Securing Frame Communication in Browsers

Adam Barth
Stanford University
abarth@cs.stanford.edu

Collin Jackson
Stanford University
collinj@cs.stanford.edu

John C. Mitchell
Stanford University
mitchell@cs.stanford.edu

Abstract

Many web sites embed third-party content in frames, relying on the browser's security policy to protect them from malicious content. Frames, however, are often insufficient isolation primitives because most browsers let framed content manipulate other frames through navigation. We evaluate existing frame navigation policies and advocate a stricter policy, which we deploy in the open-source browsers. In addition to preventing undesirable interactions, the browser's strict isolation policy also hinders communication between cooperating frames. We analyze two techniques for inter-frame communication. The first method, fragment identifier messaging, provides confidentiality without authentication, which we repair using concepts from a well-known network protocol. The second method, `postMessage`, provides authentication, but we discover an attack that breaches confidentiality. We modify the `postMessage` API to provide confidentiality and see our modifications standardized and adopted in browser implementations.

1 Introduction

Web sites contain content from sources of varying trustworthiness. For example, many web sites contain third-party advertising supplied by advertisement networks or their sub-syndicates [6]. Other common aggregations of third-party content include Flickr albums [12], Facebook badges [9], and personalized home pages offered by the three major web portals [15, 40, 28]. More advanced uses of third-party components include Yelp's use of Google Maps [14] to display restaurant locations and the Windows Live Contacts gadget [27]. A web site combining content from multiple sources is called a *mashup*, with the party combining the content called the *integrator* and integrated content called a *gadget*. In simple mashups, the integrator does not intend to communicate with the gadgets and requires only that the browser isolate frames. In more complex mashups, the integrator does intend to communicate with the gadgets and requires secure inter-frame communication.

In this paper, we study the contemporary web version of a recurring problem in computer systems: isolating untrusted, or partially trusted, software components while providing secure inter-component communication. Whenever a site integrates third-party content, such as an advertisement, a map, or a photo album, the site runs the risk of incorporating malicious content. Without isolation, malicious content can compromise the confidentiality and integrity of the user's session with the integrator. While the browser's well-known "same-origin policy" [34] restricts script running in one frame from manipulating content in another frame, the browser uses a different policy to determine whether one frame is allowed to navigate (change the location of) another frame. Although restricting navigation is essential to providing isolation, navigation also enables one form of inter-frame communication used in mashup frameworks from leading companies. Furthermore, we show that an attacker can use frame navigation to attack another inter-frame communication mechanism, `postMessage`.

Isolation. We examine the browser frame as an isolation primitive. Because frames can contain untrusted content, the browser's security policy restricts frame interactions. Many browsers, however, insufficiently restrict the ability of one frame to navigate another frame to a new location. These overly permissive frame navigation policies lead to a variety of attacks, which we demonstrate against the Google AdSense login page and the iGoogle gadget aggregator. To prevent these attacks, we propose tightening the browser's frame navigation policy while maintaining compatibility with existing web content. We have collaborated with browser vendors to deploy this policy in Firefox 3 and Safari 3.1. As the policy is already implemented in Internet Explorer 7, the policy is now deployed in the three most-used browsers.

	Confidentiality	Authentication	Network Analogue
Fragment identifier channel	✓		Public Key Encryption
`postMessage` channel		✓	Public Key Signatures
`postMessage` (our proposal)	✓	✓	SSL/TLS

Table 1: Security properties of frame communication channels

Communication. With strong isolation, frames are limited in their interactions, raising the issue of how isolated frames can cooperate as part of a mashup. We analyze two techniques for inter-frame communication: fragment identifier messaging and `postMessage`. The results of our analysis are summarized in Table 1.

- *Fragment identifier messaging* uses characteristics of frame navigation to send messages between frames. As it was not designed for communication, the channel has less-than-desirable security properties: messages are confidential but senders are not authenticated. To understand these properties, we draw an analogy between this communication channel and a network channel in which senders encrypt their messages to their recipient's public key. For concreteness, we examine the `Microsoft.Live.Channels` library [27], which uses fragment identifier messaging to let the Windows Live Contacts gadget communicate with its integrator. The protocol used by Windows Live is analogous to the Needham-Schroeder public-key protocol [29]. We discover an attack on this protocol, related to Lowe's anomaly in the Needham-Schroeder protocol [23], in which a malicious gadget can impersonate the integrator to the Contacts gadget. We suggested a solution based on Lowe's improvement to the Needham-Schroeder protocol [23], and Microsoft implemented and deployed our suggestion within days.

- *postMessage* is a new browser API designed for inter-frame communication [19]. `postMessage` is implemented in Opera, Internet Explorer 8, Firefox 3, and Safari. Although `postMessage` has been deployed since 2005, we demonstrate an attack on the channel's confidentiality using frame navigation. In light of this attack, the `postMessage` channel provides authentication but lacks confidentiality, analogous to a channel in which senders cryptographically sign their messages. To secure the channel, we propose a change to the `postMessage` API. We implemented our change in patches for Safari and Firefox. Our proposal has been adopted by the HTML 5 working group, Internet Explorer 8, Firefox 3, and Safari.

Organization. The remainder of the paper is organized as follows. Section 2 details the threat model for these attacks. Section 3 surveys existing frame navigation policies and converges browsers on a secure policy. Section 4 analyzes two frame communication mechanisms, demonstrates attacks, and proposes defenses. Section 5 describes related work. Section 6 concludes.

2 Threat Model

In this paper, we are concerned with securing in-browser interactions from malicious attackers. We assume an honest user employs a standard web browser to view content from an honest web site. A malicious "web attacker" attempts to disrupt this interaction or steal sensitive information. Typically, a web attacker places malicious content (e.g., JavaScript) in the user's browser and modifies the state of the browser, interfering with the honest session. To study the browser's security policy, which determines the privileges of the attacker's content, we define the web attacker threat model below.

Web Attacker. A *web attacker* is a malicious principal who owns one or more machines on the network. In order to study the security of browsers when rendering malicious content, we assume that the browser gets and renders content from the attacker's web site.

- **Network Abilities.** The web attacker has no special network abilities. In particular, the web attacker can send and receive network messages only from machines under his or her control, possibly acting as a client or server in network protocols of the attacker's choice. Typically, the web attacker uses at least one machine as an HTTP server, which we refer to for simplicity as `attacker.com`. The web attacker can obtain SSL certificates for domains he or she owns; certificate authorities such as `instantssl.com` provide such certificates for free. The web attacker's network abilities are decidedly *weaker* than the usual network attacker considered in studies of network security because the web attacker can neither eavesdrop on messages sent to other recipients nor forge messages from other network locations. For example, a web attacker cannot act as a "man-in-the-middle."

- **Interaction with Client.** We assume the honest user views `attacker.com` in at least one browser window, thereby rendering the attacker's content. We make this assumption because we believe that an honest user's interaction with an honest site should be secure even if the user separately visits a malicious site in a different browser window. We assume the web attacker is constrained by the browser's security policy and does not employ a browser exploit to circumvent the policy. The web attacker's host privileges are decidedly *weaker* than an attacker who can execute a arbitrary code on the user's machine with the user's privileges. For example, a web attacker cannot install or run a system-wide key logger or botnet client.

Attacks accessible to a web attacker have significant practical impact because the attacks can be mounted without any complex or unusual control of the network. In addition, web attacks can be carried out by a standard man-in-the-middle network attacker, provided the user visits a single HTTP site, because a man-in-the-middle can intercept HTTP requests and inject malicious content into the reply, simulating a reply from `attacker.com`.

There are several techniques an attacker can use to drive traffic to `attacker.com`. For example, an attacker can place web advertisements, display popular content indexed by search engines, or send bulk e-mail to attract users. Typically, simply viewing an attacker's advertisement lets the attacker mount a web-based attack. In a previous study [20], we purchased over $50{,}000$ impressions for $\$30$. During each of these impressions, a user's browser rendered our content, giving us the access required to mount a web attack.

We believe that a normal, but careful, web user who reads news and conducts banking, investment, and retail transactions, cannot effectively monitor or restrict the provenience of all content rendered in his or her browser, especially in light of third-party advertisements. In other words, we believe that the web attacker threat model is an accurate representation of normal web behavior, appropriate for security analysis of browser security, and *not* an assumption that users promiscuously visit all possible bad sites in order to tempt fate.

Gadget Attacker. A *gadget attacker* is a web attacker with one additional ability: the integrator embeds a gadget of the attacker's choice. This assumption lets us accurately evaluate mashup isolation and communication protocols because the purpose of these protocols is to let an integrator embed untrusted gadgets safely. In practice, a gadget attacker can either wait for the user to visit the integrator or can redirect the user to the integrator's web site from `attacker.com`.

Out-of-Scope Threats. Although *phishing* [11, 7] can be described informally as a "web attack," the web attacker defined above does not attempt to fool the user by choosing a confusing domain name (such as `bankofthevvest.com`) or using other social engineering. In particular, we do *not* assume that a user treats `attacker.com` as if it were a site other than `attacker.com`. The attacks presented in this paper are "pixel-perfect" in the sense that the browser provides the user no indication whatsoever that an attack is underway. The attacks do not display deceptive images over the browser security indicators nor do they spoof the location bar and or the lock icon. In this paper, we do not consider *cross-site scripting* attacks, in which an attacker exploits a bug in an honest principal's web site to inject malicious content into another security origin. None of the attacks described in this paper rely on the attacker injecting content into another principal's security origin. Instead, we focus on privileges the browser itself affords the attacker to interact with honest sites.

3 Frame Isolation

Netscape Navigator 2.0 introduced the HTML `<frame>` element, which allows web authors to delegate a portion of their document's screen real estate to another document. These frames can be navigated independently of the rest of the main content frame and can, themselves, contain frames, further delegating screen real estate and creating a frame hierarchy. Most modern frames are embedded using the more-flexible `<iframe>` element, introduced in Internet Explorer 3.0. In this paper, we use the term *frame* to refer to both `<frame>` and `<iframe>` elements. The main, or *top-level*, frame of a browser window displays its location in the browser's location bar. Subframes are often indistinguishable from other parts of a page, and the browser does not display their location in its user interface. Browsers decorate a window with a lock icon only if every frame contained in the window was retrieved over HTTPS but do *not* require the frames to be served from the same host. For example, if `https://bank.com/` embeds a frame from `https://attacker.com/`, the browser will decorate the window with a lock icon.

Organization. Section 3.1 reviews browser security policies. Section 3.2 describes cross-window frame navigation attacks and defenses. Section 3.3 details same-window attacks that are not impeded by the cross-window defenses. Section 3.4 analyzes stricter navigation policies and advocates the "descendant policy." Section 3.5 documents our implementation and deployment of the descendant policy in major browsers.

3.1 Background

Scripting Policy. Most web security is focused on the browser's scripting policy, which answers the question "when is script in one frame permitted to manipulate the contents of another frame?" The scripting policy is the most important browser security policy because the ability to script another frame is the ability to control its appearance and behavior completely. For example, if `otherWindow` is another window's frame,

```
var stolenPassword =
    otherWindow.document.forms[0].
    password.value;
```

attempts to steal the user's password in the other window. Modern web browsers permit one frame to read and write all the DOM properties of another frame only when their content was retrieved from the same *origin*, i.e. when the scheme, host, and port number of their locations match. If the content of `otherWindow` was retrieved from a different origin, the browser's security policy will prevent this script from accessing `otherWindow.document`.

Navigation Policy. Every browser must answer the question "when is one frame permitted to navigate another frame?" Prior to 1999, all web browsers implemented a permissive policy:

> *Permissive Policy*
> A frame can navigate any other frame.

For example, if `otherWindow` includes a frame,

```
otherWindow.frames[0].location =
    "https://attacker.com/";
```

navigates the frame to `https://attacker.com/`. This has the effect of replacing the frame's document with content retrieved from that URL. Under the permissive policy, this navigation succeeds even if `otherWindow` contains content from a different security origin. There are a number of other idioms for navigating frames, including

```
window.open("https://attacker.com/",
            "frameName");
```

which requests that the browser search for a frame named `frameName` and navigate the frame to the specified URL. Frame names exist in a global name space and are not restricted to a single security origin.

Top-level Frames. Top-level frames are often exempt from the restrictions imposed by the browser's frame navigation policy. Top-level frames are less vulnerable to frame navigation attacks because the browser displays their location in the location bar. Internet Explorer and Safari do not restrict the navigation of top-level frames at all. Firefox restricts the navigation of top-level frames based on their *openers*, but this restriction can be circumvented [2]. Opera implements a number of restrictions on the navigation of top-level frames based on the current location of the frame.

3.2 Cross-Window Attacks

In 1999, Georgi Guninski discovered that the permissive frame navigation policy admits serious attacks [16]. Guninski discovered that, at the time, the password field on the CitiBank login page was contained within a frame. Because the permissive frame navigation policy lets any frame navigate any other frame, a web attacker can navigate the password frame on CitiBank's page to `https://attacker.com/`, replacing the frame with identical-looking content that sends the user's password to `attacker.com`. In the modern web, this *cross-window attack* might proceed as follows:

1. The user reads a popular blog that displays a Flash advertisement provided by `attacker.com`.
2. The user opens a new window to `bank.com`, which displays its password field in a frame.
3. The malicious advertisement navigates the password frame to `https://attacker.com/`. The location bar still reads `bank.com` and the lock icon is *not* removed.
4. The user enters his or her password, which is then submitted to `attacker.com`.

Of the browsers in heavy use today, Internet Explorer 6 and Safari 3 both implement the permissive policy. Internet Explorer 7 and Firefox 2 implement stricter policies (described in subsequent sections). However, Flash Player can be used to circumvent the stricter navigation policy of Internet Explorer 7, effectively reducing the policy to "permissive." Many web sites are vulnerable to this attack, including Google AdSense, which displays its password field inside a frame; see Figure 1.

Window Policy. In response to Guninski's report, Mozilla implemented a stricter policy in 2001:

> *Window Policy*
> A frame can navigate only frames in its window.

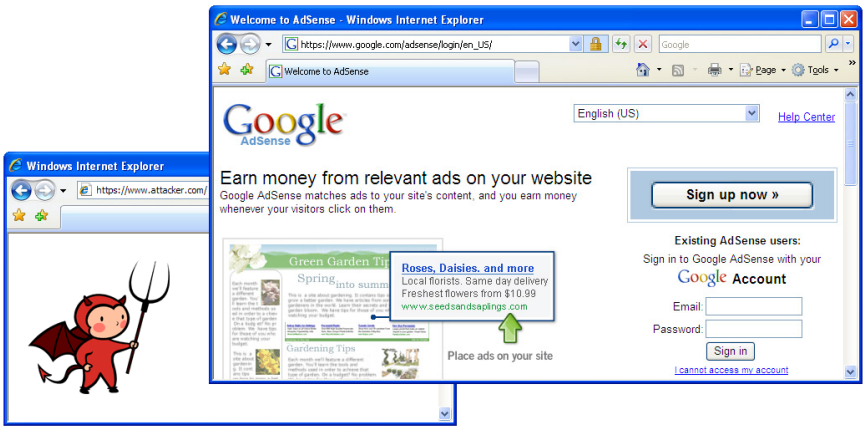


Figure 1: Cross-Window Attack: The attacker controls the password field because it is contained within a frame.

This policy prevents the cross-window attack because the web attacker does not control a frame in the same window as the CitiBank or the Google AdSense login page. Without a foothold in the window, the attacker cannot navigate the login frame to `attacker.com`.

3.3 Same-Window Attacks

The window frame navigation policy is neither universally deployed nor sufficiently strict to protect users on the modern web because mashups violate its implicit security assumption that an honest principal will not embed a frame to a dishonest principal.

Mashups. A *mashup* combines content from multiple sources to create a single user experience. The party combining the content is called the *integrator* and the integrated content is called a *gadget*.

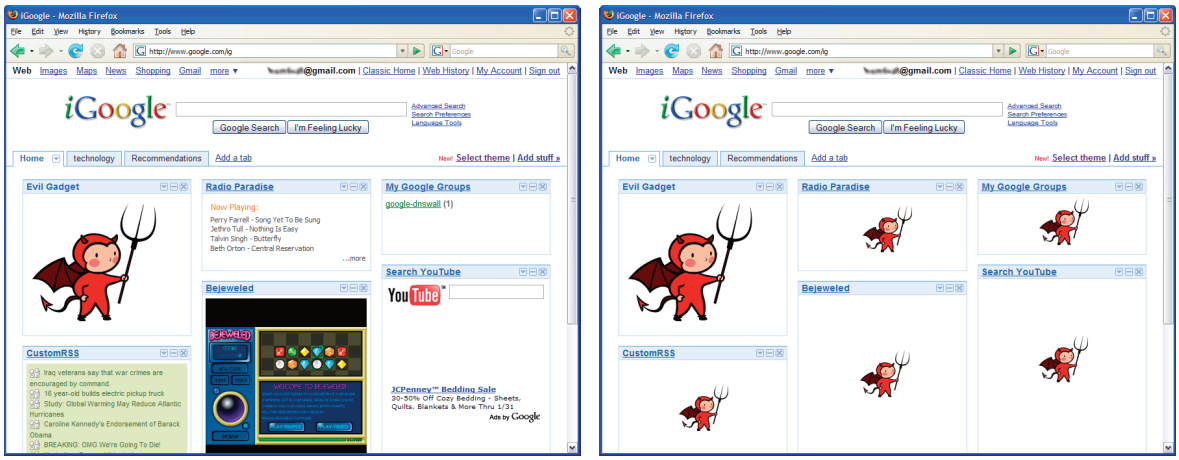
- **Aggregators.** Gadget aggregators, such as iGoogle [15], My Yahoo [40], and Windows Live [28], are one form of mashup. These sites let users customize their experience by selecting gadgets (such as stock tickers, weather predictions, news feeds, etc) to include on their home page. Third parties are encouraged to develop gadgets for the aggregator. These mashups embed the selected gadgets in a frame and rely on the browser's frame isolation to protect users from malicious gadgets.

- **Advertisements.** Web advertising is a simple form of mashup, combining first-party content, such as news articles or sports statistics, with third-party advertisements. Typically, the publisher (the integrator) delegates a portion of its screen real estate to an advertisement network, such as Google, Yahoo, or Microsoft, in exchange for money. Most advertisements, including Google AdWords, are contained in frames, both to prevent the advertisers (who provide the gadgets) from interfering with the publisher's site and to prevent prevent the publisher from using JavaScript to click on the advertisements.

We refer to aggregators and advertisements as *simple mashups* because these mashups do not involve communication between the gadgets and the integrator. Simple mashups rely on the browser to provide isolation but do not require inter-frame communication.

Gadget Hijacking Attacks. Mashups invalidate an implicit assumption of the window policy, that an honest principal will not embed a frame to a dishonest principal. A gadget attacker, however, does control a frame embedded by the honest integrator, giving the attacker the foothold required to mount a *gadget hijacking* attack [22]. In such an attack, a malicious gadget navigates a target gadget to `attacker.com` and impersonates the gadget to the user.

- **Aggregator Vulnerabilities.** iGoogle is vulnerable to gadget hijacking in browsers, such as Firefox 2, that implement the permissive or window policies; see Figure 2. Consider, for example, one popular iGoogle gadget that lets users access their Hotmail inbox. (This gadget is neither provided nor endorsed by Microsoft.) If the user is not logged into Hotmail, the gadget requests the user's Hotmail password. A malicious gadget can replace the Hotmail gadget with content that asks the user for his or her Hotmail password. As in the cross-window attack, the user is unable to distinguish the malicious password field from the honest password field.



(a) Before (b) After

Figure 2: Gadget Hijacking Attack. Under the window policy, the attacker gadget can navigate the other gadgets.

- **Advertisement Vulnerabilities.** Although text advertisements often do not contain active content (e.g., JavaScript), other forms of advertising, such as Flash advertisements, do contain active content. An attacker who provides such an advertisement can steal advertising impressions allotted to other advertisers via gadget hijacking. A malicious advertisement can traverse the page's frame hierarchy and navigate frames containing other advertisements to `attacker.com`, replacing the existing content with the attacker's advertisement.

3.4 Stricter Policies

Although browser vendors do not document their navigation policies, we were able to reverse engineered the navigation policies of existing browsers, and we confirmed our understanding with the browsers' developers. The existing policies are shown in Table 2. In addition to the permissive and window policies described above, we discovered two other frame navigation policies:

Descendant Policy
A frame can navigate only its descendants.

Child Policy
A frame can navigate only its direct children.

The Internet Explorer 6 team wanted to enable the child policy by default, but shipped the permissive policy because the child policy was incompatible with a large number of web sites. The Internet Explorer 7 team designed the descendant policy to balance the security requirement to defeat the cross-window attack with the compatibility requirement to support existing sites [33].

Pixel Delegation. The descendant policy provides the most attractive trade-off between security and compatibility because it is the least restrictive policy that respects pixel delegation. When one frame embeds another frame, the parent frame delegates a region of the screen to the child frame. The browser prevents the child frame from drawing outside of its bounding box but does allow the parent frame to draw over the child using the `position: absolute` style. The descendant policy permits a frame to navigate a target frame precisely when the frame could overwrite the screen real estate of the target frame. Although the child policy is stricter than the descendant policy, the additional strictness does not prevent many additional attacks because a frame can simulate the visual effects of navigating a grandchild frame by drawing over the region of the screen occupied by the grandchild frame. The child policy's added strictness does, however, reduce the policy's compatibility with existing sites, discouraging browser vendors from deploying the child policy.

Origin Propagation. A strict interpretation of the descendant policy prevents a frame from navigating its siblings, even if the frame is from the same security origin as its parent. In this situation, the frame can navigate its sibling indirectly by injecting script into its parent, which can then navigate the sibling because the sibling is a descendant of the parent frame. In general, browsers should decide whether or not to permit a navigation based on the active frame's security origin. Browsers should let an active frame navigate a target frame if there *exists* a frame in the same security origin as the active frame that has the target frame as a descendant. By recognizing this *origin propagation*, browsers can achieve a better trade-off

between security and compatibly. These additional navigations do not sacrifice security because an attacker can perform the navigations indirectly, but allowing them is more convenient for honest web developers.

3.5 Deployment

We collaborated with the HTML 5 working group [18] and browser vendors to deploy the descendant policy in several browsers:

- **Safari.** We implemented the descendant policy as a patch for Safari. Apple accepted our patch and deployed the descendant policy to Mac OS X and Windows Safari users as a security update [30]. Apple also deployed our patch to all iPhone and iPod touch users.
- **Firefox.** We implemented the descendant policy as a patch for Firefox. Before accepting our patch, Mozilla requested tests for all their previous frame navigation regressions. We provided them with approximately 1000 lines of regression tests for their automatic test harness, covering the frame navigation security vulnerabilities from the past ten years. Mozilla accepted our patch and deployed the descendant policy in Firefox 3 [1].
- **Flash.** We reported to Adobe that Flash Player bypasses the descendant policy in Internet Explorer 7. Adobe agreed to ship a patch to all Internet Explorer users in their next security update.
- **Opera.** We notified Opera Software about inconsistencies in Opera's child policy that can be used in gadget hijacking attacks. They plan to fix these vulnerabilities in the upcoming release of Opera 9.5, and are evaluating the compatibility benefits of adopting the descendant policy [35].

4 Frame Communication

Over the past few years, web developers have built sophisticated mashups that, unlike simple aggregators and advertisements, are comprised of gadgets that communicate with each other and with their integrator. Yelp, which integrates the Google Maps gadget, motivates the need for secure inter-frame communication by illustrating how communicating gadgets are used in real deployments. Sections 4.1 and 4.2 analyze and improve fragment-identifier messaging and `postMessage`.

Google Maps. One popular gadget is the Google Maps API [14]. Google provides two mechanisms for integrating Google Maps:

- **Frame.** In the frame version of the gadget, the integrator embeds a frame to `maps.google.com`, which Google fills with a map centered at the specified location. The user can interact with map, but the integrator is oblivious to this interaction and cannot interact with the map directly.
- **Script.** In the script version of the gadget, the integrator embeds a `<script>` tag that executes JavaScript from `maps.google.com`. This script creates a rich JavaScript API the integrator can use to interact with the map, but the script runs with all of the integrator's privileges.

Yelp. Yelp is a popular review web site that uses the Google Maps gadget to display the locations of restaurants and other businesses it reviews. Yelp requires a high degree of interactivity with the Maps gadget because it places markers on the map for each restaurant and displays the restaurant's review when the user clicks on the marker. In order to deliver these advanced features, Yelp must use the script version of the Maps gadget. This design requires Yelp to trust Google Maps completely because Google's script runs with Yelp's privileges in the user's browser, granting Google the ability to manipulate Yelp's reviews and steal Yelp's customer's information. Although Google might be trustworthy, the script approach does not scale beyond highly respected gadget providers. Secure inter-frame communication provides the best of both alternatives: Yelp (and similar sites) can realize the interactivity of the script version of Google Maps gadget while maintaining the security of the frame version of the gadget.

4.1 The Fragment Identifier Channel

Although the browser's scripting policy isolates frames from different security origins, clever mashup designers have discovered an unintended channel between frames: the *fragment identifier channel* [3, 36]. This channel is regulated by the browser's less-restrictive frame navigation policy. This "found" technology lets mashup developers place each gadget in a separate frame and rely on the browser's security policy to prevent malicious gadgets from attacking the integrator and honest gadgets.

Mechanism. Normally, when a frame is navigated to a new URL, the browser retrieves the URL from the network and replaces the frame's document with the retrieved content. However, if the new URL different from the old URL only in the fragment (the portion after the `#`), then the browser does not reload the frame. If `frames[0]` is currently located at `http://example.com/doc`,

IE 6 (default)	IE 6 (optional)	IE 7 (default)	IE 7 (optional)	Firefox 2	Safari 3	Opera 9
Permissive	Child	Descendant	Permissive	Window	Permissive	Child

Table 2: Frame navigation policies deployed in existing browsers.

```
frames[0].location =
    "http://example.com/doc#message";
```

changes the frame's location without reloading the frame or destroying its JavaScript context. The frame can observe the value of the fragment by periodically polling `window.location.hash` to see if the fragment identifier has changed. This technique can be used to send short string messages entirely within the browser, avoiding network latency. However, the communication channel is somewhat unreliable because, if two navigations occur between polls, the first message will be lost.

Security Properties. Because it was "found" and not designed, the fragment identifier channel has less-than-ideal security properties. The browser's scripting policy prevents security origins other than the one preceding the # from eavesdropping on messages because they are unable to *read* the frame's location (even though the navigation policy permits them to *write* to the frame's location). Browsers also prevent arbitrary security origins from tampering with portions of messages. Other security origins can, however, overwrite the fragment identifier in its entirety, leaving the recipient to guess the sender of each message.

To understand these security properties, we develop an analogy with well-known properties of network channels. We view the browser as guaranteeing that the fragment identifier channel has *confidentiality*: a message can be read only by its intended recipient. The fragment identifier channel fails to be a secure channel because it lacks *authentication*, the ability of the recipient to unambiguously determine the sender of a message. The channel also fails to be *reliable* because messages might not be delivered, and the attacker might be able to replay previous messages using the browser's `history` API.

The security properties of the fragment identifier channel are analogous to a channel on an untrusted network secured by a public-key cryptosystem in which each message is encrypted with the public key of its intended recipient. In both cases, if Alice sends a message to Bob, no one except Bob learns the contents of the message (unless Bob forwards the message). In both settings, the channel does *not* provide a reliable procedure for determining who sent a given message. There are two interesting differences between the fragment identifier channel and the public-key channel:

1. The public-key channel is susceptible to traffic analysis, but an attacker cannot determine the length of a message sent over the fragment identifier channel. An attacker can extract timing information by frequently polling the browser's clock, but obtaining a high-resolution timing signal significantly degrades the browser's performance.

2. The fragment identifier channel is constrained by the browser's frame navigation policy. In principle, this could be used to construct protocols secure for the fragment identifier channel that are insecure for the public-key channel (by preventing the attacker from navigating the recipient), but in practice this restriction has not prevented us from constructing attacks on existing protocol implementations.

Despite these differences, we find the network analogy useful in analyzing inter-frame communication.

Windows Live Channels. Microsoft uses the fragment identifier channel in its Windows Live platform library to implement a higher-level channel API, `Microsoft.Live.Channels` [36]. The Windows Live Contacts gadget uses this API to communicate with its integrator. The integrator can instruct the gadget to add or remove contacts from the user's contacts list, and the gadget can send the integrator details about the user's contacts. Whenever the integrator asks the gadget to perform a sensitive action, the gadget asks the user to confirm the operation and displays the integrator's host name to aid the user in making trust decisions.

`Microsoft.Live.Channels` attempts to build a secure channel over the fragment identifier channel. By reverse engineering the implementation, we determined that it uses two sessions of the following protocol (one in each direction) to establish a secure channel:

$$A \rightarrow B : N_A, \text{URI}_A$$
$$B \rightarrow A : N_A, N_B$$
$$A \rightarrow B : N_B, \text{Message}_1$$

In this notation, A and B are frames, N_A and N_B are fresh nonces (numbers chosen at random during each run of the protocol), and URI_A is the location of A's frame. Under the network analogy described above, this protocol is analogous to a variant of the classic Needham-Schroeder key-establishment protocol [29].

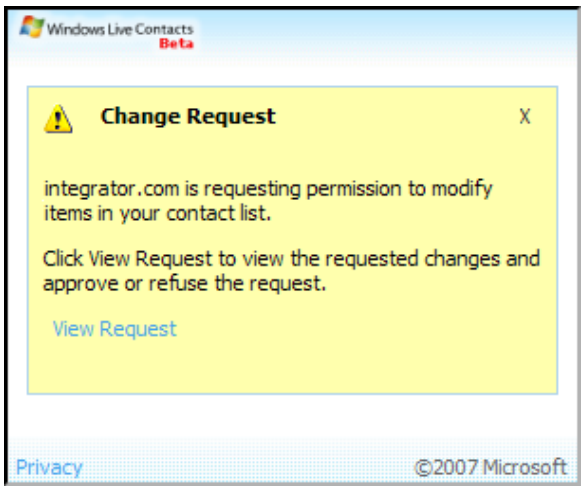


Figure 3: Lowe Anomaly: This Windows Live Contacts gadget received a message that appeared to come from `integrator.com`, but in reality the request was made by `attacker.com`.

The Needham-Schroeder protocol was designed to establish a shared secret between two parties over an insecure channel. In the Needham-Schroeder protocol, each message is encrypted with the public key of its intended recipient. The Windows Live protocol does not employ encryption because the fragment identifier channel already provides the required confidentiality.

The Needham-Schroeder protocol has a well-known anomaly, due to Lowe [23], which leads to an attack in the browser setting. In the Lowe scenario, an honest principal, Alice, initiates the protocol with a dishonest party, Eve. Eve then convinces honest Bob that she is Alice. In order to exploit the Lowe anomaly, an honest principal must be willing to initiate the protocol with a dishonest principal. This requirement is met in mashups because the integrator initiates the protocol with the gadget attacker's gadget in order to establish a channel. The Lowe anomaly can be exploited to impersonate the integrator to the Windows Live Contacts gadget as follows:

$$\begin{aligned}
\text{Integrator} &\rightarrow \text{Attacker} : N_I, \text{URI}_I \\
\text{Attacker} &\rightarrow \text{Gadget} : N_I, \text{URI}_I \\
\text{Gadget} &\rightarrow \text{Integrator} : N_I, N_G \\
\text{Integrator} &\rightarrow \text{Attacker} : N_G, \text{Message}_1
\end{aligned}$$

After these four messages, the attacker possesses N_I and N_G and can impersonate the integrator to the gadget. We have successfully implemented this attack against the Windows Live Contacts gadget. The issue is easily observable for the Contacts gadget because the gadget displays the integrator's host name to the user in its security user interface; see Figure 3.

SMash and OpenAjax 1.1. A recent paper [22] from IBM proposed another protocol for establishing a secure channel over the fragment identifier channel. They describe their protocol as follows:

> The SMash library in the mashup application creates the secret, an unguessable random value. When creating the component, it includes the secret in the fragment of the component URL. When the component creates the tunnel iframe it passes the secret in the same manner.

The SMash developers have contributed their code to the OpenAjax project, which plans to include their fragment identifier protocol in version 1.1. The SMash protocol can be understood as follows:

$$\begin{aligned}
A &\rightarrow B : N, \text{URI}_A \\
B &\rightarrow A : N \\
A &\rightarrow B : N, \text{Message}_1
\end{aligned}$$

This protocol admits the following simple attack:

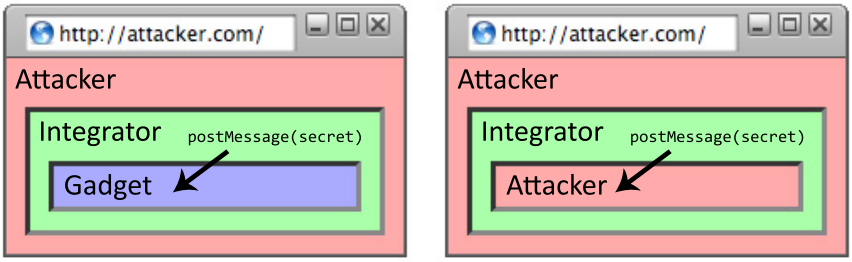
$$\begin{aligned}
\text{Attacker} &\rightarrow \text{Gadget} : N, \text{URI}_I \\
\text{Gadget} &\rightarrow \text{Integrator} : N \\
\text{Attacker} &\rightarrow \text{Gadget} : N, \text{Message}
\end{aligned}$$

We have confirmed this attack by implementing the attack against the SMash implementation. Additionally, the attacker is able to conduct this attack covertly by blocking the message from the gadget to the integrator because the message waits for the `load` event to fire.

Secure Fragment Messaging. The fragment identifier channel can be secured using a variant of the Needham-Schroeder-Lowe protocol [23]. The main idea in Lowe's improvement of the Needham-Schroeder protocol is that the responder must include his identity in the second message of the protocol, letting the honest initiator determine that an attack is in progress and abort the protocol.

$$\begin{aligned}
A &\rightarrow B : N_A, \text{URI}_A \\
B &\rightarrow A : N_A, N_B, \text{URI}_B \\
A &\rightarrow B : N_B \\
&\cdots \\
A &\rightarrow B : N_A, N_B, \text{Message}_i \\
B &\rightarrow A : N_A, N_B, \text{Message}_j
\end{aligned}$$

We contacted Microsoft, IBM, and the OpenAJAX Alliance about the vulnerabilities in their fragment identifier messaging protocols and suggested the above protocol improvement. Microsoft adopted our suggestions and deployed a patched version of



(a) Integrator sends secret messages to child

(b) Attacker hijacks integrator's child

Figure 4: Recursive Mashup Attack

`Microsoft.Live.Channels` and of the Windows Live Contacts gadget. IBM adopted our suggestions and revised their SMash paper. The OpenAJAX Alliance adopted our suggestions and updated their codebase. All three now use the above protocol to establish a secure channel using fragment identifiers.

4.2 The postMessage Channel

HTML 5 [19] specifies a new browser API for asynchronous communication between frames. Unlike the fragment identifier channel, the `postMessage` channel was designed for cross-site communication. The `postMessage` API was originally implemented in Opera 8 and is now supported by Internet Explorer 8, Firefox 3 [37], and Safari [24].

Mechanism. To send a message to another frame, the sender calls the `postMessage` method:

```
frames[0].postMessage("Hello world.");
```

The browser then generates a `message` event in the recipient's frame that contains the message, the origin (scheme, port, and domain) of the sender, and a JavaScript pointer to the frame that sent the message.

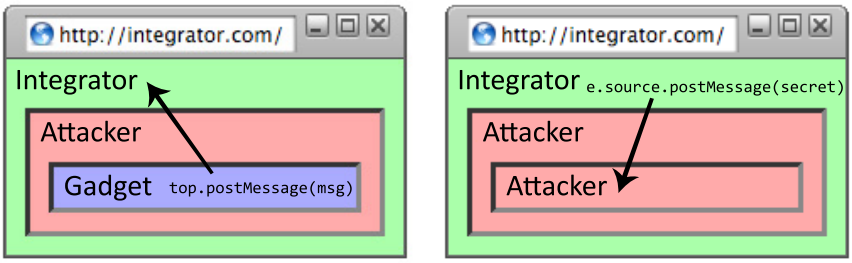
Security Properties. The `postMessage` channel guarantees *authentication*, messages accurately identify their senders, but the channel lacks confidentiality. Thus, `postMessage` has almost the "opposite" security properties as the fragment identifier channel. Where the fragment identifier channel has confidentiality without authentication, the `postMessage` channel has authentication without confidentiality. The security properties of the `postMessage` channel are analogous to a channel on a untrusted network secured by an existentially unforgeable signature scheme. In both cases, if Alice sends a message to Bob, Bob can determine unambiguously that Alice sent the message. With `postMessage`, the `origin` property accurately identifies the sender; with cryptographic signatures, verifying the signature on a message accurately identifies the signer of the message. One difference between the channels is that cryptographic signatures can be easily replayed, but the `postMessage` channel is resistant to replay attacks. In some cases, however, an attacker might be able to mount a replay attack by reloading honest frames.

Attacks. Although `postMessage` is widely believed to provide a secure channel between frames, we show an attack on the confidentiality of the channel. A message sent with `postMessage` is directed at a frame, but if the attacker navigates that frame to `attacker.com` before the `message` event is generated, the attacker will receive the message instead of the intended recipient.

- **Recursive Mashup Attack.** Suppose, for example, that an integrator embeds a frame to a gadget and then calls `postMessage` on that frame. The attacker can load the integrator inside a frame and carry out an attack without violating the descendant frame navigation policy. After the attacker loads the integrator inside a frame, the attacker navigates the gadget frame to `attacker.com`. Then, when the integrator calls `postMessage` on the "gadget's" frame, the browser delivers the message to the attacker whose content now occupies the "gadget's" frame; see Figure 4. The integrator can prevent this attack by "frame busting," i.e., by refusing to render the mashup if `top !== self`, indicating that the integrator is contained in a frame.

- **Reply Attack.** Another `postMessage` idiom is also vulnerable to interception, even under the child frame navigation policy:

```
window.onmessage = function(e) {
  if (e.origin == "https://b.com")
    e.source.postMessage(secret);
};
```



(a) Gadget requests secret from integrator

(b) Integrator's reply is delivered to attacker

Figure 5: Reply Attack

The `source` attribute of the `MessageEvent` is a JavaScript reference to the frame that sent the message. It is tempting to conclude that the reply will be sent to `https://b.com`. However, an attacker might be able to intercept the message. Suppose that the honest gadget calls `top.postMessage("Hello")`. The gadget attacker can intercept the message by embedding the honest gadget in a frame, as depicted in Figure 5. After the gadget posts its message to the integrator, the attacker navigates the honest gadget to `https://attacker.com`. (This navigation is permitted under both the child and descendant frame navigation policies.) When the integrator replies to the `source` of the message, the message will be delivered to the attacker instead of to the honest gadget.

Securing postMessage. It might be feasible for sites to build a secure channel using `postMessage` as an underlying communication primitive, but we would prefer that `postMessage` provide a secure channel natively. In MashupOS [39], we proposed a new browser API, `CommRequest`, to send messages between origins. When sending a message using `CommRequest`, the sender addresses the message to a principal:

```
var req = new CommRequest();
req.open("INVOKE",
         "local:https://b.com//inc");
req.send("Hello");
```

Using this interface, `CommRequest` protects the confidentiality of messages because the `CommServer` will deliver messages only to the specified principal. Although `CommRequest` provides adequate security, the `postMessage` API is further along in the standardization and deployment process. We therefore propose extending the `postMessage` API to provide the additional security benefits of `CommRequest` by including a second parameter: the origin of the intended recipient. If the sender specifies a target origin, the browser will deliver the message to the targeted frame *only if* that frame's current security origin matches the argument. The browser is free to deliver the message to any principal if the sender specifies a target origin of `*`. Using this improved API, a frame can reply to a message using the following idiom:

```
window.onmessage = function(e) {
  if (e.origin == "https://b.com")
    e.source.postMessage(secret,
                         e.origin);
};
```

As shown in this example use, the API uses the same origin syntax for both sending and receiving messages. The scheme is included in the origin for those developers who wish to defend against active network attackers by distinguishing between HTTP and HTTPS. We implemented this API change as a patch for Safari and a patch for Firefox. Our proposal was accepted by the HTML 5 working group [17]. The new API is now included in Firefox 3 [38], Safari [32], and Internet Explorer 8 [25].

5 Related Work

Mitigations for Gadget Hijacking. SMash [22] mitigates gadget hijacking (which the authors refer to as "frame phishing") without modifying the browser by carefully monitoring the frame hierarchy and browser events for evidence of unexpected navigation. Neither the integrator nor the gadget can prevent these navigations, but the mashup can alert the user and refuse to function if it detects an illicit navigation. This approach lets an attacker mount a denial-of-service attack against the mashup, but a web attacker can already mount a denial-of-service attack against the entire browser by issuing a blocking `XMLHttpRequest` or entering an infinite loop.

Unfortunately, this approach can lead to false positives. SMash waits 20 seconds for a gadget to load before assuming that the gadget has been hijacked and warning the user. An attacker might be able to fool the user into entering sensitive information during this time interval. Using a shorter time interval might cause users with slow network connections to receive warnings even though no attack is in progress. We expect that the deployment of the descendant policy will obviate the need for server-enforced gadget hijacking mitigations.

Safe Subsets of HTML and JavaScript. One way to sidestep the security issues of frame-based mashups is to avoid using frames entirely and render the gadgets together with the integrator in a single document. This approach forgoes the protections of the browser's security policy because all the gadgets and the integrator share a single browser security context. To maintain security, this approach requires gadgets to be written in a "safe subset" of HTML and JavaScript that prevents a malicious gadget from attacking the integrator or other gadgets. Analyzing the security and usability of these subsets is an active area of research. Several open-source [13, 4] and closed-source [31, 10] implementations are available. FBML [10] is currently the most successful of these subsets and is used by millions of users as the foundation of the Facebook Platform.

Writing programs in one of these safe subsets is often awkward because the language is highly constrained to avoid potentially dangerous features. To improve usability, the safe subsets are often accompanied by a compiler that transforms untrusted HTML and JavaScript into the subset, possibly at the cost of performance. These safe subsets will become easier to use over time as these compilers become more sophisticated and more libraries become available, but with the deployment of `postMessage` and the descendant policy, we expect that frame-based mashup designs will continue to find wide use as well.

Other Frame Isolation Proposals. There are several other proposals for frame isolation and communication:

- **Subspace.** In Subspace [21], we used a multi-level hierarchy of frames that coordinated their `document.domain` property to communicate directly in JavaScript. Similar to most frame-based mashups, the descendant frame navigation policy is required to prevent gadget hijacking.

- **Module Tag.** The proposed `<module>` tag [5] is similar to an `<iframe>` tag, but the module runs in an unprivileged security context, without a principal, and the browser prevents the integrator from overlaying content on top of the module. Unlike `postMessage`, the communication primitive used with the module tag is intentionally unauthenticated: it does not identify the sender of a message. It is unknown whether navigation can be used to intercept messages as there are no implementations of the `<module>` tag.

- **Security=Restricted and Jail.** Internet Explorer supports a `security` attribute [26] of frames that can be set to `restricted`. With `security="restricted"`, the frame's content cannot run JavaScript. Similarly, the proposed `<jail>` tag [8] encloses untrusted content and prevents the sandboxed content from running JavaScript. However, eliminating JavaScript prevents gadgets from offering interactive experiences.

- **MashupOS.** Our MashupOS proposal [39] includes new primitives for isolating web content while allowing secure communication. Our improvements to `postMessage` and frame navigation policies allow web authors to obtain some of the benefits of MashupOS using existing web APIs.

6 Conclusions

Web browsers provide a platform for web applications. These applications rely on the browser to isolate frames from different security origins and to provide secure inter-frame communication. To provide isolation, browsers implement a number of security policies, including a frame navigation policy. The original frame navigation policy, the permissive policy, admits a number of attacks. The modern frame navigation policy, the descendant policy, prevents these attacks by permitting one frame to navigate another only if the frame could draw over the other frame's region of the screen. The descendant policy provides an attractive trade-off between security and compatibility, is deployed in the major browsers, and has been standardized in HTML 5.

In existing browsers, frame navigation can be used as an inter-frame communication channel with a technique known as fragment identifier messaging. If used directly, the fragment identifier channel lacks authentication. To provide authentication, `Windows.Live.Channels`, SMash, and OpenAjax 1.1 use messaging protocols. These protocols are vulnerable to attacks on authentication but can be repaired in a manner analogous to Lowe's variation of the Needham-Schroeder protocol [23].

The `postMessage` communication channel suffered the converse security vulnerability: using frame navigation, an attacker can breach the confidentiality of the channel. We propose providing confidentiality by extending the `postMessage` API to let the sender specify

an intended recipient. Our proposal was adopted by the HTML 5 working group, Internet Explorer 8, Firefox 3, and Safari.

With these improvements to the browser's isolation and communication primitives, frames are a more attractive feature for integrating third-party web content. Two challenges remain for mashups incorporating untrusted content. First, a gadget is permitted to navigate the top-level frame and can redirect the user from the mashup to a site of the attacker's choice. This navigation is made evident by the browser's location bar, but many users ignore the location bar. Improving the usability of the browser's security user interface is an important area of future work. Second, a gadget can subvert the browser's security mechanisms if the attacker employs a browser exploit to execute arbitrary code. A browser design that provides further isolation against this threat is another important area of future work.

Acknowledgments

We thank Mike Beltzner, Sumeer Bhola, Dan Boneh, Gabriel E. Corvera, Ian Hickson, Koji Kato, Eric Lawrence, Erick Lee, David Ross, Maciej Stachowiak, Hallvord Steen, Peleus Uhley, Jeff Walden, Sam Weinig, and Boris Zbarsky for their helpful suggestions and feedback. This work is supported by grants from the National Science Foundation and the US Department of Homeland Security.

References

[1] Adam Barth et al. Adopt "descendant" frame navigation policy to prevent frame hijacking. `https://bugzilla.mozilla.org/show_bug.cgi?id=408052`.

[2] Adam Barth and Collin Jackson. Protecting browsers from frame hijacking attacks, December 2007. `http://crypto.stanford.edu/frames/`.

[3] James Burke. Cross domain frame communication with fragment identifiers. `http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html`.

[4] Douglas Crockford. ADsafe: Making JavaScript safe for advertising. `http://adsafe.org/`.

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

[6] Neil Daswani, Micheal Stoppelman, et al. The anatomy of Clickbot.A. In *Proc. HotBots*, 2007.

[7] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on human factors in computing systems*, 2006.

[8] Brendan Eich. JavaScript: Mobility and ubiquity. `http://kathrin.dagstuhl.de/files/Materials/07/07091/07091.EichBrendan.Slides.pdf`.

[9] Facebook. Badges. `http://www.facebook.com/help.php?page=4`.

[10] Facebook. Facebook Markup Language (FBML). `http://wiki.developers.facebook.com/index.php/FBML`.

[11] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An Internet con game. In *Proceedings of the 20th National Information Systems Security Conference*, 1996.

[12] Flickr API. `http://flickr.com/services/api/`.

[13] Google. Caja: A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`.

[14] Google. Google Maps API. `http://code.google.com/apis/maps/`.

[15] iGoogle. `http://www.google.com/ig`.

[16] Georgi Guninski. Frame spoofing using loading two frames. `https://bugzilla.mozilla.org/show_bug.cgi?id=13871`.

[17] Ian Hickson. Re: A potential slight security enhancement to postMessage, Februrary 2008. `http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-February/013949.html`.

[18] Ian Hickson. Re: HTML5 frame navigation policy, April 2008. `http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-April/014597.html`.

[19] Ian Hickson et al. HTML 5 Working Draft. `http://www.whatwg.org/specs/web-apps/current-work/`.

[20] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

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

[22] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure cross-domain mashups on unmodified browsers. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, 2008.

[23] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055. Springer Verlag, 1996.

[24] Henry Mason. No support for MessageEvent interface, 2007. `https://bugs.webkit.org/show_bug.cgi?id=14994`.

[25] Microsoft. postMessage method. `http://msdn.microsoft.com/en-us/library/cc197015(VS.85).aspx`.

[26] Microsoft. SECURITY attribute. `http://msdn2.microsoft.com/en-us/library/ms534622(VS.85).aspx`.

[27] Microsoft. Try the Windows Live Contacts control. `http://dev.live.com/mashups/trypresencecontrol/`.

[28] Microsoft. Windows Live. `http://home.live.com/`.

[29] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[30] National Institute of Standards and Technology. CVE-2007-5858, December 2007.

[31] Charlie Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[32] Adam Roben et al. Change postMessage/MessageEvent to match HTML5 wrt. exposing origin vs. domain/uri. `https://bugs.webkit.org/show_bug.cgi?id=17331`.

[33] David Ross, 2008. Personal communication.

[34] J. Ruderman. JavaScript Security: Same Origin. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[35] Hallvord Steen, 2008. Personal communication.

[36] Danny Thorpe. Secure cross-domain communication in the browser. *The Architecture Journal*, 12:14–18, July 2007. `http://msdn2.microsoft.com/en-us/library/bb735305.aspx`.

[37] Jeff Walden. Implement HTML5's cross-document messaging API (postMessage), 2007–2008. `https://bugzilla.mozilla.org/show_bug.cgi?id=387706`.

[38] Jeff Walden et al. Update postMessage and MessageEvent to reflect domain/uri being replaced by origin, optional origin argument. `https://bugzilla.mozilla.org/show_bug.cgi?id=417075`.

[39] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[40] Yahoo! My Yahoo! `http://my.yahoo.com/`.