DOI:10.1145/1516046.1516066

Securing Frame Communication in Browsers

By Adam Barth, Collin Jackson, and John C. Mitchell

Abstract

Many Web sites embed third-party content in frames, relying on the browser's security policy to protect against malicious content. However, frames provide insufficient isolation in browsers that let framed content navigate other frames. 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 affects communication between cooperating frames. We therefore analyze two techniques for interframe communication between isolated frames. The first method, fragment identifier messaging, initially provides confidentiality without authentication, which we repair using concepts from a well-known network protocol. The second method, `postMessage`, initially provides authentication, but we discover an attack that breaches confidentiality. We propose improvements in the `postMessage` API to provide confidentiality; our proposal has been 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.[3] Other common aggregations of third-party content include Flickr albums, Facebook badges, and personalized home pages offered by the three major Web portals (iGoogle, My Yahoo! and Windows Live). More advanced uses of third-party components include Yelp's use of Google Maps to display restaurant locations, and the Windows Live Contacts gadget. 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 provide isolation. In more sophisticated mashups, the integrator does wish to communicate and requires secure interframe communication. When a site wishes to provide isolation and communication between content on its pages, the site inevitably relies on the browser rendering process and isolation policy, because Web content is rendered and viewed under browser control.

In this paper, we study a contemporary Web version of a recurring problem in computer systems: isolating untrusted, or partially trusted, components while providing secure intercomponent 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. Although the browser's well-known "same-origin policy"[19] restricts script running in one frame from manipulating content in another frame, browsers use a different policy to determine whether one frame is allowed to navigate (change the location of) another. Although browsers must restrict navigation to provide isolation, navigation is the basis of one form of interframe communication used by leading companies and navigation can be used to attack a second interframe communication mechanism.

Many recent browsers have overly permissive frame navigation policies that lead to a variety of attacks. To prevent attacks, we demonstrate against the Google AdSense login page and the iGoogle gadget aggregator, we propose tightening the browser's frame navigation policy. Based on a comparison of four policies, we advocate a specific policy that restricts navigation while maintaining compatibility with existing Web content. We have collaborated with the HTML 5 working group to standardize this policy and with browser vendors to deploy this policy in Firefox 3, Safari 3.1, and Google Chrome. Because the policy is already implemented in Internet Explorer 7, our preferred policy is now standardized and deployed in the four most-used browsers.

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 interframe communication: fragment identifier messaging and `postMessage`. Table 1 summarizes our results.

- Fragment identifier messaging uses frame navigation to send messages between frames. This channel lacks an important security property: messages are confidential but senders are not authenticated. These properties are analogous to a network channel in which senders encrypt their messages with the recipient's public key. `The Microsoft.Live.Channels` library uses fragment identifier messaging to let the Windows Live Contacts gadget communicate with its integrator, following an authentication protocol analogous to the Needham–Schroeder public-key protocol.[17]

The original version of this paper was published in the *Proceedings of the 17th USENIX Security Symposium*, July 2008.

Table 1: Security properties of frame communication channels.

	Confidentiality	Authentication	Network Analogue
Fragment identifier messaging	✓		Public Key Encryption
Original postMessage		✓	Public Key Signatures
Improved postMessage	✓	✓	SSL/TLS

We discover an attack on this protocol, related to Lowe's anomaly in the Needham–Schroeder protocol,[15] 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[15] that Microsoft implemented and deployed.

- `postMessage` is a browser API designed for interframe communication[10] that is implemented in Internet Explorer 8, Firefox 3, Safari 4, Google Chrome, and Opera. Although `postMessage` has been deployed in Opera 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 modifying the API. Our proposal has been adopted by the HTML 5 working group and all the major browsers.

The remainder of the paper is organized as follows. Section 2 details our threat models. Section 3 surveys existing frame navigation policies and standardizes 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 section, we define precise threat models so that we can determine how effectively browser mechanisms defend against specific classes of attacks. We consider two kinds of attackers, a "Web attacker" and a slightly more powerful "gadget attacker." Although *phishing*[4, 6] can be described informally as a Web attack, we do not assume that either the Web attacker or the gadget attacker can fool the user by using a confusing domain name (such as `bankofthevvest.com`) or by other social engineering. Instead, we assume the user uses every browser security feature, including the location bar and lock icon, accurately and correctly.

2.1. Web attacker

A *Web attacker* is a malicious principal who owns one or more machines on the network. To study the browser security policy, we assume that the user's browser 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 as `attacker.com`. The Web attacker has HTTPS certificates for domains he or she owns; certificate authorities provide such certificates for free. The Web attacker's network abilities are decidedly *weaker* than the usual network attacker considered in network security because the Web attacker can neither eavesdrop on messages to nor forge messages from other network locations. For example, a Web attacker cannot be a network "man-in-the-middle."
- **Client Abilities:** We assume that the user views `attacker.com` in a popular browser, rendering the attacker's content. We make this assumption because an honest user's interaction with an honest site should be secure even if the user visits a malicious site in another browser window. The Web attacker's content is subject to the browser's security policy, making the Web attacker decidedly *weaker* than an attacker who can execute an arbitrary code with the user's privileges. For example, a Web attacker cannot install a system-wide key logger or botnet client.

We do not assume that the user treats `attacker.com` as a site other than `attacker.com`. For example, the user never gives a `bank.com` password to `attacker.com`. We also assume that honest sites are free of cross-site scripting vulnerabilities.[20] In fact, none of the attacks described in this paper rely on running malicious JavaScript as an honest principal. Instead, we focus on privileges the browser itself affords the attacker to interact with honest sites.

In addition to our interest in protecting users that visit malicious sites, our assumption that the user visits `attacker.com` is further supported by several techniques for attracting users. For example, an attacker can place Web advertisements, host popular content with organic appeal, or send bulk e-mail encouraging visitors. Typically, simply *viewing* an attacker's advertisement (such as on a search page) lets the attacker mount a Web attack. In a previous study,[12] 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.

Attacks accessible to a Web attacker have significant practical impact because these attacks do not require unusual control of the network. Web attacks can also be carried out by a standard man-in-the-middle network attacker, once the user visits a single HTTP site, because a man-in-the-middle can inject malicious content into the HTTP response, simulating a reply from `attacker.com`.

2.2. 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`.

3. FRAME ISOLATION

Web sites can use frames to delegate portions of their screen real estate to other Web sites. For example, a site can sell parts of their pages to adverting networks. The browser displays the location of the main, or *top-level*, frame in its location bar. Subframes are often visually indistinguishable from other parts of a page, and the browser does not display their location in its user interface.

3.1. Background

The browser's scripting policy answers the question "when can one frame manipulate the contents of another frame?" The scripting policy is the most important part of the browser security policy because a frame can act on behalf of every other frame it can script. For example,

```
otherWindow.document.forms[0].password.value
```

attempts to read the user's password from another window. Modern Web browsers let one frame read and write all the properties of another frame only when their content was retrieved from the same *origin*, i.e. when the scheme (e.g., `http` or `https`), host, and port of their locations match. If the content of `otherWindow` was retrieved from a different origin, the browser's security policy will prevent the script above from accessing `otherWindow.document`.

In addition to enforcing the scripting 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/`. Under the permissive policy, the browser navigates `otherWindow` even if it contains content from another origin. There are a number of idioms for navigating frames, including

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

which navigates a frame named `frameName`. Frame names exist in a global name space that is shared across origins.

3.2. Cross-window attacks

In 1999, Georgi Guninski discovered that the permissive frame navigation policy admits serious attacks.[7] At the time, the password field on the CitiBank login page was contained within a frame, and the Web attacker could navigate that frame to `https://attacker.com/`, letting the attacker fill the frame with identical-looking content that steals the password. This *cross-window attack* proceeds as follows:

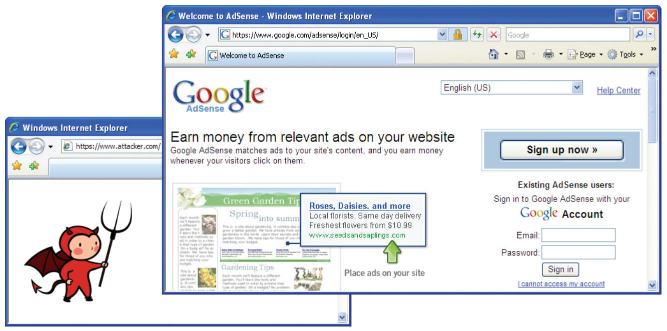
1. The user views a blog that displays the attacker's ad.
2. Separately, the user visits `bank.com`, which displays its password field in a frame.
3. The advertisement navigates the password frame to `https://attacker.com/`. The location bar remains `https://bank.com` and the lock icon remains present.
4. The user enters his or her `bank.com` password into the `https://attacker.com/` frame on the `bank.com` page, submitting the password to `attacker.com`.

Of the browsers in heavy use today, Internet Explorer 6 and Safari 3 both implement the permissive policy and allow this attack. Internet Explorer 7 and Firefox 2 implement stricter policies (described in subsequent sections). Many Web sites, including Google AdSense, display their password field in a frame and are vulnerable to this attack; see Figure 1.

3.3. Same-window attacks

In 2001, Mozilla prevented the cross-window attack by implementing a stricter policy:

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



Earn money from relevant ads on your website
Google AdSense matches ads to your site's content, and you earn money whenever your visitors click on them.

Roses, Daisies, and more
Local florists. Same day delivery
Freshest flowers from $10.99
www.seedsandsaplings.com

Existing AdSense users:
Sign in to Google AdSense with your Google Account

Figure 1: Cross-window attack. The attacker hijacks the password field, which is in a frame.

This policy prevents the cross-window attack because the Web attacker does not control a frame in a trusted window and, without a foothold in the window, the attacker cannot navigate the login frame. However, the window policy is insufficiently strict to protect users because the gadget attacker does have a foothold in a trusted window in a mashup. (Recall that, in a mashup, the integrator combines gadgets from different sources into a single experience.)

- **Aggregators:** Gadget aggregators, such as iGoogle, My Yahoo! and Windows Live, provide one form of mashup. These sites let users customize their experience by including gadgets (such as stock tickers, weather predictions, and news feeds) on their home page. These sites put third-party gadgets in frames and rely on the browser to protect users from malicious gadgets.
- **Advertisements:** Web advertising produces mashups that combine first-party content, such as news articles or sports statistics, with third-party advertisements. 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 the publisher from using JavaScript to click on the advertisements.

We refer to pages with advertisements as *simple mashups* because the integrator and the gadgets do not communicate. Simple mashups rely on the browser to provide isolation but do not require interframe communication.

The windows policy offers no protection for mashups because the integrator's window contains untrusted gadgets. A gadget attacker who supplies a malicious gadget does control a frame in the honest integrator's window, giving the attacker the foothold required to mount a *gadget hijacking* attack.[14] A malicious gadget can navigate a target gadget to `attacker.com` and impersonates the gadget to the user. For example, iGoogle is vulnerable to gadget hijacking in browsers, such as Firefox 2, that implement the permissive or window policies; see Figure 2. Consider an iGoogle gadget that lets users access their Hotmail account. 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 and steal the user's Hotmail password. As in the cross-window attack, the user is unable to distinguish the malicious password field from the honest password field.

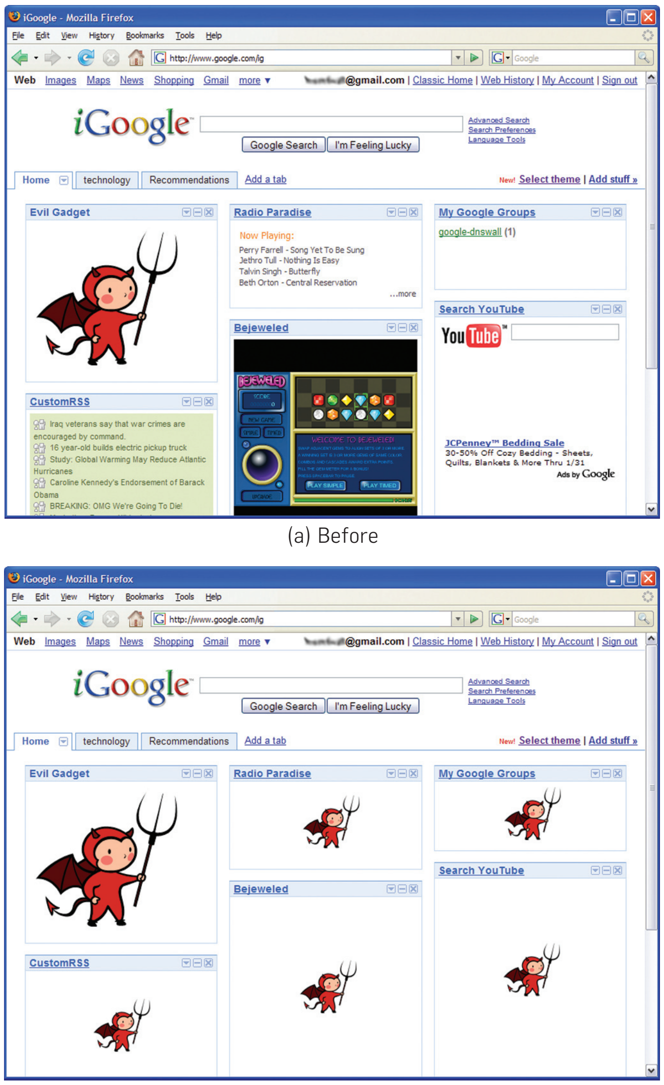
3.4. Stricter policies

Although browser vendors do not document their navigation policies, we reverse engineered the policies of existing browsers (see Table 2). In addition to the permissive and window policies, we found two other policies:

Descendant Policy
A frame can navigate only its descendants.

Child Policy
A frame can navigate only its direct children.

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



(a) Before

(b) After

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.[18]

To select a frame navigation policy that provides the best trade-off between security and compatibility, we appeal to the principle of *pixel delegation*. When one frame embeds a child 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. Frame navigation attacks hinge on the attacker escalating his or her privileges and drawing on otherwise inaccessible regions of the screen. The descendant policy is the most permissive (and therefore

Table 2: Frame navigation policies deployed in existing browsers prior to our work.

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

most compatible) policy that prevents the attacker from overwriting screen real estate "belonging" to another origin. Although the child policy is stricter than the descendant policy, the added strictness does not provide a significant security benefit because the attacker 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.

Maximizing the compatibility of the descendant policy requires taking the browser's scripting policy into account. Consider one site that embeds two child frames from a second origin. Should one of those child frames be permitted to navigate its sibling? Strictly construed, the descendant policy forbids this navigation because the target frame is a sibling, not a descendant. However, this navigation should be allowed because an attacker can perform the navigation by injecting a script into the sibling frame that causes the frame to navigate itself. The browser lets the attacker inject this script because the two frames are from the same origin. More generally, the browser can maximize the compatibility of the descendant policy by recognizing *origin propagation* and letting an active frame navigate a target frame if the target frame is the descendant of a frame in the same origin as the active frame. Defined in this way, the frame navigation policy avoids creating a suborigin privilege.[11] This added permissiveness does not sacrifice security because an attacker can perform the same navigations indirectly, but the refined policy is more convenient for honest Web developers.

We collaborated with the HTML 5 working group[9] and standardized the descendant policy in the HTML 5 specification. The descendant policy has now been adopted by Internet Explorer 7, Firefox 3, Safari 3.1, and Google Chrome. We also reported a vulnerability in Flash Player that could be used to bypass Internet Explorer 7's frame navigation policy. Adobe fixed this vulnerability in a security update.

4. FRAME COMMUNICATION

Unlike simple aggregators and advertisements, sophisticated mashups comprise gadgets that communicate with each other and with their integrator. For example, Yelp integrates the Google Maps gadget, illustrating the need for secure interframe communication in real deployments. Google provides two versions of its Maps gadget:

- **Frame:** In the frame version, the integrator embeds a frame to `maps.google.com`, in which Google displays a map of the specified location. The user can interact with the map, but the integrator cannot.
- **Script:** In the script version, the integrator embeds a `<script>` tag that runs JavaScript from `maps.google.com`. This script creates a rich JavaScript API that the integrator can use to interact with the map, but the script runs with all of the integrator's privileges.

Yelp, a popular review Web site, uses the Google Maps gadget to display the locations of restaurants and other businesses. 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. To deliver these advanced features, Yelp must use the script version of the Maps gadget, but this design requires Yelp to trust Google Maps completely because Google's script runs with Yelp's privileges, 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 interframe communication promises the best of both alternatives: sites with functionality like Yelp 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. Fragment identifier messaging

Although the browser's scripting policy isolates frames from different origins, clever mashup designers have discovered an unintended channel between frames, *fragment identifier messaging*,[1, 21] which 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. We analyze fragment identifier messaging in use prior to our analysis and propose improvements that have since been adopted.

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

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

changes the frame's location without reloading the frame or destroying its JavaScript context. The frame can read its fragment by polling `window.location.hash` to see if the fragment has changed. This technique can be used to send messages between frames while avoiding network latency.

Security Properties: The fragment identifier channel has less-than-ideal security properties. The browser's scripting policy prevents other origins from eavesdropping on messages because they are unable to *read* the frame's location (even though the navigation policy lets them *write* the frame's location). Browsers also prevent arbitrary 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 draw an analogy with the 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, however, because it lacks *authentication*: a recipient cannot determine the sender of a message unambiguously. The attacker might be able to replay previous messages using the browser's `history` API.

The fragment identifier channel is analogous to a channel on an untrusted network in which each message is encrypted with the public key of its intended recipient. In both cases, when 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 key differences between the fragment identifier channel and the public-key channel:

1. 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 polling the browser's clock, but obtaining high-resolution timing information degrades performance.
2. 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 implementations.

Despite these differences, we find the network analogy useful in analyzing interframe communication.

Windows Live Channels: Microsoft uses fragment identifier messaging in its Windows Live platform library to implement a higher-level channel API, `Microsoft.Live.Channels`.[21] 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. Prior to our analysis, `Microsoft.Live.Channels` used a protocol to add authentication to the fragment identifier channel. By reverse engineering the implementation, we determined that the library used the following protocol to establish a secure channel:

$$\begin{aligned} A \to B &: N_A, \mathrm{URI}_A \\ B \to A &: N_A, N_B \\ A \to B &: N_B, \mathrm{Message}_1 \end{aligned}$$

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 the classic Needham–Schroeder public-key protocol.[17] The Needham–Schroeder protocol was designed to establish a shared secret between two parties over an insecure channel. Instead of using encryption as in the Needham–Schroeder protocol, Windows Live relies on the fragment identifier channel to provide confidentiality.

The Needham–Schroeder public-key protocol has a well-known anomaly, due to Lowe,[15] that 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 when the mashup is initialized. The Lowe anomaly can be exploited to impersonate the integrator to the gadget:

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

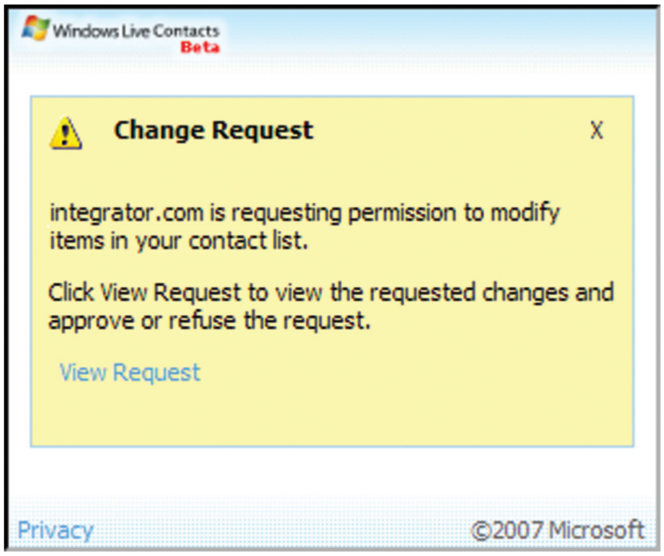
After these four messages, the attacker possesses N_I and N_G and can impersonate the integrator to the gadget. We have implemented this attack against the Windows Live Contacts gadget. The anomaly is especially problematic for the Contacts gadget because it displays the integrator's host name to the user in its security user interface (see Figure 3).

Securing Fragment Identifier Messaging: The channel can be secured using a variant of the Needham–Schroeder–Lowe protocol.[15] As in Lowe's improvement to the original protocol, we recommend including the responder's identity in the second message of the protocol, letting the honest initiator detect the attack and abort the protocol:

$$\begin{aligned} A \to B &: N_A, \mathrm{URI}_A \\ B \to A &: N_A, N_B, \mathrm{URI}_B \\ A \to B &: N_B \end{aligned}$$

We contacted Microsoft, the OpenAJAX Alliance, and IBM about the vulnerabilities in their fragment identifier messaging protocols. Microsoft and the OpenAJAX Alliance have adopted our suggestions and deployed the above protocol in

Figure 3: Lowe Anomaly. The gadget believes the request came from `integrator.com`, but in reality the request was made by `attacker.com`.



Change Request

integrator.com is requesting permission to modify items in your contact list.

Click View Request to view the requested changes and approve or refuse the request.

updated versions of their libraries. IBM adopted our suggestions and revised their SMash[14] paper.

4.2. postMessage

HTML 5[10] specifies a new browser API for asynchronous communication between frames. Unlike fragment identifier messaging, `postMessage` was designed for cross-origin communication. The `postMessage` API was originally implemented in Opera 8 and is now supported by Internet Explorer 8, Firefox 3, Safari 3.1, and Google Chrome. We discovered a vulnerability in an early version of the API, which has since been eliminated by modifications we suggested. To send a message to another frame, the sender calls the `postMessage` method:

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

In the recipient's frame, the browser generates a message event with the message, the origin (scheme, host, and port) of the sender, and a reference to the sender's frame.

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 fragment identifier messaging. The `postMessage` channel is analogous to a channel on an untrusted network in which each message is cryptographically signed by its sender. In both settings, if Alice sends a message to Bob, Bob can determine unambiguously that Alice sent the message. With `postMessage`, the `origin` property identifies the sender; with cryptographic signatures, the signature identifies signer. One difference between the channels is that cryptographic signatures can be easily replayed, but `postMessage` resists replay attacks.

Attacks: We discover an attack that breaches the confidentiality of the `postMessage` channel. Because a message sent with `postMessage` is directed at a frame, an attacker can intercept the message by navigating the frame to `attacker.com` before the browser generates the `message` event:

- **Recursive Mashup Attack:** If an integrator calls `postMessage` on a gadget contained in a frame, the attacker can load the integrator inside a frame and intercept the message by navigating the gadget frame (a descendant of the attacker's frame) to `attacker.com`. When the integrator calls `postMessage` on the "gadget's" frame, the browser delivers the message to the attacker (see Figure 4).
- **Reply Attack:** Suppose the integrator uses the `origin` to decide whether to reply to a `message` event:

```
 if (evt.origin == "https://gadget.com")
   evt.source.postMessage(secret);
```

- The attacker can intercept the secret by navigating the source frame before the browser generates the `message` event. This attack can succeed even under the child frame navigation policy if the honest gadget sends its messages via `top.postMesage(...)`. The attacker's gadget can embed a frame to the honest gadget and navigate the honest gadget before the integrator replies to the "gadget's" frame (see Figure 5).

Securing postMessage: Although sites might be able to build a secure channel using the original `postMessage` API, we recommend that `postMessage` provide confidentiality natively. In MashupOS,[22] we previously proposed that inter-frame communication APIs should let the sender specify the origin of the intended recipient. Similarly, we propose

Figure 4: Recursive mashup attack. The attacker navigates the gadget's frame to `attacker.com`.

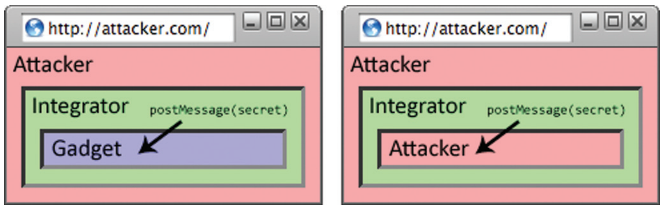
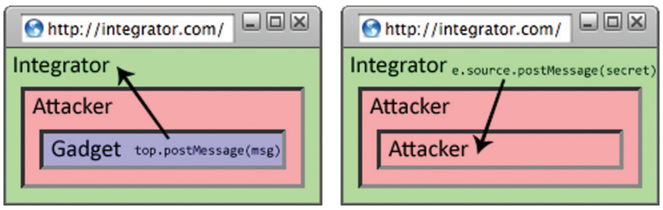


Figure 5: Reply attack. The attacker intercepts the integrator's response to the gadget's message.



extending the `postMessage` API with a second parameter: `targetOrigin`. The browser will deliver the message *only if* the frame's current origin matches the specified `targetOrigin`. If the sender uses "*" as the `targetOrigin`, the browser will deliver the message to any origin. Using this improved API, a frame can reply to a message using the following idiom:

```
  if (evt.origin == "https://gadget.com")
    evt.source.postMessage(secret, evt.origin);
```

We implemented this API change as patches for Firefox and Safari. Our proposal was accepted by the HTML 5 working group.[8] The improved API is now available in Internet Explorer 8, Firefox 3, Safari 4, and Google Chrome.

5. RELATED WORK

5.1. Mitigations for gadget hijacking

SMash[14] mitigates gadget hijacking (also known as "frame phishing") by carefully monitoring the frame hierarchy and browser events for unexpected navigations. Although neither the integrator nor the gadgets can prevent these navigations, the mashup can alert the user and refuse to function if it detects an illicit navigation. SMash waits 20s for a gadget to load before assuming that the gadget has been hijacked. An attacker might be able to fool the user into entering sensitive information during this interval, but using a shorter interval might cause users with slow network connections to receive spurious warnings. The descendant policy makes such mitigation unnecessary.

5.2. Safe subsets of HTML and JavaScript

One way to sidestep the security issues of frame-based mashups is to avoid using frames by combining the gadgets and the integrator into a single document. This approach forgoes the protections afforded by the browser's security policy and 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. Several open-source implementations (FBML, ADsafe, and Caja) are available. FBML is currently the most successful subsets and is used by the Facebook Platform.

5.3. Subspace

In Subspace,[13] we used a multilevel 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.

5.4. Module tag

The proposed `<module>` tag[2] is similar to the `<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 explicitly unauthenticated because the module lacks a principal.

5.5. Security = restricted and jail

Internet Explorer supports a `security` attribute[16] for frames. When set to `restricted`, the frame's content cannot run JavaScript. Similarly, the proposed `<jail>` tag[5] encloses untrusted content and prevents the jailed content from running JavaScript. Unfortunately, eliminating JavaScript prevents gadgets from offering interactive experiences.

5.6. MashupOS

In MashupOS,[22] we proposed new primitives both for isolation and communication. Our improvements to frame navigation policies and `postMessage` let developers realize some of the benefits of MashupOS using existing browsers.

6. CONCLUSION

Web sites that combine content from multiple sources can leverage browser frame isolation and interframe communication. Although the browser's same-origin security policy restricts direct access between frames, recent browsers have used differing policies to regulate when one frame may navigate another. The original permissive frame navigation policy admits a number of attacks, and the subsequent window navigation policy leaves mashups vulnerable to similar attacks. The better descendant policy, which we collaborated with the HTML 5 working group to standardize, balances security and compatibility and has been adopted by Internet Explorer 7 (independently), Firefox 3, Safari 3.1, and Google Chrome.

In existing browsers, frame navigation can be used for interframe communication via a technique known as fragment identifier messaging. If used directly, fragment identifier messaging lacks authentication. We showed that the authentication protocols used by `Windows.Live.Channels`, SMash, and OpenAjax 1.1 were vulnerable to attacks but can be repaired in a manner analogous to Lowe's variation of the Needham–Schroeder protocol.[15] This improvement has been adopted by Microsoft Windows Live, IBM Smash,[14] and the OpenAjax Alliance.

Originally, `postMessage`, another interframe communication channel, suffered the converse vulnerability: using frame navigation, an attacker could breach confidentiality. We propose extending the `postMessage` API to provide confidentiality by letting the sender specify an intended recipient. Our proposal has been adopted by the HTML 5 working group, Internet Explorer 8, Firefox 3, Safari 4, Google Chrome, and Opera.

With these improvements, frames provide stronger isolation and better communication, becoming a more attractive feature for integrating third-party Web content. One important area of future work is improving the usability of the browser's security user interface. For example, a gadget is permitted to navigate the top-level frame, redirecting the user from the mashup to a site of the attacker's choice. Although the browser's location bar makes this navigation evident, many users ignore the location bar. Another area for future work is improving isolation in the face of browser implementation errors, which could let a gadget subvert the browser's security mechanisms.

Acknowledgments

We thank Mike Beltzner, Sumeer Bhola, Dan Boneh, Gabriel E. Corvera, Ian Hickson, Koji Kato, Eric Lawrence, Erick Lee, David Lenoe, 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. C

References

1. Burke, J. Cross domain frame communication with fragment identifiers. http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html.
2. Crockford, D. The <module> tag. http://www.json.org/module.html.
3. Daswani, N., Stoppelman, M. et al. The anatomy of Clickbot.A. In Proceedings of the HotBots (2007).
4. Dhamija, R., Tygar, J.D., Hearst, M. Why phishing works. In CHI '06: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (2006).
5. Eich, B. JavaScript: Mobility and ubiquity. http://kathrin.dagstuhl.de/files/Materials/07/07091/07091 EichBrendan.Slides.pdf.
6. Felten, E.W., Balfanz, D., Dean, D., Wallach, D.S. Web spoofing: An Internet con game. In Proceedings of the 20th National Information Systems Security Conference (1996).
7. Guninski, G. Frame spoofing using loading two frames. Mozilla Bug 13871.
8. Hickson, I. Re: A potential slight security enhancement to postMessage, Februrary 2008. http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-February/013949.html.
9. Hickson, I. Re: HTML5 frame navigation policy, April 2008. http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-April/014597.html.
10. Hickson, I. et al. HTML 5 Working Draft. http://www.whatwg.org/specs/web-apps/current-work/.
11. Jackson, C., Barth, A. Beware of finer-grained origins. In Proceedings of the Web 2.0 Security and Privacy (W2SP) (2008).
12. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D. Protecting browsers from DNS rebinding attacks. In Proceedings of of the 14th ACM Conference on Computer and Communications Security (CCS) (2007).
13. Jackson, C., Wang, H.J. Subspace: Secure cross-domain communication for web mashups. In Proceedings of the 16th International World Wide Web Conference (WWW) (2007).
14. De Keukelaere, F., Bhola, S., Steiner, M., Chari, S., Yoshihama, S. SMash: Secure cross-domain mashups on unmodified browsers. In Proceedings of the 17th International World Wide Web Conference (WWW) (2008). To appear.
15. Lowe, G. Breaking and fixing the Needham–Schroeder public-key protocol using FDR. In Proceedings of TACAS (volume 1055, 1996), Springer Verlag.
16. Microsoft. SECURITY attribute (FRAME, IFRAME). http://msdn2.microsoft.com/en-us/library/ms534622(VS.85).aspx.
17. Needham, R.M., Schroeder, M.D. Using encryption for authentication in large networks of computers. Commun. ACM, 21, 12 (1978), 993–999.
18. Ross, D., January 2008. Personal communication.
19. Ruderman, J. JavaScript Security: Same Origin. http://www.mozilla.org/projects/security/components/same-origin.html.
20. Stuttard, D., Pinto, M. The Web Application Hacker's Handbook. Wiley, 2007.
21. Thorpe, D. Secure cross-domain communication in the browser. Archit. J. 12 (2007), 14–18.
22. Wang, H.J., Fan, X., Howell, J., Jackson, C. Protection and communication abstractions for web browsers in MashupOS. In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP) (2007).

Adam Barth (abarth@eecs.berkeley.edu), UC Berkeley.

Collin Jackson (collinj@cs.stanford.edu), Stanford University.

John C. Mitchell (mitchell@cs.stanford.edu), Stanford University.