Server-driven Outbound Web-application Sandboxing
FP7-ICT-2009-5, Project No. 256964

https://www.websand.eu

Deliverable D4.1
Secure composition policies

Abstract

This deliverable reports on the exploration of secure composition policies in the context of mashup compositions with third-party components. The presented security policies restrict the behavior of third-party components and put fine-grained constraints on their interaction capabilities.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *30.09.2011* Due on: *M12*
Editors: *Lieven Desmet and Frank Piessens* Total pages: *73*

List of Contributors:
Philippe De Ryck, Lieven Desmet, Martin Johns, Wouter Joosen, Jonas Magazinius, Frank Piessens, Phu H. Phung, Andrei Sabelfeld, David Sands, Steven Van Acker

Project details

Start date: *October 01, 2010* Duration: *36 months*
Project Coordinator: *Martin Johns*
Partners: SAP, Siemens, CHALMERS, KUL, UNI PASSAU

Executive summary

In this deliverable, relevant secure composition policies are identified in web mashup applications and are described in more detail.

As a first step, to better understand the impact of running arbitrary JavaScript code, the upcoming HTML5 specification and accompanying APIs have been studied, and a list of security-sensitive operations available to mashup components has been enumerated (Section 2). This includes, among others, access to sensitive information in the DOM, the history and cookies, sensitive device information, inter-frame communication, cross-domain communication, and various features in media, UI, and rendering. This list of security-sensitive operations is a key ingredient in defining privileges granted to mashup components as part of the secure composition policies.

Next, the seven most relevant composition scenarios selected from deliverable D1.2 (i.e. Google Maps, Facebook application, Online advertisement, Interactive Avatar, third-party payment, third-party authentication and the Holiday Picture mashup), are described in more detail in Section 3. In addition to the informal functional description, the relevant technical data on how the scenario is constructed is described, as well as an analysis of security requirements for the particular scenario.

In Section 4, two types of secure composition policies are developed. These policies include both behavioural constraints as well as communication constraints. First, the *least-privilege composition policy* enables the specification of allowed privileges granted to a mashup component. The privileges consist of the 86 previously identified security-sensitive operations, grouped in 9 separate categories. Second, to capture recurring communication patterns between cooperative mashup components (as is the case in third party payment as well as in third party authentication compositions), the more fine-grained *trusted delegation policy* between cooperative components specifies how an integrator can securely delegate control back and forth to a third-party mashup component (e.g. for 3rd party payment). In addition, the two policy types are applied on the seven small composition scenarios.

Finally, various implementation strategies have been explored to early test the implementation feasibility of the two types of secure composition policies, including the use of JavaScript wrappers, deep aspect technology in the JavaScript engine, as well as browser extensions (Section 5).

The collection of small composition scenarios, the concrete composition policies and the early exploration of implementation strategies are direct input for the server-driven enforcement in WP4, and their characteristics will further drive the selection and development of enforcement techniques.

The composition policies and enforcement techniques presented in this

deliverable have also been described in [19] (accepted at ACSAC 2011), [4] (presented at ESORICS 2011), [9] (presented at NordSec 2010), and [10] (presented at CSF 2011). For a more detailed description of the presented work we would like to refer to these papers.

Contents

List of Figures

1 Introduction

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 [16] 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.

Mashup security is based on the de facto security policy of the Web: the Same Origin Policy (SOP) [41]. The SOP states that scripts from one origin should not be able to access content from other origins. This prevents scripts from stealing data, cookies or login credentials from other sites. Additionally to the SOP, browsers also apply a frame navigation policy, which restricts the navigation of frames to its descendants [2].

The two most-widespread techniques to integrate third party components into a web mashup are through script inclusion or via (sandboxed) iframe-integration. Loading components from different origins in iframes causes them to be separated by the SOP. Using script inclusion causes the script to be loaded in the protection domain of the including page, which is a straightforward way to achieve interaction between components. Communication with the origin of the page containing the script can be achieved using the XMLHttpRequest object of the JavaScript language.

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 provides the “sandbox” attribute for the iframe element, which disables any security-sensitive feature. Specific features can be allowed by setting the value of the attribute, such as enabling script execution with the “allow-scripts” keyword. Obviously, this very coarse-grained control has only a very limited applicability in a web mashup context.

Examining the traditional techniques in the light of the previously proposed security requirements yields some interesting results. Iframes offer full separation between different origins, but not within the same origin, and provide no interaction between components. Script inclusion offers no separation at all, but provides full interaction. This interaction is not authenticated, nor can confidentiality or integrity be ensured.

As discussed in deliverable D.1.1 [39], to cope with the hybrid aggregation of content and functionality from different trust domains and the ensuing fragmentation of ownership, web mashups demand stronger separation guarantees, but also require the possibility of interaction between separated components. This includes among others inter-component behavioral restrictions (e.g. a least-privilege execution of third party components), secure cross-component interactions and secure remote communication.

One of the main challenges for the secure composition is the need to support fine-grained, expressive composition policies at the server-side. These policies needs to be at the right level of abstraction for the application developer or composer and need to control all relevant security-sensitive operations as well as cross-component and remote communication. To do so, it is important that the composition of the different components is made explicit, and that policies of the different stakeholders can be specified in terms of the components or their composition.

In this deliverable, relevant secure composition policies are identified in web mashup applications and are described in more detail. To do so, existing and emerging web standards are studied to understand the impact of running arbitrary JavaScript code, and to identify and enumerate the set of operations available to mashup components (Section 2). Next, for the seven most relevant composition scenarios, the scenario description and the security requirements has been described in detail (Section 3), and expressed as a secure composition policy (Section 4). These policies include both behavioral constraints as well as communication constraints. Finally, we have explored and experimented with various implementation strategies to acquire first insights in their feasibility and complexity (Section 5). The collection of small composition scenarios, the concrete composition policies and the early exploration of implementation strategies are direct input for the server-driven enforcement in WP4, and their characteristics will further drive the selection and development of enforcement techniques.

The composition policies and enforcement techniques presented in this deliverable have also been described in [19] (accepted at ACSAC 2011), [4] (presented at ESORICS 2011), [9] (presented at NordSec 2010), and [10] (presented at CSF 2011). For a more detailed description of the presented work we would like to refer to these papers.

2 Security-sensitive 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[27], 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.

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 1, 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.

This list of security-sensitive operations plays an important role in defining least-privilege policies for mashup component, as will be done in Section 4.

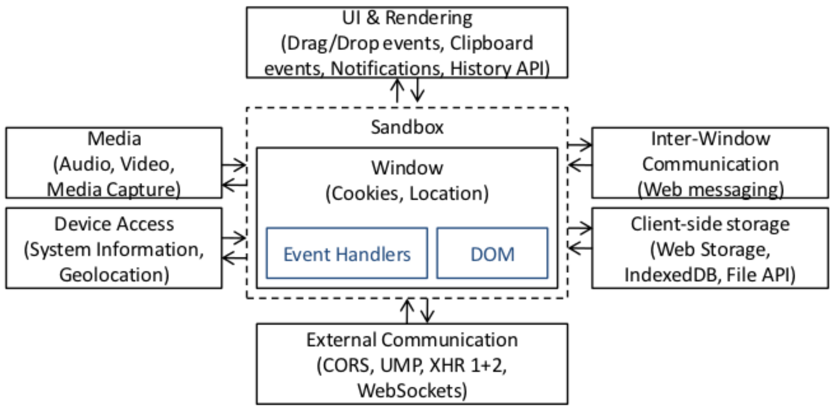

UI & Rendering
(Drag/Drop events, Clipboard events, Notifications, History API)
Sandbox
Window
(Cookies, Location)
Event Handlers
DOM
Media
(Audio, Video, Media Capture)
Device Access
(System Information, Geolocation)
Inter-Window Communication
(Web messaging)
Client-side storage
(Web Storage, IndexedDB, File API)
External Communication
(CORS, UMP, XHR 1+2, WebSockets)

Figure 1: Synthesized model of the emerging HTML5 specifications

Central in the model is the *window* concept, containing the document.

The window manifests 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 and numerous client-side APIs. The functionality offered by the emerging web specifications as JavaScript APIs is grouped into blocks based on offered functionality, and are depicted around the window in Figure 1.

2.1 Window and Sandbox

As mentioned before, the browser window and its associated window object enclose a document with a specific origin and location (a URL). A window can contain multiple documents (i.e. a browsing history) but only one of these documents can be active at any given time. Since the relation between window and document at one moment in time is one-to-one, we do not separate a window and a document when this is not relevant. Additionally, a *sandbox*[7] (shown by the dotted line in the model) can impose coarse-grained restrictions on an iframe, as mentioned in Section 1.

The two functional blocks inside the window (Event Handlers and DOM) represent two cornerstone pieces of functionality for dynamic web pages. Event handlers are used extensively to register handlers for a specific event, such as receiving messages from other windows or being notified of mouse clicks. Access to the DOM enables a script to read or modify the document's structure on the fly.

2.2 Inter-window Communication

An important aspect of composed applications is communication between several windows (e.g. sending messages between mashup components). This functional block covers window navigation and the corresponding policy, the descendant policy (HTML5). Additionally, this block includes message passing as defined by the Web Messaging specification.

The Web Messaging[28] specification defines two mechanisms for communicating between browsing contexts in HTML documents: cross-document messaging and channel messaging. The former sends a message in the form of a single asynchronous event. The latter opens a channel between two contexts, allowing the asynchronous posting and receiving of messages through the channel. Both mechanisms allow the passing of messages of any type, but every message is cloned before it is sent.

Cross-document messaging[6] allows scripts to post messages to another window. The posting of a message with the postMessage operation requires the specification of the target origin, which can either be a URI, a wildcard (*) or the current origin (/).

Channel messaging creates two ports, of which one needs to be transferred to the remote browsing context. Ports can be passed between a context using the postMessage mechanism on a window object (as used in cross-document messaging). Message ports belonging to the same channel remain entangled, thus effectively creating a channel with two endpoints. Messages can be sent over message ports using the postMessage mechanism on a port object. Note that the latter postMessage mechanism no longer requires the explicit specification of a destination origin. After use, ports should be closed explicitly to avoid needless resource consumption.

2.3 External Communication

An interactive client-side web page often requires to communicate with remote parties, for example to load contacts from an address book. This block covers the specification of XMLHttpRequest and the WebSocket API, mechanisms to communicate, and new policies to enable secure cross-origin communication (CORS and UMP).

The XMLHttpRequest[36] specification defines an API that provides scripted client functionality for transferring data between a client and a server. It offers scripts the possibility to initiate HTTP requests and process the responses. The XHR specification has evolved over time, which is why there is a Level 1 and Level 2 version. Level 1 offers same-origin communication, while level 2 extends the functionality towards cross-origin communication.

The WebSocket API[35] specification enables two-way communication with a remote host over a single HTTP connection. Using regular HTTP connections, e.g. when using XMLHttpRequests, the client must setup a new connection to the remote host for every message sent. The WebSocket API defines a way to keep the HTTP channel open and transmit/receive any number of messages while the channel is open.

CORS (Cross-Origin Resource Sharing)[21] defines a mechanism to enable client-side cross-origin requests. CORS only defines algorithms which can be followed by an implementing API, such as XMLHttpRequest Level 2[37]. The main idea behind the specification is that a client provides the server with adequate information to decide whether the requesting origin has access to the requested data or not. This decision is sent to the client, which effectively enforces this decision by either granting or denying access to the requested resource.

UMP (Unified Messaging Policy)[32] enables cross-site messaging while avoiding attacks abusing cookies and other credentials. UMP allows the resource owner to consent to cross-origin retrieval and enforces origin independent messaging. This is realized using a *uniform request* and a *uniform response*. A *uniform request* is a fully anonymized GET or POST request that does not contain any authentication information and has specific characteristics with regard to its content-type and headers. Likewise, a *uniform response* must also adhere to certain characteristics. A non-uniform response as an answer to a uniform request must be met by an error. It must not be made available to the requesting node and if it is a redirect, it must not be followed.

2.4 Client-side storage

The client-side storage specifications enable applications to temporarily or persistently store data in the client-side environment. One example is a calendar application which stores a copy of your calendar at the client-side. Various specifications are emerging, including Web Storage, IndexedDB and the File API.

The Web Storage API[34] provides each origin its separate client-side storage area for key-value pairs, similar to the cookie mechanism. A valid key is any string, including the empty string. A valid value is any piece of data supported by the structured clone algorithm. Each value is cloned when storing it and is cloned when retrieving it. Modifying the storage (setItem, removeItem, clear) also triggers a storage event to all documents that have a reference to this particular storage instance. The event contains the information about the change.

Web Storage supports two different storage areas: localStorage and sessionStorage. Local storage is persistent storage with no limited lifetime. Session storage is meant to support concurrent transactions in separate windows, a case that cookies do not support.

The Indexed Database API[29] provides a system to store *(key,value)* pairs in a persistent database. The database records are indexed and can be retrieved either by their index or by their key.

The File API[22] allows a web developer to represent, select, read and write file objects through a scripted interface on the client side. The data in each file can be accessed at a byte-level.

2.5 Device access

Several specifications introduce new APIs to retrieve client-side information, such as contextual data (e.g. geolocation) as well as system/device properties such as battery level, CPU information and ambient sensors (light, sound, movement, ...).

The Geolocation API[23] provides scripted access to current geographical location information associated with the hosting device. Geolocation offers both “one-shot” location information or repeated monitoring of location changes. The API is agnostic of the underlying location mechanism (e.g. GPS, network-based) and provides no guarantees about the accuracy of the location.

The System Information API[31] provides web applications with access to various properties of the system on which they are running, as typically available to an operating system. Examples are CPU properties, audio/video codecs or different sensors (ambient light, ambient noise, temperature, ...). The API offers one-shot access and continued monitoring, which can be canceled by the user. Currently, only reading information is supported, but a future version may offer write-support.

2.6 Media

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

A media element (audio or video) can be used to embed a video or audio object in the web page [26]. Pages can script their own controls for the stream, and the interface of media elements offers origin-independent access to the controls and the stream’s properties, such as playback information (e.g. the timeranges of the video that have actually been rendered).

The Media Capture API[30] provides programmatic access to audio, image and video capabilities of the device (via the microphone and webcam). Scripts can issue capture operations that are invoked asynchronously and return their result using a success or error callback handler. The actual capturing is handled by an external application that is part of the browser or the underlying platform. The API does not allow live capturing, as needed in interactive applications (e.g.video chat, ...).

2.7 UI & rendering

In addition to the obvious rendering of documents, the browser also enables the user to interact with the web application in numerous ways (mouse movement, entering text with the keyboard, entering new URLs, using the back and forward button, ...). Newly-added features are the use of drag-and-drop, the History Interface, Clipboard API and Web Notifications.

In HTML5, elements can use the draggable[24] attribute to indicate they can be dragged and the dropzone attribute to indicate they can receive drop events. The appropriate actions can be implemented using handlers for the different drag/drop events. The data involved in the drag and drop operation is stored in the drag data store. Drag and drop is also supported from external sources (e.g. dragging files from your desktop to a web page).

Furthermore, each browsing context has a distinct session history, that consists of a sequence of documents. One of the documents of the session history is known as the current entry. The history interface[25] allows via a JavaScript API to jump backwards and forwards, and to add and update history state.

The Clipboard API and events[20] provide programmatic access to clipboard operations such as copy, cut and paste.

Web Notifications[33] are simple notifications generated to alert the user of a webpage, and are displayed either on the webpage itself, inside the chrome of the webbrowser or even outside of the webbrowser.

3 Secure composition requirements

From the scenarios listed in deliverable D1.2 [40], the seven most representative composition scenarios for WP4 are described in more detail in this section:

1. Google Maps
2. Facebook Application
3. Online Advertising
4. Interactive Avatar
5. Third-party Payment
6. Third-party Authentication
7. Holiday Pictures Mashup

The *Google Maps* scenario is often pointed to as the *de facto* example of mashup composition. It illustrates the use of a service API and integration via script inclusion. The *Facebook Application* scenario on the other hand is a very popular example of mashup integration via iframes. Another widespread example of third-party composition is the inclusion of *online advertising* on websites. Next, the *interactive avatar* scenario explores the use of novel UI and rendering techniques in community, as will they possibly emerge in the near future. Additionally, the *third-party payment* and *third-party authentication* scenario were added to investigate more complex communication interactions between cooperative mashup components. Finally, the *Holiday Pictures Mashup* integrates service APIs from multiple service providers into a single application, and was added to explore secure composition policies for mashup compositions with multiple parties involved.

For each of the scenarios, a brief summary is given of the functional scenario, followed by a more technical description how components are loaded and combined in the composition scenarios. Finally, for each scenario an informal requirements analysis describes the stakeholders involved as well as meaningful security requirements and informal policies.

3.1 Google Maps

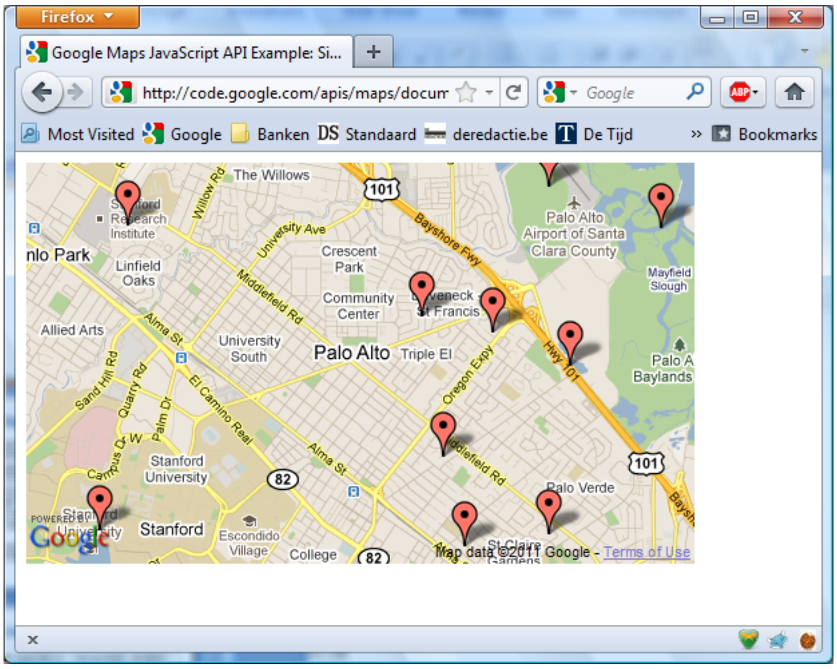

Firefox
Google Maps JavaScript API Example: Si...
http://code.google.com/apis/maps/docum
Most Visited
Google
Banken
DS Standaard
deredactie.be
De Tijd
Bookmarks
Palo Alto
Stanford University
Palo Alto Airport of Santa Clara County
Map data ©2011 Google - Terms of Use

Figure 2: Example of Google Maps integration

3.1.1 General description

Google Maps (GMaps) is the most commonly used API in mashups[16]. In this scenario, the map functionality offered by Google is combined with application-specific data, such as geometric coordinates, points-of-interest, moving GPS coordinates, or custom overlays. This data is either directly available for the integrator, or is in its turn received from a third-party service provider.

The scenario runs as follows. A user is invited to enter the name of a location. This name is then looked up in the Google Geocoding API and a set of GPS coordinates is returned. Using the GMaps API, a marker is then positioned at these coordinates on the map. Multiple queries can be performed in sequence and the previous markers are not erased.

3.1.2 Technical description

This mashup consists of a single component (the GMaps API) and some glue code, as illustrated in the annotated example code in Figure 3.

To make use of GMaps, the integrating webpage needs to load a JavaScript library using a script tag (highlighted in red in Figure 3). The JavaScript library is located at `http://maps.google.com/maps/api/js`. Next, the developer must include a div element with a specific `id` in the integrating page (highlighted in green in Figure 3) and pass that `id` to the GMaps API. The GMaps code will then be able use that div element to load its maps and interact with the user. Finally, the API can be used to navigate the map, place markers, draw polygons, ... etc. An example of such glue code is illustrated in the blue rectangle of Figure 3)

The glue code contained in the integrating webpage is used to initialize the GMaps API and to tie an event handler to an input field in a form. The form with the input field is where the user can enter a query. On submission of this form, the event handler is called. The event handler launches a geocoding query to translate the user input into GPS coordinates. These coordinates are then fed into the GMaps API and a marker is placed on the map.

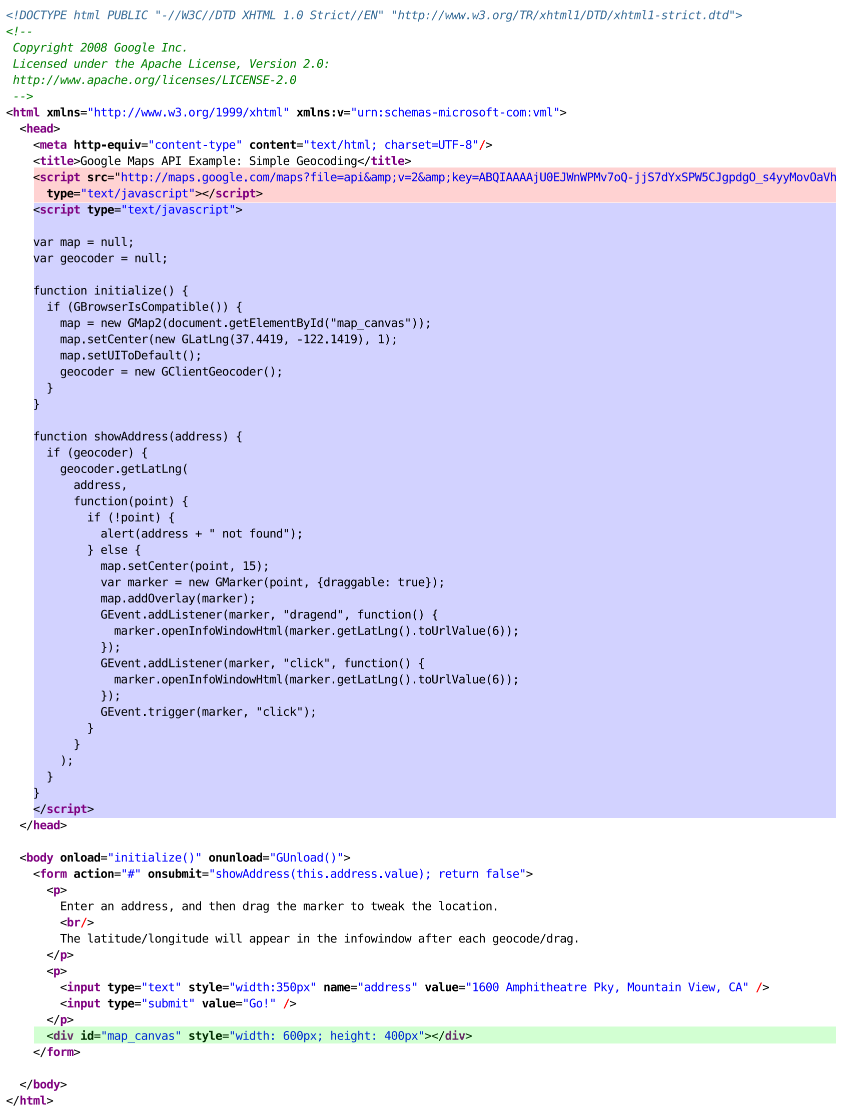
The combination of GMaps service API and client-side code can either run directly in the (outermost) document (i.e. mashup integrator document), or can be combined in a separate iframe. A important challenge when choosing for the ‘script inclusion’ composition type, is that code loaded from the Google servers will be executed in the same JavaScript context as the integrating page. While Google can (arguably) be trusted to behave, parties may become less trusted over time, or APIs may get compromised at some point in time. Moreover, this composition scenario is just particular instance of a widely class of script inclusion scenarios.

3.1.3 Requirements

This mashup requires the ability to contact the remote GMaps and geocoding APIs, it must be able to write to the div element provided by the developer and be able to interact with the user through the form input field and mouse events.

More concrete, this mashup consists of gluecode written by the integrator and the service API of GMaps. The gluecode must be able to execute an eventhandler when input from the textfield is entered. It must then be able to launch an XMLHttpRequest towards the Geocoding API and read back the response, which is a set of GPS coordinates. When received, the GPS coordinates must be handed over to the GMaps component, which either requires access to the GMaps component’s DOM to call a function, or, if using GMaps in an iframe, requires sending an inter-frame message using e.g. postMessage.

The GMaps service API requires the ability to communicate with Google

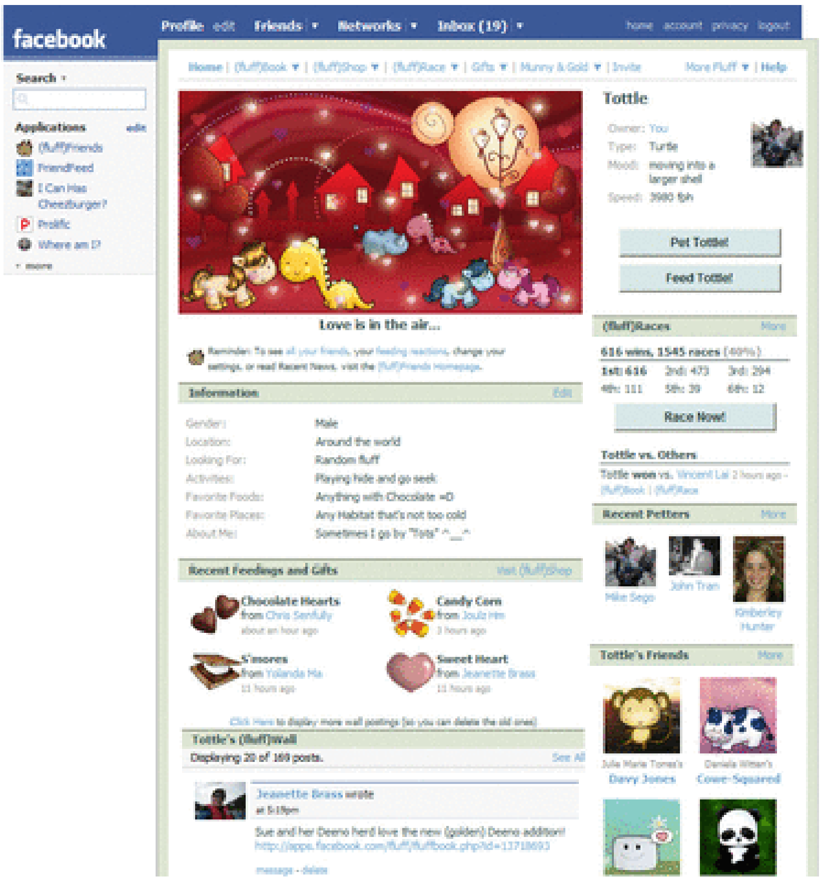


```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!--
 Copyright 2008 Google Inc.
 Licensed under the Apache License, Version 2.0:
 http://www.apache.org/licenses/LICENSE-2.0
 -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:v="urn:schemas-microsoft-com:vml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
    <title>Google Maps API Example: Simple Geocoding</title>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=ABQIAAAAjU0EJWnWPMv7oQ-jjS7dYxSPW5CJgpdgO_s4yyMovOaVh
      type="text/javascript"></script>
    <script type="text/javascript">

    var map = null;
    var geocoder = null;

    function initialize() {
      if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map_canvas"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 1);
        map.setUIToDefault();
        geocoder = new GClientGeocoder();
      }
    }

    function showAddress(address) {
      if (geocoder) {
        geocoder.getLatLng(
          address,
          function(point) {
            if (!point) {
              alert(address + " not found");
            } else {
              map.setCenter(point, 15);
              var marker = new GMarker(point, {draggable: true});
              map.addOverlay(marker);
              GEvent.addListener(marker, "dragend", function() {
                marker.openInfoWindowHtml(marker.getLatLng().toUrlValue(6));
              });
              GEvent.addListener(marker, "click", function() {
                marker.openInfoWindowHtml(marker.getLatLng().toUrlValue(6));
              });
              GEvent.trigger(marker, "click");
            }
          }
        );
      }
    }
    </script>
  </head>

  <body onload="initialize()" onunload="GUnload()">
    <form action="#" onsubmit="showAddress(this.address.value); return false">
      <p>
        Enter an address, and then drag the marker to tweak the location.
        <br/>
        The latitude/longitude will appear in the infowindow after each geocode/drag.
      </p>
      <p>
        <input type="text" style="width:350px" name="address" value="1600 Amphitheatre Pky, Mountain View, CA" />
        <input type="submit" value="Go!" />
      </p>
      <div id="map_canvas" style="width: 600px; height: 400px"></div>
    </form>

  </body>
</html>
```

Figure 3: Annotated sourcecode of the Google Maps example: API script inclusion in the red rectangle, gluecode in the blue triangle, the div element in the green triangle. This sourcecode can be found on http://gmaps-samples.googlecode.com/svn/trunk/geocoder/singlegeocode.html

Maps back-end servers to download maptiles and write access to its DIV to render the maptiles. It also requires the ability to receive mouse- and keyboard events from that DIV. Finally, if rendered in an iframe, the component should be able to use inter-frame communication to receive GPS coordinates from the gluecode.

3.2 Facebook application



Tottle

Owner: You
Type: Turtle
Mood: moving into a larger shell
Speed: 3980 fph

Love is in the air...

Reminder: To see all your friends, your feeding reactions, change your settings, or read Recent News, visit the (fluff)Friends Homepage.

Information

Gender:	Male
Location:	Around the world
Looking For:	Random fluff
Activities:	Playing hide and go seek
Favorite Foods:	Anything with Chocolate =O
Favorite Places:	Any Habitat that's not too cold
About Me:	Sometimes I go by "Tots" ^__^

(fluff)Races

616 wins, 1545 races (40%)

1st: 616	2nd: 473	3rd: 294
4th: 111	5th: 39	6th: 12

Tottle vs. Others

Tottle won vs. Vincent Lai 2 hours ago - (fluff)Book | (fluff)Race

Recent Petters

Mike Sego
John Tran
Kimberley Hunter

Recent Feedings and Gifts

Chocolate Hearts from Chris Senfully about an hour ago
Candy Corn from Joulz Hm 3 hours ago
S'mores from Yolanda Ma 11 hours ago
Sweet Heart from Jeanette Brass 11 hours ago

Click Here to display more wall postings (so you can delete the old ones)

Tottle's (fluff)Wall

Displaying 20 of 169 posts.

Jeanette Brass wrote at 5:19pm

Sue and her Deeno herd love the new (golden) Deeno addition! http://apps.facebook.com/fluff/fluffbook.php?id=13718693

Tottle's Friends

Julie Marie Torres's Davy Jones
Daniela Witten's Cowe-Squared

Figure 4: Example of a Facebook application

3.2.1 General description

Facebook is an online social networking site with more than 750 million active users as of July 2011. Third-party application developers can create applications and integrate them with Facebook to make use of Facebook's userbase. Facebook applications can use Facebook's Graph API to retrieve information about Facebook users, like personal information, friend list, photo's and messages, ...

In the past, there were two ways to create Facebook applications: integrating it in an iframe, or writing it in FBML (Facebook markup language). Applications created in FBML were displayed to the end-user as if they were hosted on the Facebook servers. To avoid security problems, these applications were not allowed to use regular JavaScript, but rather use a restricted subset of JavaScript called FBJS (Facebook JavaScript). Recently, Facebook has deprecated this approach and no longer allows new applications in FBML. All new Facebook applications must be hosted externally and integrated into Facebook with iframes.

The scenario described here is that of a generic Facebook application making use of the Graph API, but we will consider 2 toy examples so that discussion later on is not too abstract: an application using geolocation, and another using audiovisual media.

3.2.2 Technical description

A Facebook application is hosted on a third-party server and rendered in a frame on a Facebook page. The application loads pieces of JavaScript code containing Facebook's API code, using script tags. These APIs allow the application to contact Facebook and request information on the visiting Facebook user.

To make use of a user's profile information, a Facebook application must be authorized. Authorization of Facebook applications is based on OAuth 2.0 and comes in 2 flows: server-side or *authentication code flow*, and client-side or *implicit flow*. The former is used when access to the Graph API is needed from the server-side (e.g. using PHP) and the latter when such access is needed from the client-side (e.g. using JavaScript).

Server-side flow The server-side flow, depicted in Figure 5, works like this: When a Facebook users visits an application (YourApp) integrated in Facebook, YourApp is displayed in an iframe on facebook.com. The first GET request to YourApp in the iframe will trigger a redirect from YourApp to the Facebook OAuth login page. The Facebook URL that YourApp redirects to, contains *client_id*, the application identifier and *redirect_uri*, the YourApp URL that Facebook should redirect to once authentication is complete.

```
https://www.facebook.com/dialog/oauth?client_id=YOUR_APP_ID
    &redirect_uri=YOUR_URL
```

On the Facebook login page, 2 steps are performed. First, the user is authenticated in case he is not logged in to Facebook. Second, Facebook displays an authorization page listing the permissions that YourApp requires, and asks the user for approval. After this authentication and authorization step, Facebook generates an authorization code and appends it as the *code* parameter to `YOUR_URL`, the URL previously received in the *redirect_uri* parameter.

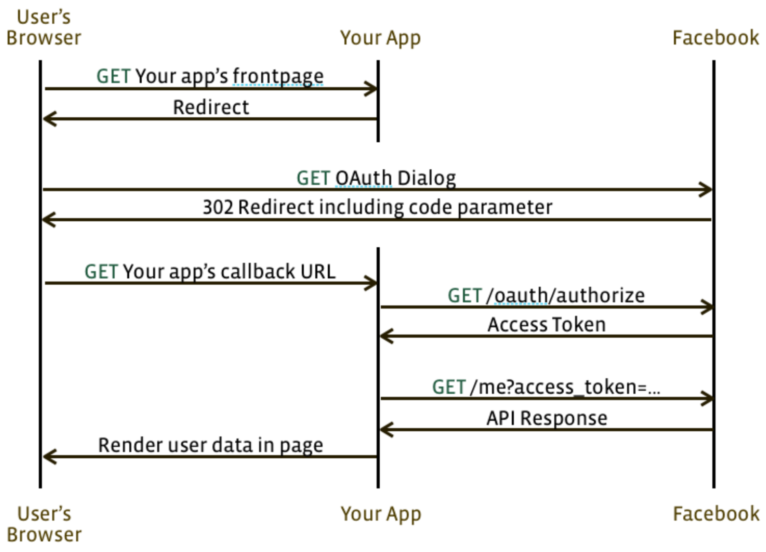


Figure 5: Diagram of the HTTP calls made during the serverside flow of the Facebook authentication process (Source: https://developers.facebook.com/docs/authentication/)

```
http://YOUR_URL?code=A_CODE_GENERATED_BY_SERVER
```

At this point, the server-side YourApp application can use the authorization code in combination with its *app secret* to request an *access token* from the Facebook servers. Using this access token, YourApp can access the user's profile using the Graph API. This access token will expire after a specified amount of time, after which YourApp can request a new access token. To receive an access token with an infinite expire time, the *offline_access* permissions must be granted to YourApp.

Client-side flow The client-side flow, depicted in Figure 6, is similar to the server-side flow. The user is redirected to the same login page, but an extra parameter *response_type* is set to *token* to identify the type of response that is expected:

```
https://www.facebook.com/dialog/oauth?client_id=YOUR_APP_ID
    &redirect_uri=YOUR_URL
    &response_type=token
```

After authentication and authorization, the user is also redirected towards `YOUR_URL`, but the *code* parameter containing the authorization code is not appended. Instead, the access token is appended as a fragment identifier:

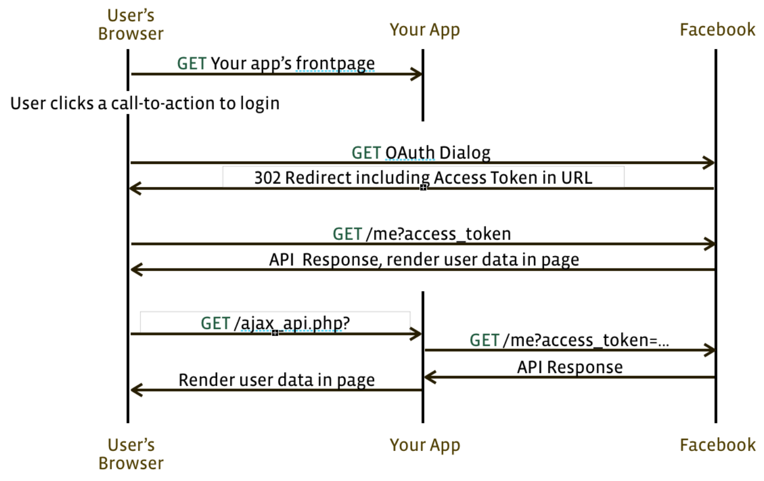


Figure 6: Diagram of the HTTP calls made during the clientside flow of the Facebook authentication process (Source: https://developers.facebook.com/docs/authentication/)

```
http://YOUR_URL#access_token=166942940015970%7C2.sa0
    &expires_in=64090}
```

Because access token is passed in a fragment identifier, only client-side code can access it. The access token can be used from the client-side directly, or it can be passed to the server-side.

3.2.3 Requirements

This scenario has 3 security stakeholders: Facebook, the third-party application provider and the end-user. The application is rendered in an iframe, so most security problems are avoided because of the same-origin policy. Besides requiring access to its own DOM, the application also requires the ability to receive keyboard and mouse events, and should be allowed to communicate out-of-band with Facebook APIs.

3.3 Online advertising

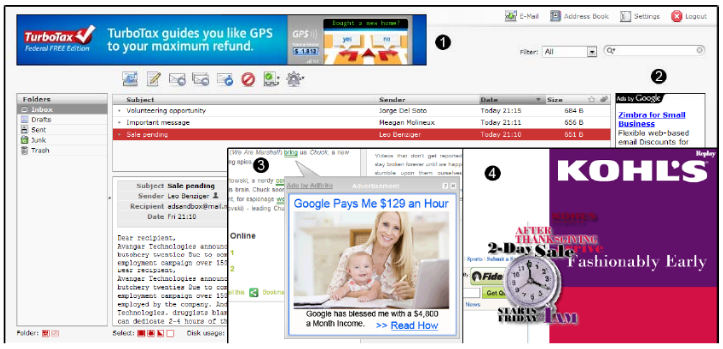

TurboTax
TurboTax guides you like GPS to your maximum refund.
Ads by Google
Zimbra for Small Business
Flexible web-based email Discounts for
Google Pays Me $129 an Hour
Google has blessed me with a $4,800 a Month Income.
>> Read How
KOHL'S
2-Day Sale
Fashionably Early
Folders
Inbox
Drafts
Sent
Junk
Trash

Figure 7: Sample of various ad types. A webmail application with (1) banner and (2) skyscraper ads. Also illustrated are (3) an inline text and (4) a floating ad. (Example taken from AdJail [18])

3.3.1 General description

Online advertising is a multi-billion dollar per year business, where organizations (advertisers) attempt to attract more customers to their business by displaying advertisements (ads) on popular third-party websites (advertisement publishers). Typically, this is achieved through an advertisement network, which provides the technology to display the ads on publisher webpages and also handles the buying and selling of ad space and ads. Examples of such advertisement networks are Google Adwords, Adbrite, DoubleClick, Zedo, ... etc. Over the years, a lot of ad types have been invented: banners (text, graphical and rich media), pop-ups and pop-unders, overlays, skinning, page pushers, ... etc. Through the use of JavaScript, these ads can be made aware of the webpage that incorporates them and alter their behavior based on the content of this page. This contextual advertising technique is very interesting to advertisers because it helps them to better target their customer groups.

The most popular online ad formats according to the 2010 report from IAB [8] are 'Search' (Listing an advertiser's company in search results) and 'Display/Banner' (The typical ad banners). The ad considered in this scenario is a typical contextual ad that displays a graphical or rich media banner on a webpage.

3.3.2 Technical description

The ad in this scenario is integrated on the website by including a piece of third-party JavaScript. This script analyzes the contents of the webpage for keywords in order to provide a contextual ad, The result of this analysis is transmitted to an ad network which returns a piece of JavaScript that needs to be written to the webpage and executed. This last piece of JavaScript contains the actual ad banner. It could be a graphical banner, an animated image or a flash application in an iframe.

3.3.3 Requirements

There are 3 security stakeholders in this mashup: the ad network, the ad publisher and the end-user. The integrating webpage must allow read- and write-access to parts of its DOM to the ad. In particular, the ad needs write access to write and update the part of the DOM where the ad is integrated, and the ad script may need to retrieve context information to select the most appropriate ad (e.g. access to meta-tags and keywords or even to the full webpage). If the ad is executed and rendered in an iframe, this implies inter-frame communication between the webpage and the ad. The ad must have the ability to communicate with its ad network, and be able to react to keyboard- and mouse events from the user.

3.4 Interactive Avatar

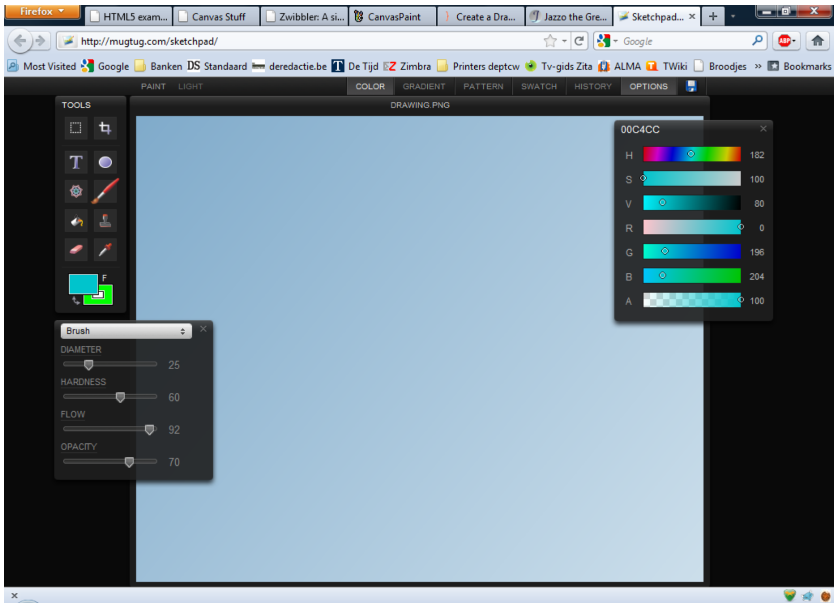


Figure 8: Example of an interactive avatar drawing tool component

3.4.1 General description

Most websites these days offer the visitor the ability to register an account and interact with other visitors of that website. Good examples of such sites are forums and blogs, where a registered user is often allowed to post an entry or a comment. The user is allowed to edit information about himself into his account details. Besides textual information like location, age, name, contact information and other, it is usually also possible to upload or select an avatar image. This avatar image is displayed next to any post the user makes.

In this scenario, the registered user is allowed to create a more interactive avatar instead of just a plain image. The avatar can access data about the user it belongs to and display it. E.g., the avatar can show the current state of the user is (logged in, editing a post, away, . . .), and render this information in an interactive way.

Avatars can be created by the user himself using JavaScript or a subset of JavaScript. More realistically, a community of avatar-builders could exist that shares pre-made interactive avatars with other users. In such a case, a user could select a pre-made avatar from a list instead of creating one from scratch. These libraries of avatars can be hosted on the same server where the user account is registered and used, but it could also be hosted elsewhere.

3.4.2 Technical description

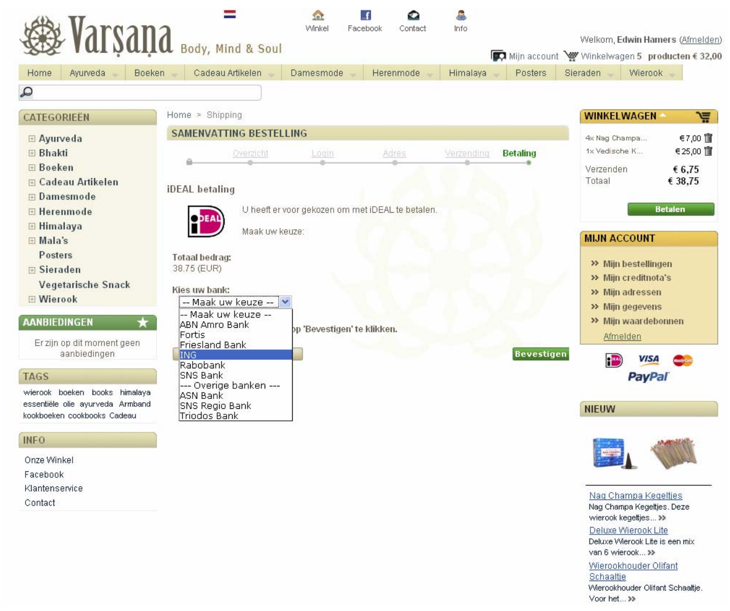
The avatar can be rendered on an HTML5 canvas element, allowing for realtime animation and full interaction with visitors. Data about the user can be retrieved in the background using a local API of the blog/forum or by using XMLHttpRequests. Avatar code could be included using script tags, like any other JavaScript code, or loaded inside an iframe.

3.4.3 Requirements

In this scenario, there are 3 security stakeholders: the blog or forum where the user profile exists, the provider where the avatar code is hosted and the end-user. The avatar needs to be able to interact with the end-user through mouse- and keyboard events. It must also be able to write to a specified location in the DOM to render an image using canvas graphics. In addition, it should be able to make use of an API on the blog/forum to request information about the user to which the avatar belongs.

In case the avatar code is hosted on the same site as the blog or forum post in which it is rendered, care should be taken that the script can not access sensitive data like cookies, or perform unwanted operations like posting additional comments.

3.5 Third-Party Payment



Er zijn op dit moment geen aanbiedingen

U heeft er voor gekozen om met iDEAL te betalen.

Maak uw keuze:

Totaal bedrag: 38.75 (EUR)

op 'Bevestigen' te klikken.

4x Nag Champa...	€7,00
1x Vedische K...	€25,00
Verzenden	€ 6,75
Totaal	€ 38,75

Nag Champa Kegeltjes
Nag Champa Kegeltjes. Deze wierook kegeltjes... »

Deluxe Wierook Lite
Deluxe Wierook Lite is een mix van 6 wierook... »

Wierookhouder Olifant Schaaltje
Wierookhouder Olifant Schaaltje. Voor het... »

Figure 9: Composition with Third-Party Payment provider

3.5.1 General description

A mashup can choose to integrate a third-party payment component, which is able to handle all payment transactions within the mashup. This relieves the mashup developr and integrator from the burden of implementing multiple payment systems (e.g. online accounts, debit or credit cards, ...), as well as potential regulations associated with those payment systems.

A common way to use a third-party payment provider is by completely delegating control of the browsing session to the third-party payment provider. After having completed the transaction, the payment provider will transfer the control back to the originating site, which can then further handle the completion of the transaction. In a mashup scenario, the third-party payment provider is integrated as a component of the originating site. Instead of transferring control of the entire session, the control of only one part of the mashup will be delegated to the payment provider. This ensures the continuous operation of the remainder of the mashup application.

Once the control is delegated to the payment provider, the payment provider is responsible for executing the payment. How exactly this is implemented is not important for the originating site. One example of a third-party payment service is PayPal, where a user has a membership account. Once the user is authenticated, the user is able to approve the payment. Another example of a third-party payment service is a front-end for different kinds of online banking applications. A user approves a payment by generating a response to a given challenge using a debit or credit card.

3.5.2 Technical description

The integration of the third-party payment component can be achieved by embedding an iframe. Within this iframe, the originating site can delegate control to the third-party payment provider. Outside of this iframe, the mashup continues to operate as before. If supported, the mashup and payment provider can even communicate (using inter-window communication techniques) while the control within the iframe has been transferred to the payment service.

Delegating control back and forth is achieved using cross-origin communication. Since this communication takes place within an iframe, it is part of the normal flow within a browsing context (i.e. loading documents). During control delegation, already existing sessions are maintained using traditional session management techniques, such as cookies.

The workflow of delegating control to a third-party payment provider and then back to the originating site are shown in Figure 12.

3.5.3 Requirements

In the third-party payment provider scenario, there are three stakeholders: the site integrating the payment service, the payment provider itself and the end-user. Each of these stakeholders share a different set of security requirements.

A first important security requirement is the secure integration of the payment service into the mashup. This secure integration relies both on isolation and the principle of least privilege. The payment provider needs to be isolated from the rest of the mashup, but delegating control back and forth needs to remain possible.

A second security requirement is to protect the payment provider (and thus the end-user) from unwanted cross-origin traffic. Cross-origin traffic related to delegating control from the originating site to the payment provider is considered legitimate, but other traffic is considered to be potentially harmful. Similarly, the originating site (and thus the end user) needs to be protected against unwanted cross-origin traffic as well. Again, transferring control back from the payment provider to the originating site is considered

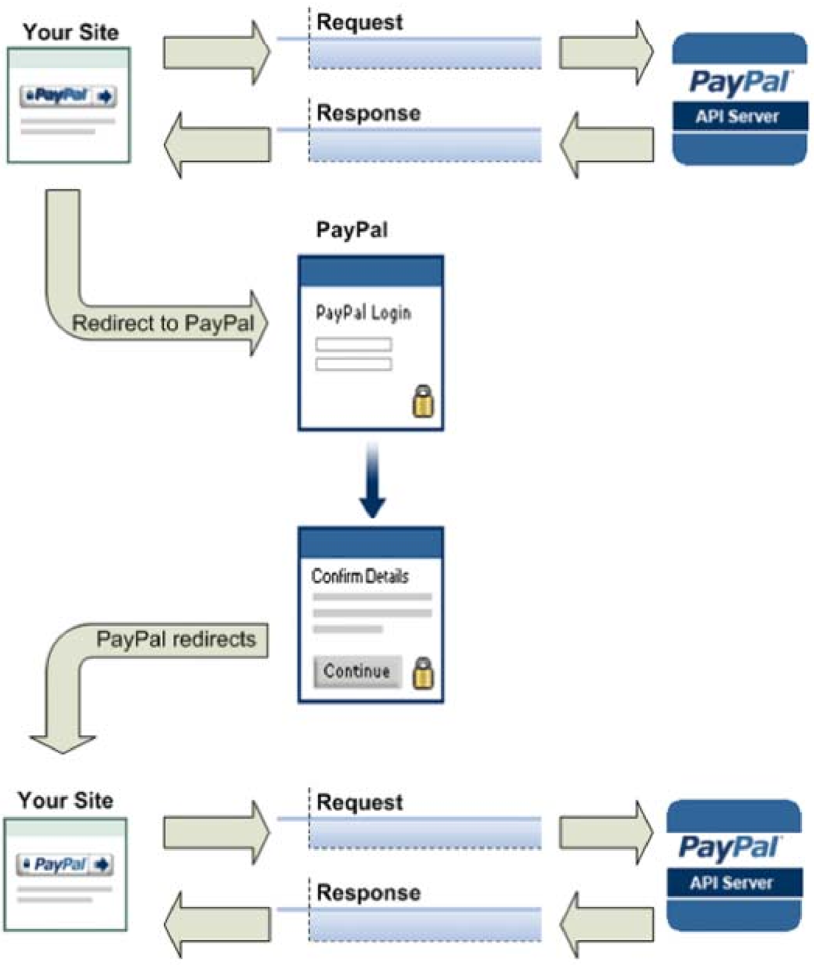
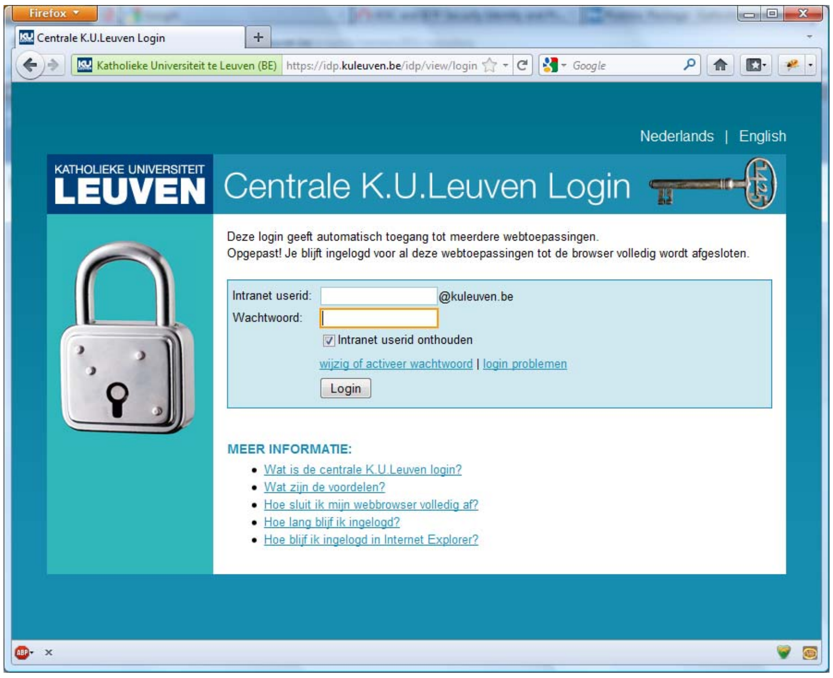


Figure 10: Third-Party Payment Workflow Overview (PayPal)

legitimate, but other traffic is not. Note that this includes traffic from the payment provider in cases where the originating site has no pending delegations for which control can be transferred back.

3.6 Third-Party Authentication


Firefox
Centrale K.U.Leuven Login
Katholieke Universiteit te Leuven (BE)
https://idp.kuleuven.be/idp/view/login
Google

Nederlands | English

KATHOLIEKE UNIVERSITEIT LEUVEN Centrale K.U.Leuven Login

Deze login geeft automatisch toegang tot meerdere webtoepassingen.
Opgepast! Je blijft ingelogd voor al deze webtoepassingen tot de browser volledig wordt afgesloten.

Intranet userid: @kuleuven.be
Wachtwoord:
[✓] Intranet userid onthouden
wijzig of activeer wachtwoord | login problemen
Login

MEER INFORMATIE:
- Wat is de centrale K.U.Leuven login?
- Wat zijn de voordelen?
- Hoe sluit ik mijn webbrowser volledig af?
- Hoe lang blijf ik ingelogd?
- Hoe blijf ik ingelogd in Internet Explorer?

Figure 11: Third-Party Authentication

3.6.1 General description

Using a third-party authentication provider is an easy way to integrate an authentication mechanism, and greatly increases the usability of the application (e.g. no new username/password combination for users to remember). A concrete example of third-party authentication providers are OpenID implementers.

A common way to use a third-party authentication provider is by delegating control of the browsing session to the provider. The provider will allow the user to authenticate in any of the supported ways (e.g. username/-password, token, electronic ID, etc.). After having authenticated the user, the authentication provider will transfer the control back to the originating site. The site can now continue loading the user profile, either with local data or data obtained from the authentication provider. In a mashup scenario, the third-party authentication provider is integrated as a component of the originating site. Instead of transferring control of the entire session, the control of only one part of the mashup will be delegated to the authentication provider. This ensures the continuous operation of the remainder of the mashup application.

3.6.2 Technical description

The integration of the third-party authentication component can be achieved by embedding an iframe. Within this iframe, the originating site can delegate control to the authentication provider. Outside of this iframe, the mashup continues to operate as before. If supported, the mashup and authentication provider can even communicate (using inter-window communication techniques) while the control within the iframe has been transferred to the authentication service. After authentication, the other mashup components can start loading user-specific content or services.

Delegating control back and forth is achieved using cross-origin communication. Since this communication takes place within an iframe, it is part of the normal flow within a browsing context (i.e. loading documents). During control delegation, already existing sessions are maintained using traditional session management techniques, such as cookies.

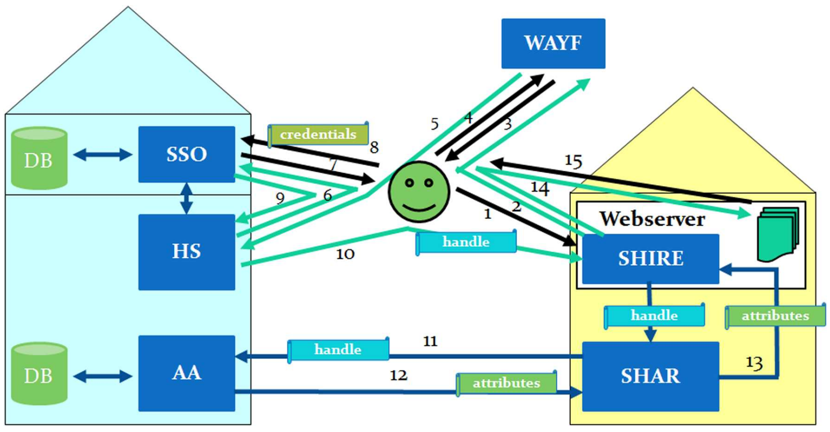


Figure 12: Third-Party Authentication Workflow Overview (Shibboleth)

3.6.3 Requirements

In the third-party authentication provider scenario, there are three stakeholders: the site integrating the authentication service, the authentication provider itself and the end-user. Each of these stakeholders share a different set of security requirements.

A first important security requirement is the secure integration of the authentication component into the mashup. This secure integration relies both on isolation and the principle of least privilege. The authentication provider obviously needs to be isolated from the rest of the mashup, but delegating control back and forth needs to remain possible.

A second security requirement is to protect the authentication provider (and thus the end-user) from unwanted cross-origin traffic. Cross-origin traffic related to delegating control from the originating site to the authentication provider is considered legitimate, but other traffic is considered to be potentially harmful. Similarly, the originating site (and thus the end user) needs to be protected against unwanted cross-origin traffic as well. Again, transferring control back from the authentication provider to the originating site is considered legitimate, but other traffic is not. Note that this includes traffic from the authentication provider in cases where the originating site has no pending delegations for which control can be transferred back.

3.7 Holiday Pictures Mashup

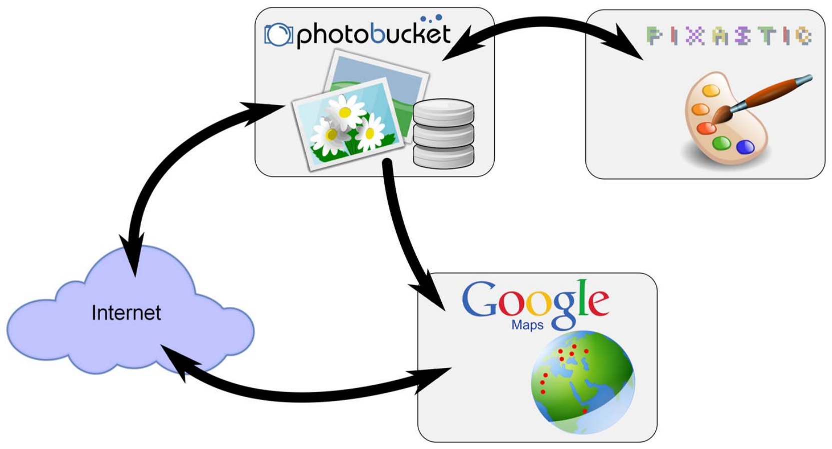


Figure 13: Holiday pictures mashup example

3.7.1 General description

This scenario combines a few of the previous scenarios into a bigger mashup.

Consider a holiday pictures mashup consisting of three third-party components integrated with some glue code as depicted in Figure 13: a gallery component that stores the pictures, a photo editor component that can be used to edit the pictures and a map component used to draw geotagging information on a world map.

In the gallery component, a user can view, upload and organize pictures. Pictures in the gallery component are stored on a remote server and accessed through an API. But it is also possible to store copies of pictures locally in the browser, so that the user may preview or edit them first. Once the user is ready to publish the pictures, they are transferred to the remote server.

The photo editor allows a user to make changes to a photo. These changes could e.g. be resizing or cropping, adjusting white-balance, applying image filters, adding a caption, altering some meta-data, ... etc. Pictures are loaded into the photo editor from local storage only, so that the editor can be used even when there is no access to the internet. Likewise, edited photos are saved back to local storage.

The map component uses geotagging information to organize photos geographically on a world map. Photos store meta-data about the conditions under which they were taken: lighting conditions, aperture size, the type of camera, ... and also GPS coordinates of the location where the photo was taken. The map component uses the geotagging information of selected photos to draw markers or thumbnails on a map. When photos are displayed on a map in this fashion, a user can easily see which photos belong together and group them e.g. by holiday trip.

3.7.2 Technical description

The gallery component stores and retrieves photos using an online photo gallery application like PhotoBucket [12]. To this end it must be able to contact the PhotoBucket API through e.g. XMLHttpRequest [36], read and write photos from and to disk through the File API[22], and make use of local storage [34] to store a local data cache.

The photo editor component, similar to Pixastic [14], can be used to edit photos within the gallery. This component does not need to communicate with any remote API. Its functionality can be implemented completely in JavaScript by making use of an HTML 5 canvas element. Photos are loaded into the editor from local storage only.

The map component displays images in the gallery on a world map e.g. using Google Maps [5] and positions them based on their geotagging information. Communication with the Google Maps API is essential for this component. Optionally, this component can also retrieve the current location of the user through Geolocation, and position a marker on the map to indicate where the user is in relationship to the other markers.

Components communicate with eachother via the glue code through an inter-frame communication mechanism (e.g. postMessage [6]) in case of iframe-integration, or via direct function calls or data manipulation in case of script inclusion [3]. The gallery and photo editor components communicate to indicate which photos are to be edited or have been altered. The gallery component communicates to the map component to indicate where a marker should be placed on the map and what it should look like. There is no communication between the photo editor and the map component.

In addition to the inter-frame communication via the gluecode, the gallery and photo editor components share a local storage database to exchange large images. To edit an image in the gallery component, it must be saved to local storage and then accessed in the photo editor component. Once edited, the image is stored in local storage where it is then available to both components.

3.7.3 Requirements

In this larger scenario, there are 5 security stakeholders: the provider hosting the mashup's gluecode, the 3 component providers and the end-user. All

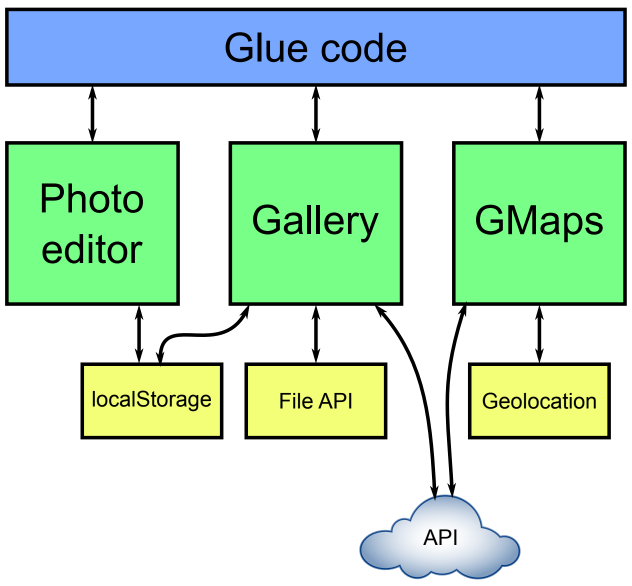


Figure 14: Holiday pictures mashup: interactions between gluecode, components and browser functionality

components are integrated via iframes and so the same-origin policy takes care of a lot of security problems.

As shown in Figure 14, none of the 3 components requires the ability to directly contact any other component, but they must all be able to communicate with the glue code. All components must be able to interact with the user through keyboard- and mouse-events and have access to their DOM. The gallery component must be able to contact its remote API and make use of local storage to store and retrieve holiday pictures. The photo editor component does not need to contact any external API, since it is completely implemented as a clientside JavaScript application, but it must also be able to access local storage. Finally, like in Subsection 3.1, the map component also requires the ability to communicate with its remote API. In addition the map component should be able to access Geolocation information.

4 Secure composition policies

In this section, two secure composition policies will be defined to secure mashup compositions.

The first policy, the *least-privilege composition policy*, expresses the execution constraints to securely execute third-party components, i.e. the policy specifies the minimum set of privileges allowed by the component. This maps back to the security-sensitive operations (discussed in Section 2), and includes both behavioral as well as communication constraints.

To cope with more advanced communication constraints between cooperative mashup components (as is the case in the third-party authentication and third-party payment scenario), the *trusted delegation policy* will be defined. This second policy captures an often recurring communication pattern between cooperative components, which can only be coarse-grained guarded with the least-privilege composition policy. With the trusted delegation policy, a one-to-one mapping can be achieved between cooperative components giving back-and-forward control to each other.

Finally, this section briefly considers some first steps in studying composition of information flow properties – connecting WP3 and 4.

4.1 Least-privilege composition policy

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.

Instead, a new client-side security architecture is needed that enables least-privilege integration of components into a web mashup, based on high-level policies that restrict the available functionality in each individual component. Such a policy language can be synthesized from our study and categorization of sensitive operations in the upcoming HTML 5 JavaScript APIs (See Section 2).

4.1.1 Least-privilege composition

Taking into account the attack vectors present in current mashup compositions, 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, 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 do not exploit the insecure composition attack vectors and multiply their impact by using the security sensitive HTML5 APIs described in Section 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. or at least execute in the client-side environment.

4.1.2 Composition 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 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.

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

The full grammar of the policy file in EBNF is shown in Figure 15.

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 a mashup component can be used inside another mashup component. In such a case, the functionality in the inner component is determined by the policies imposed on enclosing components, in addition to its own policy (if it has one). Allowing sensible cascading of policies implies that "deeper" policies can only make the total policy more strict. If this were not the case, a component with a less strict policy could be used to "break out" of another component's 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 $p \subseteq b$.

```
POLICY                  =  "{", {(
                           DOMACCESSREADRULE  | DOMACCESSWRITERULE  |
                               COOKIESREADRULE |
                           COOKIESWRITERULE   | EXTCOMMRULE         |
                               INTCOMMRULE     |
                           STORAGEREADRULE    | STORAGEWRITERULE    | UIRULE
                                          |
                           MEDIARULE          | GEOLOCATIONRULE     | DEVICERULE
                           ), ","}, "}";

(* DOM Access *)
DOMACCESSREADRULE       =  ''"domaccess-read"'',   IS, (BOOLEAN | KEYSET);
DOMACCESSWRITERULE      =  ''"domaccess-write"'',  IS, (BOOLEAN | KEYSET);

(* Cookies *)
COOKIESREADRULE         =  ''"cookies-read"'',     IS, (BOOLEAN | KEYSET);
COOKIESWRITERULE        =  ''"cookies-write"'',    IS, (BOOLEAN | KEYSET);

(* External communication *)
EXTCOMMRULE             =  ''"extcomm"'',          IS, (BOOLEAN | DOMAINSET);

(* Interframe communication *)
INTCOMMRULE             =  ''"framecomm"'',        IS, (BOOLEAN | DOMAINSET);

(* Local storage *)
STORAGEREADRULE         =  ''"storage-read"'',     IS, (BOOLEAN | KEYSET);
STORAGEWRITERULE        =  ''"storage-write"'',    IS, (BOOLEAN | KEYSET);

(* UI and Rendering *)
UIRULE                  =  ''"ui"'',               IS, BOOLEAN;

(* Media *)
MEDIARULE               =  ''"media"'',            IS, BOOLEAN;

(* Geolocation *)
GEOLOCATIONRULE         =  ''"geolocation"'',      IS, BOOLEAN;

(* Device Access *)
DEVICERULE              =  ''"device"'',           IS, KEYSET;

IS                      =  " : ";
BOOLEAN                 =  ''"yes"'' | ''"no"'';
KEYSET                  =  "[", QELEM, {",", QELEM}, "]";
DOMAINSET               =  "[", QDOMAIN, {",", QDOMAIN}, "]";

QDOMAIN                 =  '"', DOMAIN, '"';
DOMAIN                  =  ALPHA, {ALPHA | "."}, ALPHA;

QELEM                   =  '"', ELEM, '"';
ELEM                    =  ALPHA, {ALPHA};

ALPHA                   =  'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
     | 'j' |
                           'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's'
                               | 't' |
                           'u' | 'v' | 'w' | 'x' | 'y' | 'z';
```

Figure 15: EBNF notation of the policy syntax

4.2 Trusted delegation policy between cooperative components

To cope with more advanced communication constraints between cooperative mashup components (as is the case in the third-party authentication and third-party payment scenario), the *trusted delegation policy* proposes a more fine-grained policy on cross-domain communication than the *least-privilege composition policy.* By doing so, the trusted delegation policy captures an often recurring communication pattern between cooperative components, and achieve a secure one-to-one mapping between cooperative components giving back-and-forward control to each other. The policy protects stakeholders against cross-origin traffic, other than the legitimate traffic, i.e. the policy is able to make the distinction trusted cross-domain communication and potentially harmful cross-origin traffic.

4.2.1 Basic policy

The core idea of our policy is the following: client-side state (i.e. session cookie headers and authentication headers) is stripped from all cross-origin requests, except for *expected* requests. A cross-origin request from origin A to B is *expected* if B previously (earlier in the browsing session) *delegated* to A. We say that B *delegates* to A if B either issues a POST request to A, or if B redirects to A using a URI that contains parameters.

The rationale behind this core idea is that (1) non-malicious collaboration scenarios follow this pattern [4], and (2) it is hard for an attacker to trick A into delegating to a site of the attacker: forcing A to do a POST or parametrized redirect to an evil site E requires the attacker to either identify a cross-site scripting (XSS) vulnerability in A, or to break into A's webserver. In both these cases, A has more serious problems than a cross-origin attack.

Obviously, a GET request from A to B is not considered a delegation, as it is very common for sites to issue GET requests to other sites, and as it is easy for an attacker to trick A into issuing such a GET request (see for instance attack scenario A2 in [4]).

4.2.2 Redirects

Unfortunately, the elaboration of this simple core idea is complicated somewhat by the existence of HTTP redirects. A web server can respond to a request with a *redirect* response, indicating to the browser that it should resend the request elsewhere, for instance because the requested resource was moved. The browser will follow the redirect automatically, without user intervention. Redirects are used widely and for a variety of purposes, so we cannot ignore them. For instance, both non-malicious scenarios in [4] heavily depend on the use of redirects. In addition, attacker-controlled websites can also use redirects in an attempt to bypass client-side protection. Akhawe et al. [1] discuss several examples of how attackers can use redirects to attack web applications, including an attack against a CSRF countermeasure. Hence, correctly dealing with redirects is a key requirement for security.

4.2.3 Filtering algorithm

The flowgraph in Figure 16 summarizes our filtering algorithm. For a given request, it determines what session state (cookies and authentication headers) the browser should attach to the request. The algorithm differentiates between simple requests and requests that are the result of a redirect.

Simple Requests Simple requests that are not cross-origin, as well as expected cross-origin requests are handled as unprotected browsers handle them today. The browser automatically attaches the last known client-side state associated with the destination origin (point 1). The browser does not attach any state to non-expected cross-origin requests (point 3).

Redirect Requests If a request is the consequence of a redirect response, then the algorithm determines if the redirect points to the origin where the response came from. If this is the case, the client-side state for the new request is limited to the client-side state known to the previous request (i.e. the request that triggered this redirect) (point 2). If the redirect points to another origin, then, depending on whether this cross-origin request is expected or not, it either gets session-state automatically attached (point 1) or not (point 3).

When Is a Request Expected? A key element of the algorithm is determining whether a request is *expected* or not. As discussed above, the intuition is: a cross-origin request from B to A is expected if and only if A first delegated to B by issuing a POST request to B, or by a parametrized redirect to B. Our algorithm stores such trusted delegations, and an assumption that we rely on (and that we refer to as the *trusted-delegation assumption*) is that sites will only perform such delegations to sites that they trust. In other words, a site A remains vulnerable to attacks from origins to which it delegates. We provide experimental evidence for this assumption in a moment.

The algorithm to decide whether a request is expected goes as follows.

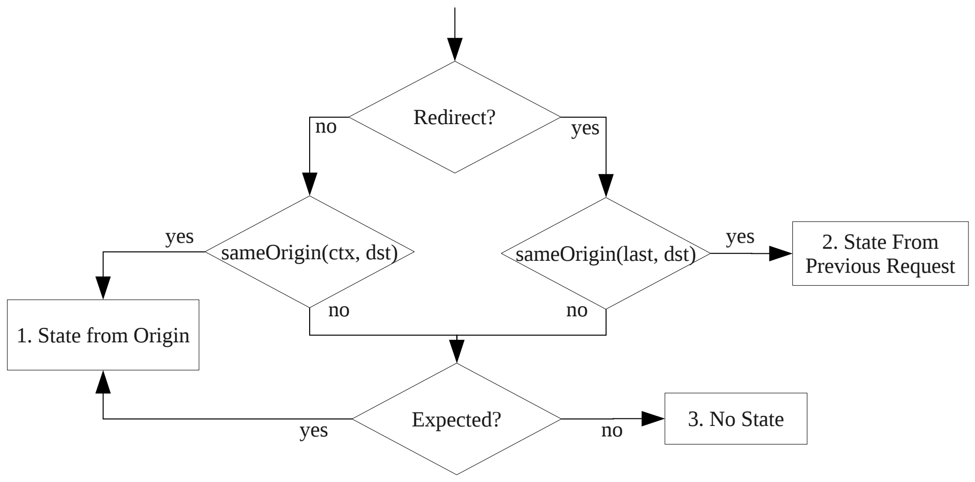


Figure 16: The request filtering algorithm

For a simple cross-origin request from site B to site A, a trusted delegation from site A to B needs to be present in the delegation store.

For a redirect request that redirects a request to origin Y (light gray) to another origin Z (dark gray) in a browsing context associated with some origin α, the following rules apply.

1. First, if the destination (Z) equals the source (i.e. $\alpha = Z$) (Figure 17a), then the request is expected if there is a trusted delegation from Z to Y in the delegation store. Indeed, Y is effectively doing a cross-origin request to Z by redirecting to Z. Since the browsing context has the same origin as the destination, it can be expected not to manipulate redirect requests to misrepresent source origins of redirects (cfr. next case).

2. Alternatively, if the destination (Z) is not equal to the source (i.e. $\alpha \neq Z$) (Figure 17b), then the request is expected if there is a trusted delegation from Z to Y in the delegation store, since Y is effectively doing a cross-origin request to Z. Now, the browsing context might misrepresent source origins of redirects by including additional redirect hops (origin X (white) in Figure 17c). Hence, our decision to classify the request does not involve X.

Finally, our algorithm imposes that expected cross-origin requests can only use the GET method and that only two origins can be involved in the request chain. These restrictions limit the potential power an attacker might have, even if the attacker successfully deceives the trusted-delegation mechanism.

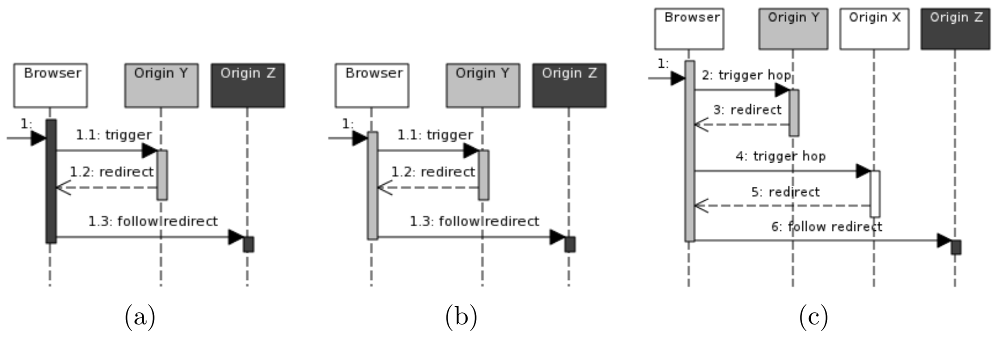


Figure 17: Complex cross-origin redirect scenarios

	# requests		POST	redir.
Third party service mashups	29.282	(52,95%)	5.321	23.961
Advertisement services	*22.343*	*(40,40%)*	*1.987*	*20.356*
Gadget provider services (appspot, mochibot, gmodules, . . .)	*2.879*	*(5,21%)*	*2.757*	*122*
Tracking services (metriweb, sitestat, uts.amazon, . . .)	*2.864*	*(5,18%)*	*411*	*2.453*
Single Sign-On services (Shibboleth, Live ID, OpenId, . . .)	*1.156*	*(2,09%)*	*137*	*1.019*
3rd party payment services (Paypal, Ogone)	*27*	*(0,05%)*	*19*	*8*
Content sharing services (addtoany, sharethis, . . .)	*13*	*(0,02%)*	*10*	*3*
Multi-origin websites	13.973	(25,27%)	198	13.775
Content aggregators	8.276	(14,97%)	0	8.276
Feeds (RSS feeds, News aggregators, mozilla fxfeeds, . . .)	*4.857*	*(8,78%)*	*0*	*4.857*
Redirecting search engines (Google, Comicranks, Ohnorobot)	*3.344*	*(6,05%)*	*0*	*3.344*
Document repositories (ACM digital library, dx.doi.org, . . .)	*75*	*(0,14%)*	*0*	*75*
False positives (wireless network access gateways)	1.215	(2,20%)	12	1.203
URL shorteners (gravatar, bit.ly, tinyurl, . . .)	759	(1,37%)	0	759
Others (unclassified)	1.795	(3,24%)	302	1.493
Total number of 3rd party delegation initiators	**55.300**	**(100%)**	**5.833**	**49.467**

Table 2: Analysis of the trusted-delegation assumption in a real-life data set of 4.729.217 HTTP requests

4.2.4 Experimental validation of the policy

The policy protects against illegitimate cross-origin traffic, but relies on the correct identification of a site delegating control. The experimental analysis below shows that only 1.17% of the requests in the data set are considered a false positive (i.e. seen as a delegation when they are not a delegation).

We conducted an extensive traffic analysis using a real-life data set of 4.729.217 HTTP requests, collected from 50 unique users over a period of 10 weeks. The analysis revealed that 1.17% of the 4.7 million requests are treated as delegations in our approach. We manually analyzed all these 55.300 requests, and classified them in the interaction categories summarized in Table 2.

For each of the categories, we discuss the resulting attack surface:

Third party service mashups. This category consists of various third party services that can be integrated in other websites. Except for the single sign-on services, this is typically done by script inclusion, after which the included script can launch a sequence of cross-origin GET and/or POST requests towards offered AJAX APIs. In addition, the service providers themselves often use cross-origin redirects for further delegation towards content delivery networks.

As a consequence, the origin A including the third-party service S becomes vulnerable to cross-origin traffic from S. This attack surface is unimportant, as in these scenarios, S can already attack A through script inclusion, which offers much more power than a cross-origin attack.

In addition, advertisement service providers P that further redirect to content delivery services D are vulnerable to cross-origin attacks from D whenever a user clicks an advertisement. Again, this attack surface is unimportant: the delegation from P to D correctly reflects a level of trust that P has in D, and P and D will typically have a legal contract or SLA in place.

Multi-origin websites. Quite a number of larger companies and organizations have websites spanning multiple origins (such as *live.com* - *microsoft.com* and *google.be* - *google.com*). Cross-origin POST requests and redirects between these origins make it possible for such origins to attack each other. For instance, *google.be* could attack *google.com*. Again, this attack surface is unimportant, as all origins of such a multi-origin website belong to a single organization.

Content aggregators. Content aggregators collect searchable content and redirect end-users towards a specific content provider. For news feeds and document repositories (such as the ACM digital library), the set of content providers is typically stable and trusted by the content aggregator, and therefore again a negligible attack vector.

Redirecting search engines register the fact that a web user is following a link, before redirecting the web user to the landing page (e.g. as Google does for logged in users). Since the entries in the search repository come from all over the web, our policy provides little protection for such search engines. Our analysis identified 4 such origins in the data set: *google.be*, *google.com*, *comicrank.com*, and *ohnorobot.com*.

False positives. Some fraction of the cross-origin requests are caused by network access gateways (e.g. on public Wifi) that intercept and reroute requests towards a payment gateway. Since such devices have man-in-the-middle capabilities, and hence more attack power than cross-origin attacks, the resulting attack surface is again negligible.

URL shorteners. To ease URL sharing, URL shorteners transform a shortened URL into a preconfigured URL via a redirect. Since such URL shortening services are open, an attacker can easily control a new redirect target. The effect is similar to the redirecting search engines; URL shorteners are essentially left unprotected by our countermeasure. Our analysis identified 6 such services in the data set: *bit.ly*, *gravatar.com*, *post.ly*, *tiny.cc*, *tinyurl.com*, and *twitpic.com*.

Others(unclassified) For some of the requests in our data set, the origins involved in the request were no longer online, or the (partially anonymized) data did not contain sufficient information to reconstruct what was happening, and we were unable to classify or further investigate these requests.

In summary, our experimental analysis shows that the trusted delegation assumption is realistic. Only 10 out of 23.592 origins (i.e. 0.0042% of the examined origins) – the redirecting search engines and the URL shorteners – perform delegations to arbitrary other origins. They are left unprotected by our countermeasure. But the overwhelming majority of origins delegates (in our precise technical sense, i.e. using cross-origin POST or redirect) only to other origins with whom they have a trust relationship.

4.2.5 Formal Modeling and Checking

The design of web security mechanisms is complex: the behavior of (same-origin and cross-origin) browser requests, server responses and redirects, cookie and session management, as well as the often implicit threat models of web security can lead to subtle security bugs in new features or countermeasures. In order to evaluate proposals for new web mechanisms more rigorously, Akhawe et al. [1] have proposed a model of the Web infrastructure, formalized in Alloy.

The base model is some 2000 lines of Alloy source code, describing (1) the essential characteristics of browsers, web servers, cookie management and the HTTP protocol, and (2) a collection of relevant threat models for the web. The Alloy Analyzer – a bounded-scope model checker – can then produce counterexamples that violate intended security properties if they exist in a specified finite scope.

To model the trusted delegation composition policy, we needed to extend the model of Akhawe et al. to include (a) the accessible client-side state at a certain point in time, (b) the trusted delegation assumption and (c) our filtering algorithm. More details on this extension can be found in [4].

We formally define a CSRF attack as the possibility for a web attacker (defined in the base model) to inject a request with at least one existing cookie attached to it (this cookie models the session/authentication information attached to requests) in a session between a user and an honest server.

We provided the Alloy Analyzer with a universe of at most 9 HTTP events and where an attacker can control up to 3 origins and servers (a similar size as used in [1]). In such a universe, no examples of an attacker injecting a request through the user's browser were found. This gives strong assurance that the countermeasure does indeed protect against CSRF, while non-malicious scenarios (such as the third-party authentication and third-party payment scenario) are indeed permitted. From this, we can also conclude that our extension of the base model to express the trusted delegation composition policy is consistent.

4.3 Secure composition policies for the seven composition scenarios

To validate what these policies look like in practice, we have created example policies for each of the scenarios described in Section 3.

4.3.1 Google Maps

The Google Maps scenario integrates only 1 component into a webpage with gluecode: the Google Maps component. The policy for this component should reflect that the component can communicate with its remote API, have access to the DOM to render images and interact with the user.

```
{
  "domaccess-read"  : [ "gmapsDIV" ],
  "domaccess-write" : [ "gmapsDIV" ],
  "ui"              : "yes",
  "extcomm"         : [ "maps.google.com" ],
}
```

4.3.2 Facebook application

Once again, only one component is integrated: the third-party application. Both toy-examples introduced in Subsection 3.2 basically share the same policy: the application should be able to access the DOM, interact with the user and make use of external communication to both the application's API and the Facebook API.

```
{
  "domaccess-read"  : "yes",
  "domaccess-write" : "yes",
  "ui"              : "yes",
  "extcomm"         : [ "myapp.com", "api.facebook.com" ],
}
```

However, the geolocation application should also be able to make use of Geolocation:

```
{
  "domaccess-read"  : "yes",
  "domaccess-write" : "yes",
  "ui"              : "yes",
  "extcomm"         : [ "myapp.com", "api.facebook.com" ],
  "geolocation"     : "yes",
}
```

and the application using audiovisual media should be allowed to do that:

```
{
  "domaccess-read"  : "yes",
  "domaccess-write" : "yes",
  "ui"              : "yes",
  "extcomm"         : [ "myapp.com", "api.facebook.com" ],
  "media"           : "yes",
}
```

4.3.3 Online Advertising

The Ads scenario also integrates only 1 component into a webpage with gluecode: the ad itself. The ad should be able to read the DOM and write to a specific part of it. It should also be able to communicate with its ad network and interact with the user.

```
{
  "domaccess-read"  : "yes",
  "domaccess-write" : [ "adDIV" ],
  "ui"              : "yes",
  "extcomm"         : [ "adnetwork.com" ],
}
```

4.3.4 Interactive Avatar

The only component being integrated in this scenario is the interactive avatar. It should be able to write to the DOM, but not read from it. Interaction with the user should be possible, and it should be able to contact the blog or forum on which the user account exists.

```
{
  "domaccess-read"  : "no",
  "domaccess-write" : [ "avatarDIV" ],
  "ui"              : "yes",
  "extcomm"         : [ "forum.com" ],
}
```

4.3.5 Third-Party Payment

The delegation from the mashup towards the payment provider and back is enabled by the trusted delegation policy. Figure 18 shows a step-by-step flow of control in which the delegation towards the payment provider takes place (step 2). The payment provider transfers control back to the mashup in step 13.

According to the policy, the delegation in step 2 is legitimate, which means that the provider is allowed to transfer control back to the mashup. Every cross-origin request in this flow (step 2 and 4), except for the expected request (step 13) is considered to be potentially harmful. This means that the

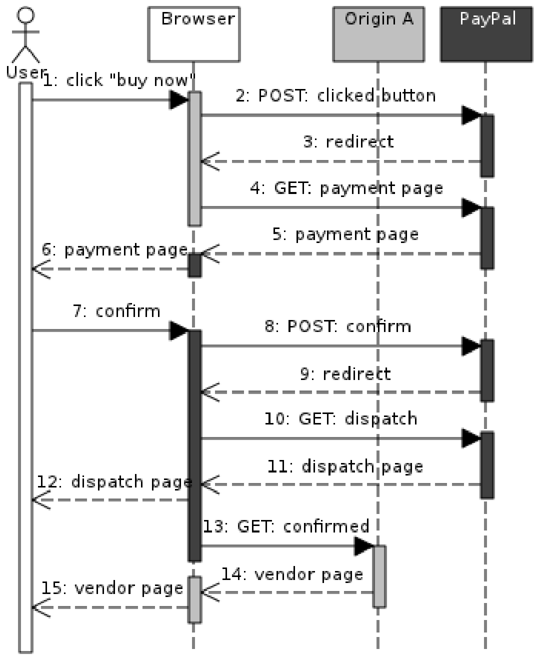


Figure 18: Step-by-step flow diagram for third-party payment

policy will render these requests harmless by removing any attached client-side state enabling session abuse. Step 13 is allowed by the policy, so when the control is transferred back to the mashup, the presence of the client-side state enables a clean delegation from the provider.

4.3.6 Third-Party Authentication

The delegation from the mashup towards the third-party authentication provider and back is enabled by the trusted delegation policy. Figure 19 shows a step-by-step flow of control in which the delegation towards the authentication provider takes place (step 4). The authentication provider transfers control back to the mashup in step 14.

According to the policy, the delegation in step 4 is legitimate, which means that the provider is allowed to transfer control back to the mashup. Step 14 is allowed by the policy, so when the control is transferred back to the mashup, the presence of the client-side state enables a clean delegation from the provider.

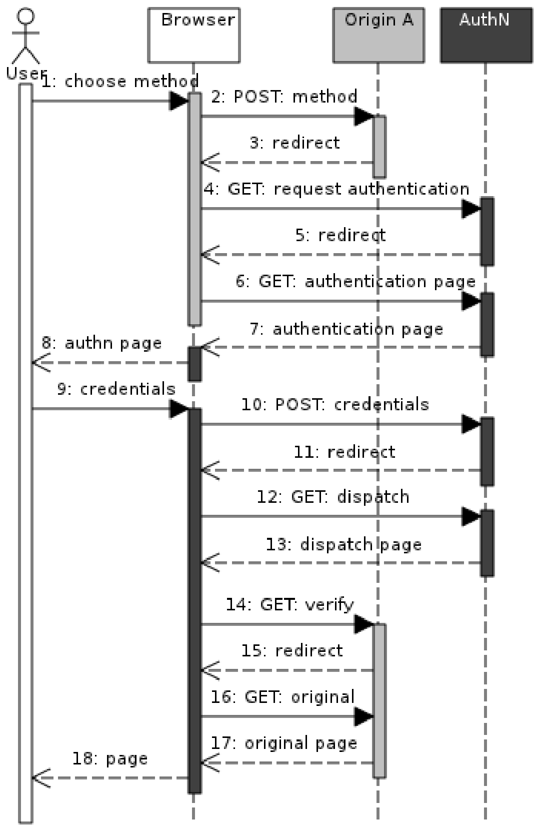


Figure 19: Step-by-step flow diagram for third-party authentication

4.3.7 Holiday Pictures Mashup

The Holiday Picture mashup consists of 4 components of which 1 is the gluecode. The other 3 each require their own policy.

All components must be able to use inter-frame communication, but only towards the gluecode. They must also be able to interact with the user and access their DOM. All except the Pixastic component must be able to contact their remote API. The PhotoBucket component requires access to client-side storage and the GMaps component needs Geolocation functionality.

PhotoBucket policy:

```
{
  "framecomm"         : [ "integrator.tld" ],
  "ui"                : "yes",
  "domaccess-read"    : "yes",
  "domaccess-write"   : "yes",
  "storage-read"      : "yes",
  "storage-write"     : "yes",
  "extcomm"           : [ "photobucket.com" ],
}
```

Pixastic policy:

```
{
  "framecomm"         : [ "integrator.tld" ],
  "ui"                : "yes",
  "domaccess-read"    : "yes",
  "domaccess-write"   : "yes",
}
```

GMaps policy:

```
{
  "framecomm"         : [ "integrator.tld" ],
  "ui"                : "yes",
  "domaccess-read"    : [ "gmapsDIV" ],
  "domaccess-write"   : [ "gmapsDIV" ],
  "geolocation"       : "yes",
  "extcomm"           : [ "maps.google.com" ],
}
```

4.4 Secure Composition for Information Flow Policies

We have also conducted some work which potentially links the present work package and WP3 on information flow. We describe this briefly here.

The idea of building secure systems by composing "secure" components is appealing, but this requires a definition of security which, in addition to taking care of top-level security goals, is strengthened appropriately in order to be compositional. For information-flow policies this is known to be difficult because when components share resources, the way they interact may influence information flows in profound ways.

Previously studies for compositional information-flow security of shared-variable concurrent programs paid a high price for compositionality: a component must be extremely pessimistic about what an environment might do with shared resources. This pessimism leads to many intuitively secure components being labelled as insecure.

Since in practice it is only meaningful to compose components which follow an agreed protocol for data access, we take advantage of this to develop a more liberal compositional security condition. The idea is to give the security definition access to the intended pattern of data usage, as expressed by assumption-guarantee style conditions associated with each component. We have illustrated the improved precision by developing the first flow-sensitive security type system that provably enforces a noninterference-like property for concurrent programs.

More details on this work are provided in the paper *Assumptions and Guarantees for Compositional Noninterference* published in CSF 2011 [10].

5 Implementation strategies

In this section, we report on the early exploration of implementation strategies to enforce secure composition policies. Various techniques have been studied ranging from browser instrumentation, over browser extensions and server-side JavaScript wrapping approaches, to a two-tier client-side architecture.

In particular, Section 5.1 proposes WebJail, a client-side security architecture that enforces the least-privilege composition policy, and discusses a possible implementation strategy via a modified browser. In Section 5.2, the trusted delegation policy is implemented as the default policy of the existing browser extension CsFire. Section 5.3 discusses the feasibility to wrap security-sensitive JavaScript operations as part of a server-side proxy, and the security considerations to take into account to achieve full mediation. In Section 5.4, a two-tier architecture for sandboxed code is proposed to combine a baseline sandbox with a stateful fine-grained policy, without browser modification or restrictions on the supported scripts.

Thanks to this early exploration, first insights were acquired in the feasibility and complexity of each of the implementation strategies. This early exploration of implementation strategies will also further drive the selection and development of enforcement techniques in the remaining tasks of WP4.

5.1 WebJail: least-privilege implementation

WebJail is a client-side security architecture that enforces the least-privilege composition policy described in Section 4.1 via the use of deep aspects in the browser. More details on WebJail are included in the WebJail paper[19], which has been accepted at ACSAC 2011.

5.1.1 Architecture

The WebJail architecture consists of three abstraction layers as shown in Figure 20. The upper layer, the *policy layer*, associates the secure composition policy with a mashup component, and triggers the underlying layers to enforce the 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.

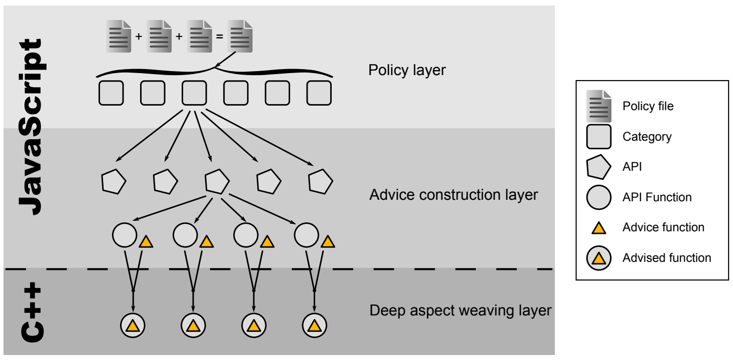


Figure 20: The WebJail architecture consists of three layers: The policy layer, the advice construction layer and the deep aspect weaving layer.

5.1.2 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.

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. To ease up parsing of a policy file, we have chosen to use the JavaScript Object Notation (JSON).

When an iframe is enclosed in another iframe, and both specify a policy, the combinatory rules defined in Subsection 4.1.2 are applied on a per-category basis.

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.

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 arguments: 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.

```
function makeAdvice(whitelist) {
  var myWhitelist = whitelist;

  return function(origf, obj, vp) {
    if(myWhitelist.ROindexOf(vp[0])>=0) {
      return origf.ROapply(obj, vp);
    } else {
      return false;
    }
  };
}

myAdvice = makeAdvice(['foo', 'bar']);
registerAdvice(myFunction, myAdvice);
disableAdviceRegistration();
```

Figure 21: Example advice function construction and weaving

The construction of a rather generic example advice function is shown in Figure 21. 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]` in the example) 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.

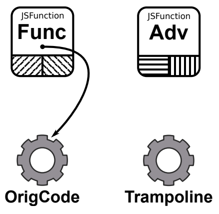
Next, each *(advice, operation)* pair is passed on to the deep aspect weaving layer to achieve the deep aspect weaving.

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.

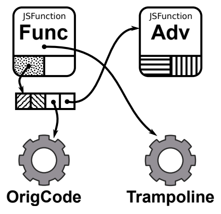
The implementation of the deep aspect weaving layer is inspired by ConScript. To register deep advice, we introduce a new function called `registerAdvice`, which takes 2 arguments: the function to advise (also referred to as the 'original' function) and its advice function. Line 14 of Figure 21 illustrates 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[1].

The process of registering advice for a function is schematically illustrated in Figure 22. 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 22: 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

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

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 22, 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.

5.2 Trusted delegation implementation as an extension to CsFire

The trusted delegation policy (described in Section 4.2) has been implemented as part of the CsFire add-on for Firefox. The CsFire add-on protects end-users from potentially harmful cross-origin traffic by applying the trusted delegation policy we discussed earlier.

Firefox comes with an open architecture is fully aimed at accommodating possible browser extensions. Extension development for Firefox is fairly simple and is done using provided XPCOM components [11]. Our Firefox extension has been developed using JavaScript and XPCOM components provided by Firefox itself.

To facilitate extensions wishing to influence the browsing experience, Firefox provides several possibilities to examine or modify the traffic. For our extension, the following four capabilities are extremely important:

- Influencing the user interface using XUL overlays
- Intercepting content-influencing actions by means of the `content-policy` event
- Intercepting HTTP requests before they are sent by observing the `http-onmodify-request` event (This is the point where the policy needs to be enforced).
- Intercepting HTTP responses before they are processed by observing the `http-on-examine-response` event

When a new HTTP request is received, the policy needs to be actively enforced to render potentially harmful traffic harmless. Next to the trusted delegation policy, the add-on allows explicitly defined exceptions to override the default policy. Implementation of the trusted delegation policy requires to keep track of delegations between sites, as well as redirects between origins, which is achieved using an internal data store.

Enforcing an allow or block decision is straightforward: allowing a request requires no interaction, while blocking a request is simply done by signaling an error to Firefox. Upon receiving this error message, Firefox will abort the request. Stripping authentication information is less straightforward and consists of two parts: stripping cookies and stripping HTTP authentication credentials. Cookies can easily be removed by editing the already present Set-Cookie header. The header containing HTTP authentication credentials however, is not yet available. Firefox can be instructed to leave out this header using the LOAD_ANONYMOUS flag, which is also used to provide private browsing features.

The CsFire add-on is available via the Mozilla add-on platform, and has been downloaded over 33.000 time since its release last year. Starting of version 1.0, the trusted delegation policy, proposed in Section 4.2, is the default security policy of CsFire.

5.3 Safe Wrappers

Safe Wrappers is a hardened implementation of the *lightweight self-protecting JavaScript* approach [13]. *Lightweight self-protecting JavaScript* is an enforcement mechanism which is implemented by inserting an enforcement library and policy code into a web page. The code wraps security-relevant events of JavaScript in the web page with pre-defined policies so that the security-relevant behaviour in the web page is monitored and controlled by the policies, and thus the page becomes *self-protected*.

The injected self-protecting JavaScript code contains two parts: wrapper code and policy code. The wrapper code is the enforcement mechanism implementation which intercepts security-relevant events with a corresponding policy. The self-protecting code is injected into the header of a web page; the body of the page remains unchanged. Injecting the self-protecting code into the header ensures that the self-protecting code is executed first, so it gets to wrap the security critical events before the attacker code can get a handle on them. The injection of self-protecting code can be performed at any point between client (web browser) and server, e.g. at server, or at a trusted proxy, or even as a browser plug-in.

The main challenges for implementing self-protecting JavaScript are *completeness*, ensuring that all security relevant events are intercepted, and *tamper-proofing*, ensuring that the malicious code cannot subvert the monitor mechanism itself. JavaScript provides reflection capabilities, in which code can be loaded and executed at runtime using e.g. the `eval` or `document.write` functions, which makes it difficult to provide completeness. Tamper-proofing is another problem because the enforcement code is placed within the same code base, therefore it can be overwritten by attacker code. The key point to resolve these problems in self-protecting JavaScript is to ensure that the original built-in methods can only be accessible via wrapper methods. Although wrapper methods can be overwritten, the reference to the original built-in method is held uniquely by the wrapper method.

Secure function and object prototype The attacker can modify a global prototype so that it can subvert e.g. built-in methods, or policy object. To prevent attacker code from subverting objects we can try to ensure that each object reference used in the policy is a local property of the object and not something inherited from its low-integrity prototype. The built-in function `hasOwnProperty` can be used for this purpose (of course the integrity of the function `hasOwnProperty` must be maintained as well). But this approach requires all object accesses to be identified and checked. This is potentially tricky for implicit accesses, e.g., the `toString`-function is called implicitly when an object is converted to a string.

Since the monitor code is the first code to be executed it can store local references to the original built-in methods used in the wrapping mechanism. Our solution is to ensure that the wrapper code only uses the locally stored copies of the original methods. As an example, `o.toString()` would be rewritten as `original_toString.apply(o,[])`. To prevent an attacker from subverting the `apply` function of the stored methods, it is made local to each stored function by assignment, i.e. `original_toString.apply=original_apply`. Now even if the prototype of the function is subverted, the `apply` function local to the object remains untouched. Again, this is not entirely foolproof since it could be hard to determine which functions are being called *implicitly*.

Full mediation A specific built-in may have several aliases pointing to the same function in the browser. The wrapper must wrap all of these aliases to have full mediation. The aliases include static and dynamic obtained from other window object (window, frame, iframe). For static aliases, our safe wrapping library compute the aliases in order to wrap all the aliases, and ensure that a policy applied to one function is applied to all its static aliases. For dynamic aliases, we provide pre-defined policies which enforce methods that potentially return a window object.

Trap the caller-chain Caller-chain is an issue (in all major browsers) where calling *caller* argument within a function returns a pointer to the function which called the current function. Thus any user code which is called from within a built-in can obtain a pointer to that built-in using caller which can be to restore the original built-in. We prevent this by wrapping operations on untrusted data in a dummy recursive function, the caller operation can be prevented from reaching the sensitive context.

Safe policies The prototype chain problem also happen for policies when a policy accesses e.g. a whitelist object which the attacker can have side-effect by modifying the *Object* or *Function* prototype. Our solution is that: for functions the policy writer must use local copies of the originals, and for objects we can ensure that they cannot access a poisoned prototype by simply removing it from its prototype chain. Our current implementation is to provide a function called *safe*, which recursively traverses an object, detaching it and all sub-objects from the prototype chain that can be modified by the user. Detaching the object is done by setting its ___proto___ property to null. Since detaching implies that the object will no longer inherit any of the methods expected to be associated with the type of the object, this functionality needs to be restored. Since determining the type of an object is difficult the safe function takes an optional argument to specify the type. Safe versions of the functions associated with this type are added to the object. The safe versions of the functions are stored locally and are detached from the prototype chain to prevent attacker influence.

5.3.1 Declarative Policies

A policy regarding a call to a built-in may depend on the value of its parameters. But inspecting parameters may have side effects, and these side effects are controlled by the untrusted caller but are executed in the context of the policy code. We provide a policy calling mechanism which fix this problem. The idea is that the policy writer writes a policy and an *inspection type* for the argument and the result. The policy code can assume that the parameters are declarative and the wrapper library will ensure this using an inspection type. An inspection type is a specification of the types of the call parameters that will be inspected by the policy code.

The parameter inspection type is an array of types. The following simple grammar of JavaScript literals represents the types used in our current implementation:

$$\begin{array}{lll} \mathsf{type} & ::= & \texttt{'string'} \mid \texttt{'number'} \mid \texttt{'boolean'} \mid \texttt{'*'} \mid \texttt{undefined} \\ & \mid & \{\mathsf{field}_1 : \mathsf{type}_1, \ldots, \mathsf{field}_n : \mathsf{type}_n\} \end{array}$$

The `'*'` type provides a reference to a value without providing access to the value itself. We expect that experience will reveal the need for a more expressive type language, such as sum-types and more flexible matching for parameter arrays – but these should not be problematic to add.

Policies are enforced as follows: the inspection type is used as a pattern to create a clone of the argument array. We will call this the *inspection argument array*. This is the generalization of the idea of *call-by-primitive-value*, except that the cloned parameters also *remove* any parts of the arguments which are not part of the type. The policy logic can only access the inspection argument array. However, when passing the parameters on to any built-in function, we permit the function access to the whole of the argument array. To do this we *combine* the original argument array with the inspection argument array. Figure 23 illustrates this process and Listing 1 outlines the code.

When cloning, the reference type `'*'` is replaced by a fresh dummy object. When combining, each such object is replaced with original value that it represented. Note that the type language does not include functions. This means that policy code cannot inspect any function parameters. However, this does not mean that we cannot have policies on built-in functions which

Example policy computation for some built-in called with (a,b,c). In this example the policy inspects `b` at type string and removes the last character, and sets the third parameter to 42 before calling `proceed()` in order to access the original built-in function. The `foo` field of the return value is incremented before it is returned to the caller. In the diagram ⊥ is an abbreviation for `undefined`, and array objects are depicted as boxes.

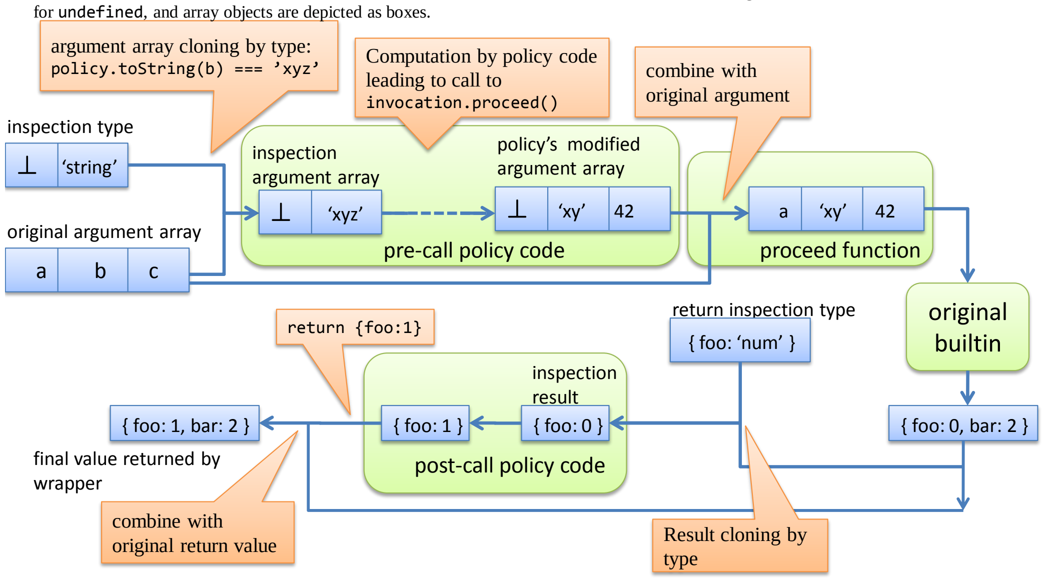


Figure 23: Illustration of policy parameter manipulation

e.g. have callbacks as arguments – it just means that we cannot make policy *decisions* based on the behavior of the callbacks. This restriction to "shallow" types does not seem to be a serious limitation, but more experience is needed to determine if this is indeed the case.

The treatment of the return value of the method is analogous to the treatment of the arguments: a return type specifies what the policy may inspect from the return value. If this is not specified then a return type of ∗ is assumed. The return value of the policy function is combined with the return value produced by the actual built-in.

```
var wrap = function(object, method, policy, inType, retType) {
  // Find function corresponding to alias
  while(!object.hasOwnProperty(method) && object.__proto__)
    object = object.__proto__;
  var original = object[method];

  object[method] = function wrapper() {
    var object = this;
    var orgArgs = arguments, orgRet;
    var polArgs = cloneByType(inType, arguments);
    var proceed = function() {
      orgRet = original.apply(object,
          combine(polArgs, orgArgs));
      return cloneByType(retType, orgRet);
    }
    var polRet = policy(polArgs, proceed);
    return combine(polRet, orgRet);
  };
}
```

Listing 1: Outline of the revised wrapper function supporting inspection types

5.4 Two-Tier Sandbox Architecture

In this section, we present a work-in-progress which proposes an architecture to load and execute untrusted JavaScript within a two-tier sandbox. The idea of the two-tier sandbox architecture is to allow untrusted code to be dynamically loaded and executed without runtime checking or transformation. The execution of untrusted code is enforced by fine-grained security policies which can be specified modularly and specifically to each application.

Given a piece of untrusted JavaScript and an API, we can dynamically load and execute the untrusted code, without static code validation[2], transformation or filtering, in a compartment created by a sandbox environment recently developed by the Google Caja Team[3] which allows untrusted code to interact with a restricted API [17]. Similar to any other sandbox models, the untrusted code can only interact with the outside environment through the API provided to it by the sandbox.

We assume there exists an API library providing baseline access to the hosting page, e.g. the Document Object Model (DOM). The soundness and confinement properties of the API are assumed, and could, for example, be established by an automated tool e.g. [17, 15]. The return value of an API call is safe and has no side-effects. The API may implement some static policies, such as sanitization of HTML input, which applies for any general untrusted application.

Our goal is to specify and enforce modular and fine-grained policies on such an API for specific untrusted applications. As an example, untrusted code is allowed to call `getElementById(..)` method of a API `document` object. If the `document` object provides capability of accessing the `getElementById(..)` method, the untrusted code can call the method to access any element of a page. In practice, a web master of a hosting page may enforce more restriction specific to the page. For example, (1) the untrusted code can only call the function a limited number of specific whitelisted element IDs in the hosting page, (2) the untrusted can call the function at most e.g. 3 times, and (3) enforce further policies on the returned object, for example, allow read-only for a critical element while allow full access to an element that is assigned for the untrusted code to read and write.

In principle, such a fine-grained policy can be embedded within the implementation of the API. However, this makes the implementation too complex and thus error-prone. Indeed, combining policy code with the implementation of a mediator object makes it difficult to maintain. Moreover, defining a new policy or modifying an existing policy requires changing the imple-

[2]Apart from computing a cheap over-approximation of the code's free variables
[3]http://code.google.com/p/es-lab/wiki/SecureEcmaScript

mentation of the API. In addition, application-specific policies should be defined and enforced for a particular untrusted code in one hosting page or a cross different hosting pages. Therefore, it is also inflexible to include policy definition and enforcement within the API implementation.

Our approach for policy enforcement in untrusted JavaScript is to separate policy definition and enforcement from implementation of an API. Before providing the API to untrusted code through a sandbox, the API is enforced by specific policies predefined for the untrusted JavaScript. The policy enforcement is executed within a sandbox environment to ensure that the policy code can only interact with the API and therefore can prevent crass policies which may e.g. return a reference to the original DOM object. The enforced API is returned and then provided as a usual API for the untrusted code through another sandbox environment. This architecture is illustrated in Fig. 24.

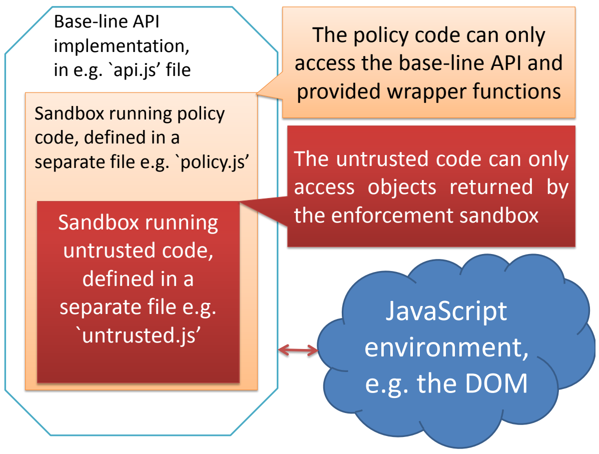

Base-line API implementation, in e.g. `api.js' file
Sandbox running policy code, defined in a separate file e.g. `policy.js'
Sandbox running untrusted code, defined in a separate file e.g. `untrusted.js'
The policy code can only access the base-line API and provided wrapper functions
The untrusted code can only access objects returned by the enforcement sandbox
JavaScript environment, e.g. the DOM

Figure 24: The two-tier architecture

Deploying Untrusted Code Untrusted code in e.g. mashup applications, is usually hosted by a third-party server, and embedded to a hosting page by a `<script>` tag, i.e. `<script src='http://mashup.org/code.js'>`. To deploy untrusted code into a hosting page using the proposed sandbox architecture, the code must be first retrieved as a string in order to be executed within a sandbox environment. There are several alternatives to get a mashup code as a string: (1) Using the Uniform Messaging Policy (UMP) [38]: browsers supporting UMP allow client-side JavaScript can request a content from a cross domain source (2) Server-side support: server-side script may retrieve a mashup source code and inject into the hosting page. In this work, we just assume that the code can be retrieved and uploaded on the hosting server so that it can be loaded at runtime using `XMLHtmlRequest`. The pseudo-code in Listing 2 illustrates this deployment.

```
var api = ...//create an API object
//using XMLHtmlRequest to get the content of file
//'policy.js' into 'policyCode' variable
var moduleMaker = cajaVM.compileModule(policyCode);
var enforcedAPI = moduleMaker(api);
load_untrustedCode(enforcedAPI);
function load_untrustedCode(api){
   //using XMLHtmlRequest to get the content of file
   //'untrustedcode.js' into 'untrustedCode' variable
   var moduleMaker = cajaVM.compileModule(untrustedCode);
   moduleMaker(api);
}
```

Listing 2: Template for enforcing policy for untrusted code in the two-tier architecture

6 Conclusion

This deliverable reported on the exploration of secure composition policies in the context of mashup compositions with third-party components.

To better understand the impact of running arbitrary JavaScript code, the upcoming HTML5 specification and accompanying APIs have been studied, and a list of security-sensitive operations available to mashup components has been enumerated

Next, seven relevant composition scenarios have been selected and analyzed. For each scenario, an informal functional description and the relevant technical data on how the scenario is constructed has been described, as well as an analysis of security requirements for the particular scenario.

In addition, two types of secure composition policies have been developed to express both behavioral constraints as well as communication constraints in mashup compositions.First, the *least-privilege composition policy* enables the specification of allowed privileges granted to a mashup component, and allows secure composition according to the least-privilege security principle. Second, to capture recurring communication patterns between cooperative mashup components (as is the case in third party payment as well as in third party authentication compositions), the *trusted delegation policy* between cooperative components specifies how an integrator can securely delegate control back and forth to a third-party mashup component (e.g. for 3rd party payment). As part of the validation, the two proposed policy types have been applied on the seven small composition scenarios.

Finally, various implementation strategies have been explored to early test the implementation feasibility of the two types of secure composition policies, including the use of JavaScript wrappers, deep aspect technology in the JavaScript engine, as well as browser extensions.

References

[1] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. *Computer Security Foundations Symposium, IEEE*, 0:290–304, 2010.

[2] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 17–30, Berkeley, CA, USA, 2008. USENIX Association.

[3] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Cs-Fire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In F. Massacci, D. Wallach, and N. Zannone, editors, *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 18–34. Springer Berlin / Heidelberg, 2010.

[4] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. In *Lecture Notes in Computer Science*, volume 6879, pages 100–116. Springer, September 2011.

[5] Google. Google Maps. http://maps.google.com/.

[6] I. Hickson and D. Hyatt. Html 5 working draft - cross-document messaging. http://www.w3.org/TR/html5/comms.html#crossDocumentMessages, June 2010.

[7] 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.

[8] IAB. IAB Internet Advertising Revenue Report 2010 Full Year Results. http://www.iab.net/media/file/IAB_Full_year_2010_0413_Final.pdf, April 2011.

[9] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *15th Nordic Conference on Secure IT Systems*, 2010.

[10] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF*, pages 218–232, 2011.

[11] Mozilla. XPCOM. https://developer.mozilla.org/en/XPCOM.

[12] Photobucket. Photobucket. http://photobucket.com/.

[13] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, New York, NY, USA, 2009. ACM.

[14] Pixastic. Pixastic, JavaScript Image Processing Library. http://www.pixastic.com/.

[15] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *20th USENIX Security Symposium*, 2011.

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

[17] A. Taly, J. C. Mitchell, U. Erlingsson, J. Nagra, and M. S. Miller. Automated analysis of security-critical javascript apis. In *Proc of IEEE Security and Privacy'11*. IEEE, 2011.

[18] M. Ter Louw, L. Karthik, T. Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements, 2009.

[19] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Webjail: Least-privilege integration of third-party components in web mashups. In *ACSAC*, December 2011. To appear.

[20] W3C. Clipboard API and events, W3C Working Draft 12 April 2011. http://www.w3.org/TR/clipboard-apis/.

[21] W3C. Cross-Origin Resource Sharing, Editor's Draft 17 November 2010. http://dev.w3.org/2006/waf/access-control/.

[22] W3C. File API, W3C Working Draft 26 October 2010. http://www.w3.org/TR/FileAPI/.

[23] W3C. Geolocation API Specification, Editor's Draft 10 February 2010. http://dev.w3.org/geo/api/spec-source.html.

[24] W3C. HTML 5 Drag and drop, Editor's Draft 14 September 2011. http://www.w3.org/TR/html5/dnd.html.

[25] W3C. HTML 5 Session history and navigation, Editor's Draft 14 September 2011. http://www.w3.org/TR/html5/history.html.

[26] W3C. HTML 5 The audio and video element, Editor's Draft 14 September 2011. http://www.w3.org/TR/html5/video.html.

[27] W3C. HTML5, A vocabulary and associated APIs for HTML and XHTML, Editor's Draft 25 March 2011. http://dev.w3.org/html5/spec/Overview.html.

[28] W3C. HTML5 Web Messaging, Editor's Draft 4 March 2011. http://dev.w3.org/html5/postmsg/#channel-messaging.

[29] W3C. Indexed Database API, W3C Working Draft 19 April 2011. http://www.w3.org/TR/IndexedDB/.

[30] W3C. The Media Capture API, W3C Editor's Draft 02 December 2010. http://dev.w3.org/2009/dap/camera/Overview-API.

[31] W3C. The System Information API, W3C Editor's Draft 16 March 2011. http://dev.w3.org/2009/dap/system-info/.

[32] W3C. Uniform Messaging Policy, Level One, Editor's Draft 15 June 2010. http://dev.w3.org/2006/waf/UMP/.

[33] W3C. Web Notifications, W3C Working Draft 01 March 2011. http://www.w3.org/TR/notifications/.

[34] W3C. Web Storage, Editor's Draft 28 February 2011. http://dev.w3.org/html5/webstorage/.

[35] W3C. WebSocket API, Editor's Draft 10 September 2011. http://dev.w3.org/html5/websockets/.

[36] W3C. XMLHttpRequest, Editor's Draft 4 March 2011. http://dev.w3.org/2006/webapi/XMLHttpRequest/.

[37] W3C. XMLHttpRequest Level 2, Editor's Draft 4 March 2011. http://dev.w3.org/2006/webapi/XMLHttpRequest-2/.

[38] W3C Working Draft. Uniform Messaging Policy. http://www.w3.org/TR/UMP/. Level One.

[39] WebSand. WebSand Deliverable D1.1: Consolidation of state-of-the-art. https://www.websand.eu/deliverables/WP1/D1.1_consolidation_of_stateoftheart.pdf.

[40] WebSand. WebSand Deliverable D1.2: Interaction Patterns and Security Properties.

[41] M. Zalewski. Browser security handbook. Whitepaper, Google Inc., http://code.google.com/p/browsersec/wiki/Main, (01/13/09), 2008.