Caja Rewriting of OpenSocial Gadgets

December 01, 2007

Introduction

This document describes how a compatible Caja translator must rewrite OpenSocial gadgets, and the interface that the rewritten gadgets will present to a gadget container.

Portions rewritten

An OpenSocial gadget is an XML document that looks like the following simple example. (See the OpenSocial documentation for a normative description of the gadget format.)

```
<?xml version="1.0" encoding="UTF-8"?>
<Module>
  <ModulePrefs title="My gadget">
    <Require feature="opensocial-1.0"/>
  </ModulePrefs>
  <Content type="html">
    <![CDATA[
      content area
    ]]>
  </Content>
</Module>
```

We will rewrite the material in the *content area* as described below. All other material will be passed through verbatim, *modulo* sanitizing of the element attribute values.

Content area rewriting

As provided by the gadget author, the content area is an HTML document fragment containing a number of markup, `style` and `script` tags, in arbitrary order. The rewritten content area will always be of the form:

```
<style>
  consolidated stylesheet content
</style>
<script>
  ___.loadModule(function(___OUTERS___) {
    translated gadget content
  });
</script>
```

where we define:

___ – the Caja global scope.

`loadModule` – the Caja module loading function.

`___OUTERS___` – a variable that will represent the sandboxed global scope to the translated JavaScript code. Compatible rewriters may use a different name for this variable, since it is not used outside the gadget, but the variable must end in a triple underscore, "`___`".

consolidated stylesheet content – all the material appearing in `style` tags in the original, consolidated into one section and rewritten.

translated gadget content – the union of the JavaScript in the original `script` tags, interspersed with more JavaScript that explicitly writes the literal HTML in the original.

In subsequent sections, we describe how we handle specific elements of the content area. At a highest level, this process is parameterized by a single input, a prefix for CSS names.

Rewriting `script` tags

In the case of a `script` tag containing literal script source, the gadget translator will apply the usual Caja translation rules and include the results in the translated gadget content section.

For scripts that retrieve their content using a `src` attribute, the translator will, at its option, either (recursively) retrieve and inline all the script content, or rewrite the target URL of the `src` attribute so that the retrieved content is passed through a well-known Caja JavaScript rewriting proxy. The translator will correctly consider the case where the `src` attribute specifies a relative URL. The URL will be retrieved from JavaScript code, so the rewritten gadget will still have only one `script` tag.

Rewriting styles

Styles will be rewritten to have a gadget-specific prefix (and corresponding HTML tags will be rewritten to match), so they affect only the containing gadget. Dangerous features of CSS will be rejected. All styles will be consolidated into one `style` element in the gadget, as noted above, which will not be accessible to user JavaScript code.

If multiple instances of the same gadget exist in the same container page, they will safely load their `style` elements multiple times.

TBD: How do we deal with `<link rel="stylesheet">`? Is this allowed in gadgets anyway?

JavaScript rewriting

The JavaScript will be rewritten per the Caja rules, specified elsewhere.

The `___OUTERS___` parameter supplied by the `loadModule` function is an object containing bindings of the DOM wrappers and OpenSocial APIs, as follows:

`_IG_*` – safely wrapped versions of the corresponding iGoogle/OpenSocial

functions.

`opensocial` – safely wrapped version of the root object of the OpenSocial APIs.

TBD: Be more specific.

JavaScript in HTML tags

NOTE: The following describes a provisional mechanism; this is under discussion.

Where we have JavaScript in HTML tags, such as:

```
<a onmouseover="alert(1)">Mouse here</a>
```

note first that, naively, we will at least rewrite this to a `document.write` call, as in:

```
___OUTERS___.document.write("
  <a onmouseover=\"
    ___OUTERS___.alert(1)\
  ">Mouse here</a>
");
```

However, we wish to confine the emitted code to the `___OUTERS___` of the specific gadget *instance*. We do this by requiring that the container provide the following function returning a string:

```
___OUTERS___.getId___
```

the return value of which can be used to gain access to the `___OUTERS___` object via a global function `getOuters`. In other words, for any instance of a gadget:

```
___.getOuters(___OUTERS___.getId___()) === ___OUTERS___
```

The value of the `id___` attribute can be used to securely retrieve the `___OUTERS___` object they were created with. Thus our final form of the rewritten HTML would be:

```
___OUTERS___.document.write("
  <a onmouseover=\"
    (function(___OUTERS___) {
      ___OUTERS___.alert(1);
    })(___.getOuters(" + ___OUTERS___.getId___() + "))
  ">Mouse here</a>
");
```

The `getId___` function is unreadable to Caja code, which therefore cannot use it to sense information about its container. `getId___` must return `string` values that are the string representation of integers, in other words, matching `/[0-9]+/`. Typically, a container assign will assign a monotonicallly increasing sequence of integers to be sure that values are non-colliding. The container may assign the values lazily.

Attribute value sanitizing

TBD: Discuss threat model and how it will be addressed.