: *Conceptual Events* need first to be (de)constructed from *DOM events* and send over the channels that connect the script space with the hosting application space. A first experiment has been conducted for the *dblpFigures* sample realized as both a *Greasemonkey* script and an interface-aware script. The results show that the indirection and communication accounts for a delay of 30 and 2 milliseconds, respectively, when compared with the *Greasemonkey* alternative (i.e. acting directly upon *DOM events*).

8 Conclusions

Web augmentation requires hosting markup and user-provided markup to run together. This raises integrity issues where malware can cause important damages on both end users and the reputation of hosting applications. We tackled this issue by combining *"iframe jails"* and "modding-interface" *HTML5 channels*. The approach benefits Web applications that now can be safely augmented. So does for script contributors that now achieve greater visibility by having their scripts uploaded at the hosting application.

Safe coexistence of external code is just one of the issues raised by crowdsourcing augmentation. The construction of communities around Web applications implies promoting/rewarding contributions, disseminating contributions through the Web application, facilitating end users to suggest augmentations, and so on. In the same way that Web2.0 APIs open data silos to achieve application composition at the back-end, we envision *Modding Interfaces* "to open" application markup to crowdsourced, front-end composition.

Acknowledgments. This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2008-06507-C02-01/TIN (MODELINE). Arellano has a doctoral grant from the Spanish Ministry of Science & Education.

References

1. Greasemonkey Homepage. http://www.greasespot.net/.
2. Monkey patch. http://en.wikipedia.org/wiki/Monkey_patch.
3. N. O. Bouvin. Unifying Strategies for Web Augmentation. In *Proceedings of the 10th ACM Conference on Hypertext and Hypermedia (HYPERTEXT'99)*, 1999.
4. Microsoft Corp. Microsoft Web Sandbox. http://websandbox.livelabs.com/.
5. Microsoft Corp. Microsoft Web Sandbox: QoS Layer. http://websandbox.livelabs.com/documentation/vm_qos.aspx.
6. O. Díaz, C. Arellano, and J. Iturrioz. Layman Tuning of Websites: Facing Change Resilience. In *the 17th International Conference on World Wide Web (WWW '08)*.
7. Dojo. Secure Mashups with dojox.secure. http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/.
8. B. Hoffman and B. Sullivan. *AJAX Security*, chapter 11 - Web Mashups and Aggregators, pages 295 – 327. Addison-Wesley, 2007.
9. Facebook Inc. Apps on Facebook. http://developers.facebook.com/docs/guides/canvas/.
10. Google Inc. Google Caja: A source-to-source translator for securing Javascript-based web content. http://code.google.com/p/google-caja/.
11. JCP. JSR 168: Portlet Specification Version 1.0, 2003. http://www.jcp.org/en/jsr/detail?id=168.
12. J. Maver and C. Popp. *Essential Facebook Development: Build Successful Applications for the Facebook Platform*. Addison-Wesley, 2009.
13. K. Möller, S. Bechhofer, and T. Heath. Semantic Web Conference Ontology, 2007. http://data.semanticweb.org/ns/swc/ontology.
14. W. Scacchi. Computer Game Mods, Modders, Modding, and the Mod Scene. *First Monday*, 15, 2010.
15. M. K. Smith, C. Welty, and D. L. McGuinness. OWL Web Ontology Language Guide. W3C Recommendation, 2004. http://www.w3.org/TR/owl-guide/.
16. J. Voas. Certification: Reducing the Hidden Costs of Poor Quality. *IEEE Software*, 16:22–25, 1999.
17. W3CDOMWG. W3C DOM Level 2. http://www.w3.org/DOM/DOMTR#dom2.
18. W3CHTML5WG. HTML5. http://www.w3.org/TR/html5/.
19. J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12:44–52, 2008.