Attacks on JavaScript Mashup Communication

Adam Barth
UC Berkeley

Collin Jackson
Stanford University

William Li
UC Berkeley

Abstract

In a mashup, two principals wish to communicate without ceding complete control to each other. In this paper, we analyze whether existing and proposed JavaScript mashup communication mechanisms have this security property. We show that a failure to account for details of JavaScript often lets one communicant completely compromise the other. We illustrate these vulnerabilities with proof-of-concept privilege escalation attacks. Based on our analysis, we recommend that mashup communication mechanisms prevent privilege escalation by using lexical authorization across a specified interface that enforces type checks and allows the communicants to exchange only primitive values. We observe that we can implement such a mechanism in today's browsers using `postMessage` *as a primitive. We demonstrate our approach by implementing a version of the Google Maps gadget that can be used without ceding complete control to Google.*

1 Introduction

Browsers typically isolate different Web sites from each other. In a mashup scenario, an *integrator* seeks to overcome this restriction and communicate with another Web site (often called a *gadget*) to produce a richer user experience. However, the integrator does not wish to entrust the third-party Web site with its full authority. To address this problem, a number of researchers have proposed a dizzying array of new communication mechanisms [21, 10, 16, 4, 6, 11, 7, 5, 13, 3] that aim to provide this controlled interaction. In creating these mashup communication mechanisms, researchers must make a number of design decisions that impact whether their schemes achieve this security goal. In this paper, we analyze the security of these schemes and find a number of vulnerabilities.

The first decision in designing a mashup communication mechanism is whether to use the browser to enforce access control or to follow an object-capability discipline. Although a number of mashup communication mechanisms (e.g., Caja [16], ADsafe [4], and FBJS [6]) have had success with an object-capability approach, we focus on mechanisms based on access control in this paper. After choosing access control, we are faced with a series of further design decisions:

1. **Lexical vs. Dynamic.** When the browser performs an access control check, the browser must determine the currently active principal. Different browsers use different algorithms for computing the active principal [10]. Some browsers use lexical authorization, which selects the principal that defined the most recent callee, and others use dynamic authorization, which selects the principal that defined the first caller. Dynamic authorization is problematic when the integrator directly calls a gadget method because the gadget's method can act with the integrator's authority.

2. **Interfaces vs. Asymmetry.** A mashup communication mechanism can either treat the integrator and the gadget as two mutually distrusting principals that communicate over a defined interface, or the mechanism can replace the symmetric same-origin policy with an asymmetric policy that lets the integrator access the gadget but not vice-versa. The asymmetric paradigm leads to security problems because the gadget can fool the integrator by replacing built-in browser APIs with malicious functions. Often the gadget can completely compromise the integrator, even under lexical authorization, by abusing various JavaScript pointers leaked during the function call.

3. **Typed vs. Untyped.** Even when communication is restricted to an interface between two mutually distrusting principals, one principal has many opportunities for attacking another principal because JavaScript is an untyped language. By passing unexpected parameters through the interface, the caller can mislead the callee into being a confused deputy [8]. For example, the caller can pass the callee its own global window object, which the callee might mistakenly operate upon. These attacks can be largely mitigated by using a typed interface that blocks these unexpected parameters.

4. **Values vs. Objects.** Even a typed interface can be dangerous if one principal leaks a JavaScript

object to another principal. By following various implicit pointers, such as `__proto__`, a malicious principal can corrupt sensitive objects, such as `Object.prototype`, of the honest principal. After corrupting the `Object.prototype` object, the attacker can use various techniques to hijack the honest principal's control flow and trick the honest principal into being a confused deputy.

We illustrate these security pitfalls with concrete examples using publicly available implementations of mashup communication mechanisms. In most cases, these pitfalls lead to a complete compromise of the honest principal under mild assumptions.

Following this chain of reasoning, we recommend that mashup communication mechanisms use lexical authorization over a typed interface between mutually distrusting principals that permits only JavaScript values (i.e., not objects) to be exchanged. Fortunately, we can implement such a mashup communication mechanism in the current generation of browsers using `postMessage` [9], which lets mutually distrusting frames exchange primitive strings. We demonstrate this approach by creating a version of the Google Maps gadget that does not require the integrator to trust Google. Our implementation of the `GMap2` interface uses a design analogous to DCOM [15] and forwards interface calls (via `postMessage`) to an iframe on an untrusted domain that actually displays the map.

Organization. Section 2 details the security implications of four decisions in designing a mashup communication mechanism. Section 3 presents a secure mashup design that uses `postMessage` as the underlying communication primitive. Section 4 discusses alternative solutions to these security issues. Section 5 concludes.

2 Design Decisions

In this section, we analyze four design decisions in mashup communication mechanisms. Although various mashup designs have made different design decisions, we find that these decisions lead to security issues, which we demonstrate with concrete proof-of-concept attacks. Examining these design decisions largely forces our hand in designing a mashup communication mechanism.

2.1 Lexical vs. Dynamic

Whenever the browser performs an access control check to decide whether to authorize a given action, the browser must determine which principal is requesting to perform the action. Popular browsers use one of two common algorithms for determining the active principal [10]:

- **Lexical Authorization.** Under lexical authorization, the browser computes the active principal based on the security origin of the document that contains the source code of the *last* JavaScript function invoked by the JavaScript engine. This is analogous to the lexical scoping rules used to look up global variables at the current program point.
- **Dynamic Authorization.** Under dynamic authorization, the browser computes the active principal based on the security origin of the document that contains the source code of the *first* JavaScript function invoked by the JavaScript engine. This is analogous to the dynamic scoping rules used to look up exception handlers at the current program point.

Internet Explorer 7 and Firefox 3 use lexical authorization of principals, but Safari 3, Chrome 1, and Opera 9.26 use dynamic authorization.

Unfortunately, dynamic authorization is problematic for mashups. Imagine that a gadget exposes a `getPublicInterface` method that is called by an integrator.

```
var i = frames[0].getPublicInterface();
```

Under dynamic authorization, the getPublicInterface method runs with the caller's authority. Thus, if the gadget is malicious, the gadget can abuse the caller's authority to hijack the privileges of the caller:

```
function getPublicInterface() {
  top.setTimeout("... attack code ...",
                 0);
}
```

We recommend that browsers adopt lexical authorization to avoid these privilege hijacking vulnerabilities. We have collaborated with Apple and Google to implement lexical authorization in Safari 4 and Chrome 2. We proposed lexical authorization to the HTML 5 working group, and the current HTML 5 draft specification now requires lexical authorization.

2.2 Interfaces vs. Asymmetry

One paradigm for letting different principals interact is to replace the usual symmetric same-origin policy with an *asymmetric access policy* that lets a "more trusted" principal access a "less trusted" principal (but not vice-versa). For example, the OpenSandbox [21] proposal lets content outside of the sandbox access content inside the sandbox but aims to prevent content inside the sandbox from escaping. The Web Inspector, a developer tool found in Safari and Chrome, also uses an asymmetric access policy to interact with the inspected page.

Asymmetric access policies are useful in several scenarios. For example, a library author might publish their code for all to use and have no expectation of confidentiality or integrity. Also, asymmetry does not require the "gadget" to opt in to the mashup explicitly, which often facilitates useful opportunistic applications beyond the scope of what the content author planned. For this reason, asymmetric access policies are used by Web developer tools to poke around at the internals of an oblivious Web page.

Because no public implementation of OpenSandbox is available as yet, we illustrate the potential challenges of asymmetry using the Web Inspector. The Web Inspector is implemented in HTML and JavaScript and is allowed to access any document (in order to debug the frame), but no documents are allowed to access the Web Inspector. This asymmetric access policy creates security challenges for the Web Inspector. In particular, we discovered the Web Inspector contains the following line of code,

```
var result = doc.querySelectorAll(query);
```

which calls the `querySelectorAll` method (from the Selectors API [20]) of the untrusted document being inspected, a pointer to which is stored in the variable `doc`.

The Web Inspector believes that this line of code calls the browser's built-in `querySelectorAll` method. However, this might not be the case because browsers let Web pages alter the built-in APIs to facilitate interoperability. For example, a Web page can simulate Internet Explorer's `attachEvent` API for registering an event handler using Firefox's `addEventListener` API. By overriding its own `querySelectorAll` method, the attacker can hijack control and abuse the fact that not all browser APIs perform access checks to inject arbitrary script into the Web Inspector, even in browsers that use lexical authorization.

```
function evilFunc() {
  var obj = evilFunc.caller;
  while (obj.arguments.length == 0 ||
         !obj.arguments[0].target) {
    obj = obj.caller;
  }
  var victimDocument = obj.
    arguments[0].target.
    ownerDocument;
  victimDocument.body.innerHTML =
    "<img onerror='...'>";
}

document.querySelectorAll = evilFunc;
```

Once the Web Inspector calls the attacker's function, the attacker can abuse a number of rarely used pointers, such as `caller` and `arguments`, to obtain a JavaScript pointer to the Web Inspector's `document` object. The attacker's function uses `caller` to walk to the runtime stack, `arguments` to reach the event that generated the call stack (in this case a keyboard event), and `ownerDocument` to move from the event object to the Web Inspector's `document` object. The browser does not enforce access control checks for the `document` object because the object is not normally visible to other principals. The attacker can use the unchecked `innerHTML` API to inject arbitrary script into the Web Inspector's document. The injected script runs with the Web Inspector's universal privileges.

We reported this vulnerability to `webkit.org` on November 15, 2007. Apple patched this security vulnerability in Safari 3.1 by changing the vulnerable line of code as follows:

```
var result = Document.prototype.
             querySelectorAll.
             call(doc, query);
```

Instead of calling the `querySelectorAll` method of the untrusted `doc` object, which might have been overridden by the attacker, the fixed code calls the Web Inspector's own `querySelectorAll` method on the untrusted `doc` object. This approach secures this line of code but is difficult to apply systematically to the entire Web Inspector.

2.3 Typed vs. Untyped

Because JavaScript is an untyped language, functions accept arguments of any type. This behavior lets developers create simple programs quickly by eliminating unnecessary type annotations. However, when a malicious principal calls another principal's function with arguments of an unexpected type, the function can behave in ways not intended by its author. The function's author can manually check the type of each argument using JavaScript's reflection facilities, but the practice of manually type checking JavaScript arguments is relatively rare and error prone.

If a function does not check the type of its arguments, a malicious caller can often escalate his or her privileges by calling the function with unexpected arguments. Operations that are harmless on arguments of one type might be dangerous on arguments of another type. By passing an unexpected argument, the caller can often trick the callee into misusing its authority to perform an operation that the caller cannot perform itself, an attack known as a *confused deputy* attack [8]. In this case, the deputy is the privileged function, which is fooled by its caller into misusing the authority of the document that defined it.

Consider a utility function `deref` that is designed to look up an index in an array:

```
function deref(arr, index) {
  return arr[index];
}
```

If exposed to an attacker, `deref` can leak confidential information because the attacker can use `deref` to read properties of objects that would normally throw a security exception if the attacker attempted to read them directly. For example, the `window` object is available across origins, but most properties of the `window` object are protected by access control checks. However, the attacker can use `deref` to bypass these checks:

```
var doc = deref(frames[0], "document");
var cookie = deref(doc, "cookie");
```

Because of lexical authorization, `deref` runs with the authority of its author (in this case `frames[0]`) and passes the access control checks.

Of course, `deref` is not the only function that an attacker can confuse. There are a number of other functions that are devastating if leaked to an attacker:

- **Indirect assignment.** If the attacker obtains a function that performs an indirect write, such as

  ```
  function assign(a,b,c) { a[b] = c; }
  ```

  then the attacker can inject arbitrary script into the victim's security context by setting the victim's window location to a `javascript:` URL:

  ```
  assign(frames[0],
         "location",
         "javascript:// attack code");
  ```

- **Substitution.** A commonly used method of strings in JavaScript is `replace`, which performs regular expression substitution. Coincidentally, a window's `location` object also has a method called `replace`, which is access checked and can be used to execute arbitrary script via `javascript:` URLs. If the gadget exposes a function like

  ```
  function replace(a,b) {
    a.replace(b, '');
  }
  ```

  then the attacker can pass in `window.location` (which is visible across origins) as the first argument and a malicious `javascript:` URL as the second argument.

- **Indirect call.** If the attacker obtains a function that calls functions stored in an array, such as `function(a,b,c) { a[b](c); }`, the attacker can invoke `setTimeout` by passing the victim's window object as the first parameter.

- **Prototype.** The Prototype JavaScript library [19] augments built-in browser interfaces with a variety of other methods that can be used as confused deputies. For example, Prototype augments array objects with an `invoke` method that behaves like the indirect call function above. If the victim uses the Prototype library, the victim must avoid exposing functions that return hashes, strings, or arrays because these types of objects leak functions that evaluate arbitrary script.

Stronger typing of arguments can mitigate these confused deputy attacks because, in most cases, the attacker will not be able to confuse the functions by passing the victim's window object as an argument. Rather than requiring that every implementation manually check the types of its arguments, we recommend using a typed Interface Description Language (IDL) [12] to describe the interface. In particular, we recommend WebIDL [14], which is an IDL used to specify the behavior of the web platform.

2.4 Values vs. Objects

Unlike objects in languages like Java, each JavaScript object contains a number of pointers to other objects. For example, each object contains a pointer to its prototype object, from which the object inherits many of its properties and methods. Current browser implement access control checks only when a script calls a Document Object Model (DOM) API. Actions within the JavaScript engine are not constrained by a reference monitor. These lax access control checks and proliferation of JavaScript pointers are problematic for passing objects between trust domains. Consider the following gadget that exposes a minimal interface with no methods:

```
function getPublicInterface() { }
```

Elsewhere in the gadget, the gadget defines this private utility function:

```
function store(x,y,z) {
  if (y != "")
    x[y] = z;
}
```

This function would be vulnerable to confused deputy attacks if it were exposed as an interface, using the attacks described in Section 2.3. However, the integrator can abuse this function even though it is not in the gadget's interface:

```
function evilValueOf() {
  var args = evilValueOf.caller.arguments;
  args[0] = top.location;
  args[1] = "href";
  args[2]  = "javascript:/*attack code*/";
}
```

```
frames[0].getPublicInterface
          .__proto__.__proto__
          .valueOf = evilValueOf;
```

The attack proceeds in 5 stages:

1. From the getPublicInterface function object itself, the attacker uses __proto__ to obtain a reference to the gadget's Object.prototype object.

2. The attacker installs a method named valueOf the gadget's Object prototype. The JavaScript language contains many implicit calls to valueOf, such as when using the == operator.

3. When the gadget utility function compares y to "", JavaScript interpreter invokes evilValueOf, transferring control to the attacker.

4. The attacker can use the Function.caller API to modify the arguments of its caller (the store function).

5. When control returns to the gadget, the store function becomes a confused deputy and launches the attacker's code by navigating the window to a malicious javascript: URL.

The toString method can also be used to hijack control. Imagine that our gadget uses the jQuery library [17] and calls alert($('body')). The attacker could replace the gadget's Object.prototype.toString method with a malicious function that abuses the privileged this pointer:

```
function evilToString() {
  this.append("<img src='' \
                    onerror='...'>");
}
frames[0].getPublicInterface
          .__proto__.__proto__
          .toString = evilToString;
```

Once an untrusted frame has a JavaScript pointer to a trusted object, there are a number of other exploit techniques, such as hijacking global variables [2]. Although meticulous developers might be able to implement a gadget that sidesteps these attacks, for example by avoiding == and global variables, implementing such a gadget by hand would be highly error prone. (Automated tools might be of some help, however.) Instead of sharing objects between trust domains, we recommend sharing values, which do not leak pointers.

3 PostMash

Fortunately, a value-based mechanism for communication between trust boundaries already exists: postMessage. This API has been specified in HTML 5 and implemented in Internet Explorer 8, Firefox 3, Safari 4, Chrome 2, and Opera 10. The postMessage API provides a confidential, authenticated channel [1] between two mutually distrusting frames (provided the sender specifies a targetOrigin and the receiver validates the origin property of the message). Instead of introducing an asymmetric access policy, postMessage lets two security origins exchange primitive strings by specifying whom can receive the string and from whom the string was received.

We can use postMessage to solve many of the same problems that motivate others to introduce asymmetric access policies. For example, we can use postMessage to simulate an access policy akin to OpenSandbox [21], which allows the integrator complete access to the gadget, using postMessage:

```
addEventListener("message", function(e){
  if (e.origin === "http://example.com")
    eval(e.data);
}, false);
```

By sending only primitive strings, postMessage avoids the challenges of direct object accesses between mutually distrusting frames.

We can also use postMessage to simulate the getPublicInterface API of OMash [3]. Instead of directly exposing an object with methods, the gadget listens for messages sent with postMessage. To call a method, the integrator sends a JSON string via postMessage that describes which method to call and contains serializations of the method's arguments.

3.1 Design

We suggest a postMessage-based mashup design analogous to DCOM [15], which we call *PostMash*. To interact with a gadget, the integrator uses a small stub library that exposes the gadget's interface. To implement the interface, the stub library creates an iframe to an untrusted origin, such as http://s24601.dfjaofije.com, which then includes the gadget implementation. Whenever the integrator calls a method in the stub library, the library serializes the method call to a string and sends the string to the untrusted frame using postMessage.

In a PostMash mashup, the stub library can be written either by the integrator or by the gadget author. In some cases, an integrator can create an "opportunistic mashup" using PostMash by writing a stub library and loading an

unsafe gadget (like Google Maps) in an untrusted iframe. When the gadget author provides the stub library, the library can be re-used by many different integrator, but each integrator much audit the stub library for security because the library runs in the integrator's security context. However, because the stub library simply proxies interface calls to the untrusted frame, the library is much less complex than the full gadget and can more easily be verified by static analysis techniques (such as Caja [16] or ADSafe [4]).

3.2 Case Study: Google Maps

To evaluate the feasibility of the PostMash design, we used PostMash to republish a less privileged version of the widely used Google Maps gadget. To use the standard Google Maps gadget, the integrator must run a script from `http://maps.google.com`, requiring the integrator to trust Google. This requirement is problematic for competing Web sites, such as Yelp, that might wish to use the gadget without trusting Google.

We were able to create a stub library that largely mimicked the `GMap2` API, making it easy for sites to port their existing uses of the unsafe Maps gadget to our safe version. For example, the API for opening an "info window" is identical:

```
map.openInfoWindow(
    new GLatLng(37.4419, -122.1419),
    "Hello, world");
```

The stub library serializes the method call to the following JSON string:

```
{
  "method": "openInfoWindow",
  "point": {
    "lat":37.4419,
    "lng":-122.1419
  },
  "elements": "Hello, world"
}
```

One limitation of this approach is that some parts of the `GMap2` API are synchronous, but `postMessage` enables only asynchronous communication with the gadget. For example, the `getCenter` API synchronously returns the latitude and longitude of the center of the map. Instead of returning the result synchronously, our stub library returns the result via an asynchronous callback:

```
map.getCenter(function(center){
  map.openInfoWindow(
      center, "Hello, world");
});
```

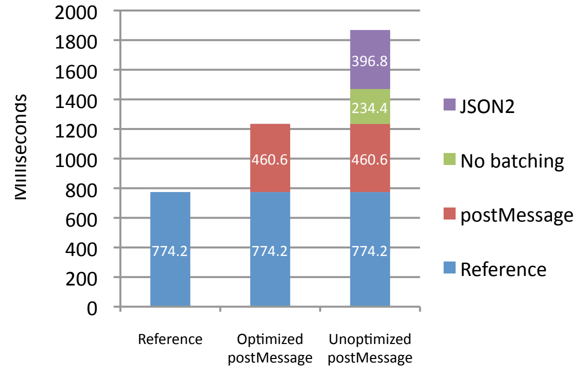

Figure 2. Adding 100 markers (Firefox 3.5)

Because `postMessage` cannot transmit object references, we serialize object references using opaque *handles*, which we implement concretely using integers. The gadget maintains a table mapping handles to objects and replaces handles with the appropriate objects when deserializing messages. For example, when the integrator creates a map marker, the actual `GMarker` object is stored in the gadget, and the integrator is given an opaque handle to the marker. If the integrator later wishes to move the marker, the integrator specifies which marker to move by its handle.

3.3 Performance

We evaluated the performance of the PostMash implementation of Google Maps using a simple benchmark that creates a map and adds 100 markers to the map. For each observation, we ran the benchmark 10 times in Firefox 3.1 Beta 3 (the latest version available at the time). Before optimizing performance, we observed a 100% slowdown compared to the unsafe Google Maps gadget.

- **Batching.** Because every PostMash method call is asynchronous, we can batch together method calls to reduce the number of messages exchanged between the integrator and the gadget. To batch method calls, the stub library appends new method calls to a buffer and flushes the buffer every 50 milliseconds. Batching improved performance by approximately 20%.

- **Native JSON.** Some newer browsers (including Firefox 3.5 and Internet Explorer 8) have native support for serializing and deserializing JSON. Using the native JSON parser instead of the JavaScript-based `json2` library improved our benchmark score by approximately 27%. We encourage browser vendors that do not currently provide native JSON support to include native JSON support in future releases.

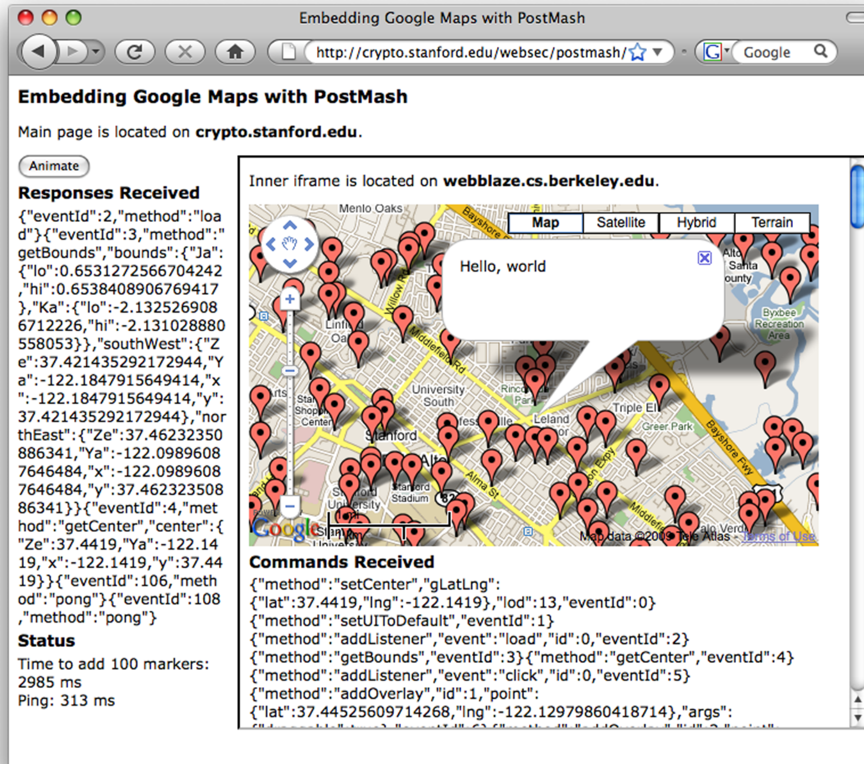

Figure 1. Demonstration of a page using the PostMash approach to embed Google Maps.

After implementing these performance optimizations, we reduced the slowdown due to PostMash to 60%.

4 Alternatives

Native Wrappers. Firefox's XPCNativeWrappers [22] use an alternative approach to securing asymmetric access policies. Instead of giving the integrator's scripts access to the gadget's JavaScript environment, XPCNativeWrappers give the integrator a direct view of the gadget's "native" Document Object Model, ignoring the gadget's JavaScript environment entirely. This prevents the gadget from usurping the integrator's privileges using the techniques in Section 2.2 because the integrator is immune to the gadget's modification of the built-in APIs. Native wrappers are appropriate when the integrator desires only access to the gadget's document but preclude the integrator from calling any JavaScript functions defined by the gadget. In the Google Maps Gadget example, the integrator could directly manipulate the HTML elements that comprise the map, but the integrator would be unable to call the `setCenter` API to scroll the map.

File URLs in Firefox. For the Firefox 3 release, the Firefox developers wanted to mitigate attacks from local HTML files by granting a file access only to other files in its own directory and in subdirectories. This access policy is similar to an asymmetric mashup communication mechanism because `file:///foo/alpha` can access `file:///foo/bar/beta` but not vice-versa. Firefox 3 shipped with this policy for network access, such as XMLHttpRequest, but used a different policy for accessing objects in memory [18] that embraces the difficulties in securing an asymmetric access policy: whenever a "more trusted" file interacts with a "less trusted" file, the "less trusted" file is explicitly granted the privileges of the "more trusted" file. This design achieves the security goal of the file URL restrictions (preventing downloaded HTML files from easily reading `/etc/passwd`) but provides insufficient isolation for mashups.

5 Conclusions

Over the last few years, researchers and browser vendors have added new communication mechanisms between sites in the hopes of enabling new and compelling mashup applications. A key security requirement for these mechanisms is that one principal can communicate with another without becoming completely compromised. In this paper, we discussed four design decisions that affect whether the communication mechanism achieves this goal: lexical vs. dynamic, interfaces vs. asymmetry, typed vs. untyped, and values vs. objects. We illustrate the security consequences of these decisions with concrete privilege escalation attacks.

Analyzing the mashup design space according to these decisions leads us to recommend a mashup communication mechanism that uses lexical authorization and a typed interface that lets mutually distrusting parties exchanged JavaScript values but does not let them exchange JavaScript objects. We observe that we can implement this mashup communication mechanism in today's browsers using `postMessage` as the underlying communication mechanism. We demonstrate this technique by creating a less privileged version of the Google Maps gadget.

References

[1] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[2] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[3] Steven Crites, Francis Hsu, and Hao Chen. OMash: enabling secure web mashups via object abstractions. In *Proc. of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[4] Douglas Crockford. ADsafe. `http://adsafe.org/`.

[5] Douglas Crockford. The <module> tag, 2006. `http://www.json.org/module.html`.

[6] Facebook. FBJS, 2008. `http://wiki.developers.facebook.com/index.php/FBJS`.

[7] Rui Guo, Bin B. Zhu, Min Feng, Aimin Pan, and Bosheng Zhou. Compoweb: a component-oriented web architecture. In *Proceedings of the 17th International World Wide Web Conference*, 2008.

[8] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[9] Ian Hickson et al. Cross-document messaging, 2009. `http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html#crossDocumentMessages`.

[10] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proc. of the 16th International World Wide Web Conference. (WWW 2007)*.

[11] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: secure component model for cross-domain mashups on unmodified browsers, 2008.

[12] David Alex Lamb. IDL: sharing intermediate representations. *ACM Trans. Program. Lang. Syst.*, 9(3):297–318, 1987.

[13] Anthony Lieuallen, Aaron Boodman, and Johan Sundström. Greasemonkey. `https://addons.mozilla.org/en-US/firefox/addon/748`.

[14] Cameron McCormack et al. Web IDL, 2008. `http://www.w3.org/TR/WebIDL/`.

[15] Microsoft. Distributed component object model (DCOM) remote protocol specification. `http://msdn.microsoft.com/en-us/library/cc201989.aspx`.

[16] Mark Miller. Caja, 2007. `http://code.google.com/p/google-caja/`.

[17] John Resig and the jQuery Team. jQuery. `http://jquery.com/`.

[18] Eric Shepherd and Boris Zbarsky. Same-origin policy for file: URIs, 2009. `https://developer.mozilla.org/En/Same-origin_policy_for_file:_URIs`.

[19] Prototype Core Team. Prototype JavaScript framework. `http://www.prototypejs.org/`.

[20] Anne van Kesteren et al. Selectors API, 2008. `http://www.w3.org/TR/selectors-api/`.

[21] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[22] Boris Zbarsky et al. XPCNativeWrapper. `https://developer.mozilla.org/en/XPCNativeWrapper`.