# Talking to Strangers Without Taking Their Candy: Isolating Proxied Content

Adrienne Felt, Pieter Hooimeijer, David Evans, Westley Weimer
University of Virginia
{felt, pieter, evans, weimer}@cs.virginia.edu

## ABSTRACT

Social networks have begun supporting external content integration with platforms like OpenSocial and the Facebook API. These platforms let users install third-party applications and are a popular example of a *mashup*. Content integration is often accomplished by proxying the third-party content or importing third-party scripts. However, these methods introduce serious risks of user impersonation and data exposure. Modern browsers provide no mechanism to differentiate between trusted and untrusted embedded content. As a result, content providers are forced to trust third-party scripts or ensure user safety by means of server-side code sanitization. We demonstrate the difficulties of server-side code filtering – and the ramifications of its failure – with an example from the Facebook Platform. We then propose browser modifications that would distinguish between trusted and untrusted content and enforce their separation.

## Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: Security and protection

## General Terms

Security, Design

## Keywords

Mashups, social networking sites, Same Origin Policy

## 1. INTRODUCTION

Modern browser architecture lags behind the evolution of Web design. Web pages are no longer static, monolithic HTML documents. Second-generation Web design emphasizes dynamic content and customization. From blog comments to Google Gadgets, third-party components are regularly included in sites. Social networking sites are among the most popular proponents of third-party gadgets. In the simplest situations, the third party input is text; in the most complicated cases, *mashups* aggregate robust JavaScript applications. Current browsers, however, offer no protection from imported or proxied third-party code.

With imported scripts, `<script>` tags are used to import an external script. The imported script runs with the privileges of the host page. When an amateur blogger adds a Flickr badge [9] to her page with a `<script>` tag, for example, Flickr gains full access to the blog's content and visitor credentials. Mashup hosts have no choice but to accept the risks of this technique.

Some professional mashup aggregators proxy third-party code. Since browsers do not differentiate between proxied code and proprietary code, the aggregators try to ensure user safety with server-side code sanitization. This is critical on sites (such as social networks) that host personal and private information. Filtering out malicious JavaScript, however, is a difficult task; cross-site scripting attacks, which exploit weaknesses in input filters, comprised nearly a quarter of publicly-disclosed vulnerabilities in 2006 [1]. The high complex-

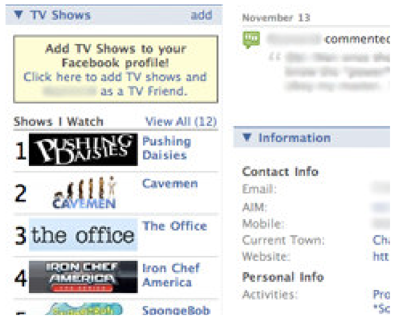

**Figure 1: This Facebook profile has an application (TV Shows) installed as an example of a mashup.**

ity of code filters places responsibility on the mashup aggregator; it is unlikely that small startup or amateur sites would be able to provide similar user protection. Further, since the filtering is done by the aggregator, the end client has no control over the policy and must trust the aggregator to perform the filtering correctly. We illustrate the difficulty of server-side filtering with an attack on the Facebook Platform and show how a single input verification vulnerability can expose an entire site.

Mashup integrators are limited in their ability to provide safe designs, since browsers do not let them distinguish between proprietary and third-party content. We propose a new `untrusted` attribute for `<div>` tags that lets mashup aggregators mark untrusted embedded content. The browser can then enforce a one-way visibility policy so that the untrusted content does not have access to the surrounding page. This shifts the responsibility of user safety from the mashup server to the browser, which is where other source-related security checks occur. The browser is the definitive authority for detecting script execution; unlike server-side filters, it will never miss executable JavaScript fragments and it has access to run-time context information.

Unlike other proposals aimed at preventing cross-site scripting, we directly address the underlying design problem behind content integration. Our new attribute accommodates safe, full-featured mashups while letting mashup integrators retain the performance and control benefits of proxying code.

Section 2 explains how the current browser security model is inadequate for common mashup services, and Section 3 describes a vulnerability in the Facebook platform that exemplifies the problem. Our proposed attribute and prototype implemented as a Firefox extension are described in Section 4. Section 5 discusses related work.

## 2. BACKGROUND

Current browsers apply an industry-standard *Same Origin Policy* (SOP) to cross-domain interactions on the client-side. The SOP governs how security and isolation mechanisms will be applied to third-party content. Section 2.1 explains the SOP and how it was designed for use with inline frames. Section 2.2 and Section 2.3 describe two alternate methods of content integration and how the SOP is not adequate for either.

### 2.1 Same Origin Policy

The Same Origin Policy isolates documents from different domains. A script from one document may access another document if and only if they are from the same source domain. All parts of the URLs except for the file name must be identical for the two domains to be considered the same; thus, `www.a.edu/XX.html` and `www.a.edu/foo/YY.html` are from the same origin, but `www.YY.a.edu` and `www.XX.a.edu` are not.

Cross-domain content integration is supported with inline frames, which provide complete isolation. The `<iframe src="...">` tag prompts a child frame to be loaded from the specified `src`. If the `src` attribute is of the same domain as the outer page, scripts can freely communicate between the two frames. If the `src` attribute points to a different domain (as is the case in mashups), complete isolation is imposed between the two frames. Web pages are modeled in a parent-child tree structure known as the *Document Object Model* (DOM), and an iframe from an external source places a permanent barrier through the tree.

### 2.2 Imported Scripts

Cross-domain `<script>` tags load scripts from other domains. An imported script will execute immediately and with the same privileges as the surrounding content. The page that has included the script must trust it fully, since the script may access anything on that page and in that same domain. The mashup host has no opportunity to validate or rewrite the script content.

Despite this security risk, the use of cross-domain scripts persists because the two-way isolation policy of the SOP is too restrictive for many applications. The mashup aggregator may need to call methods associated with the third-party script, which would not be possible with a script in an iframe.

For example, Google Maps' JavaScript API may be added to pages to display a map [10]. The map can be sandboxed in an iframe or imported as a script. Using an iframe means that the map can only be displayed; choosing to import it as a script lets the content integrator interact with the map. Such interaction is increasingly common; one example is EveryBlock, which takes advantage of the interactivity to overlay restaurant reviews, crime locations, and neighborhood postings onto city maps [5].

### 2.3 Proxying

Normally, Web content is obtained directly from the server that hosts it. When a user navigates to a page, her browser directly establishes a link with the host network server to fetch content. If cross-domain frames are included in the page, the user's browser makes a second round of requests to the third-party servers.

In the case of *proxied* content, the user's browser sees only one data transaction. When the user navigates to such a mashup page, the target mashup server relays the request and fetches extra content, such as gadgets, from third-party servers. The mashup server then directly embeds the third-party content into the page, and returns it to the user as if all of the content were from the mashup server. From the standpoint of the browser, all of the code in the aggregated page has the same origin. Under the Same Origin Policy, the gadgets receive the same privileges as the host content and can access page elements and make use of viewer credentials. If the mashup integrator wishes to incorporate external content while ensuring user safety, it must try to filter the application content at the server side.

Filtering content for security is difficult. Even regular user input fields that only allow a few HTML elements are difficult to sanitize properly. For example, in October 2005, the Samy worm took down MySpace for twelve hours by inserting JavaScript into a profile field and spreading to over a million profiles [17]. Filtering mashup content is even harder because useful mashups need to support executable content, including JavaScript. The stakes are high for large mashup providers like social networking sites, where an oversight in their code sanitizer could lead to a costly denial of service attack or a leak of personal information.

## 3. EXAMPLE MASHUP ATTACK

Facebook runs a large-scale proxying service for third-party applications that users may install through the Facebook Platform [7]. These third-party applications can be added to Facebook profiles and pages. This feature has proven extremely popular: the highest-ranked Facebook applications have over twenty-five million users [6]. Facebook applies code transformation techniques to the untrusted content to filter out disallowed executable actions. Facebook's rival social networking site platform, OpenSocial, will soon also support JavaScript rewriting techniques for partner sites who "wish to avoid the overhead of putting third party JavaScript into iframes" [11].

To illustrate the difficulty and danger of server-side code filtering, this section describes a vulnerability we discovered in the Facebook Platform[1] and the ramifications of potential attacks through such a vulnerability.

### 3.1 Improper Code Transformation

Facebook requires applications to be written in *Facebook Markup Language* (FBML). FBML is a non-executable subset of HTML that is supplemented by proprietary Facebook tags. The proprietary tags aid developers by abstracting complex scripts; for example, the simple `<fb:friend-selector>` tag is turned into a predictive type-ahead user input field.

By comparing our FBML input and Facebook's generated HTML output, we discovered an oversight in their transformation of the `<fb:swf>` tag, which embeds an Adobe Flash `.swf` file into a page. To keep ostentatious graphics and audio from annoying viewers, a static preview image is provided as a link to the dynamic Flash content. The `<fb:swf>` tag therefore includes an `imgstyle` attribute for the image. The `imgstyle` attribute was stripped of the ", <, and > characters but not checked for executable content. To take advantage of this, an attacker could inject code into a Facebook-served page as follows:

```
<fb:swf swfsrc="http://foo/flash.swf" imgsrc=
"http://foo/bar.jpg" imgstyle="-moz-binding:
url(\‘http://foo/xssmoz.xml#xss\’); background-
image:expression(...)  " />
```

After conversion by Facebook, the static preview image would be included in the final profile page as:

```
<img src="http://facebook/cached.jpg" style=
"-moz-binding:url(‘http://foo/xssmoz.xml#xss’);
background-image:expression(...)  " />
```

The content of the final `style` attribute references JavaScript. In Firefox, the `-moz-binding:url` selector loads and executes JavaScript from an XML file. In Internet Explorer, the `background-image` tag evaluates the JavaScript contents of the `expression()` function. The malicious script executes every time a Facebook user views a profile with the malicious application installed.

### 3.2 Ramifications

Once arbitrary third-party JavaScript can been introduced into a site, an attacker can impersonate the page viewer and perform malicious actions on the user's behalf. The browser does not realize that the JavaScript is untrusted, so the untrusted code's requests will be executed with the user's session state and permissions. This attack is known as *session surfing* or *Cross-Site Request Forging* (CSRF).

The injected JavaScript can crawl through the page to look for desired values (e.g., viewer ID, form keys) and open up hidden iframes that can POST forms to Facebook. Since the page viewer is logged in to the site, the forms will be associated with the user's session and will successfully submit.

Any user action on the site that can be submitted as a form can also be performed by the attacker. This includes worm-like behavior, since the injected script could install itself on viewers' accounts. On a heavily trafficked site, self-replicating code can spread rapidly; the Samy worm spread to over a million MySpace profiles in five hours [17]. Users with stored credit card information could be at risk if they are not required to re-enter their password before purchasing something on the site. (Facebook implemented a secondary login for credit card protection in Fall 2007.)

## 4. BROWSER-SIDE ENFORCEMENT

Under the current browser security model, imported

[1]Facebook was notified and patched the hole within three weeks. Full code and a video demonstration are available at [8].

scripts and proxied code are treated as if they are proprietary page content. This places a large burden on mashup integrators, puts users at risk, and limits the ability of small and amateur sites to participate in mashup design. Only scripts from well-known companies are trusted enough to be imported, and only large sites have the resources to safely proxy content. As shown in Section 3, even large sites make mistakes in their filters that allow malicious scripts through.

We argue that origin-related policy enforcement belongs in the browser. The browser will always correctly identify all JavaScript in a page, since the browser cannot execute code if it does not recognize it. Standardization is another advantage: instead of different mashup integrators offering varying levels of protection, users will always be protected by their browsers. The security feature will be available to amateur and professional sites alike, and mashup integrators won't need to keep writing the same content-filtering code.

Currently, browsers have no way to differentiate between trusted and untrusted content in aggregated documents. We propose to solve this by introducing a new `untrusted` marker, described in Section 4.1, that identifies third-party code so that security policies can be applied to it. JavaScript rewriting can be used *within* the browser to restrict untrusted content. Section 4.2 describes our prototype implementation as a Firefox extension.

## 4.1 Marking Untrusted Content

The new `<div untrusted="true">` attribute is used to mark third-party code. Any code inside an untrusted `<div>` is restricted by these policies:

- Untrusted JavaScript cannot access any page content outside of its containing `<div>`. This means that the Document Object Model (DOM) above the `<div>` is completely inaccessible.
- Untrusted JavaScript cannot access any global variables or functions. This also applies to interaction between different `untrusted` blocks.
- Untrusted code may not make XMLHttpRequests to or open iframes from the mashup's domain.

These restrictions only apply to the untrusted content; the mashup integrator's other content is trusted and unchanged.

The primary purpose of the `untrusted` tag is to protect the surrounding page content from the third-party code. Facebook, for example, could place third-party gadgets in these `untrusted` blocks without concern that the profile information displayed in the rest of the page would leak. Similarly, external `<script>` tags could be surrounded by `<div>` tags to shield the page from the script provider.

In order to ensure that the untrusted code cannot access the page in other ways, the untrusted code cannot be allowed to make XMLHttpRequests to or open iframes from the mashup's domain. If untrusted code could do either, it would be able to read the DOM by obtaining a new copy of the document.

The rules also prevent sophisticated cross-site request forging attacks. Web developers prevent simple XDRF attacks by putting secret form keys in pages. Forms then cannot be submitted without the user having actually visited the page to obtain a copy of the key. The XDRF attack outlined in Section 3.2 circumvents this security measure by using a cross-site scripting vulnerability to comb through the page DOM to find the key. Under our `untrusted` rules, however, the third-party code would be unable to access the form keys through the DOM.

Adding a new attribute to a `<div>` tag is backwards-compatible. Older browsers will simply ignore it, so their users will receive none of the benefits of our security design.

Our current rules are simple and treat all untrusted code identically. Future enhancements to the rules could provide rich communication interfaces and policies. Untrusted code could be allowed selective access to trusted content or other blocks of untrusted code. Mashup integrators could specify the embedded content's source, and users could set client-side policies that permit or deny this content based on its source. None of these future features can develop, however, until aggregators are given the ability to differentiate between proprietary and untrusted content. Our `untrusted` tag is the first step in this direction.

## 4.2 Implementation

The `untrusted` attribute and its associated properties could be easily implemented in browsers without major changes to browser design. Browser-side script rewriting can be used to mark the untrusted content and enforce the rules. The browser's HTML parser statically instruments the untrusted content while rendering the page. Runtime checks are also performed to transform dynamically-generated code.

All untrusted JavaScript variables, objects, functions, and DOM elements are renamed by the browser. Their names are prepended with either a random string or a prefix specified by the mashup integrator. For example, if the mashup integrator used a `<div untrusted ="true" name="a123">` tag, all of the enclosed variables would be prefixed with `a123`. This applies to names on both sides of assignments: `var foo = bar` would become `var a123_foo = a123_bar`. The same strategy applies to CSS references, so that styling can only be applied to elements that belong to the untrusted content.

This renaming scheme prevents the untrusted code from accessing unauthorized content or functions: any reference to a trusted element will fail because it will be redirected to a different variable name. The browser would also mediate references through the DOM to prevent the exploitation of parent-child relationships. The mashup integrator retains its privileges to the untrusted code, since it may access untrusted DOM elements and JavaScript functions by using the prefix. It is not necessary to keep the prefix secret.

Simple static renaming is not adequate to enforce restrictions, since reflexive JavaScript operations like `write` and `eval` can be used to introduce new, uncensored JavaScript. Facebook JavaScript (FBJS) [7] and Yahoo's ADsafe [20] solve this problem by disallowing these functions. At the browser side, however, we need to accept the full JavaScript language. To accomplish this and handle JavaScript's reflexivity, we use the JavaScript instrumentation techniques previously outlined by BrowserShield [16].

Function calls, method calls, object properties, object creation, `with` constructs, and `in` constructs in untrusted code are rewritten by the browser to point to runtime checks that guard the integrity of the renaming scheme. The runtime checks add the scoping prefix and insert more dynamic checks into the new content. DOM access to parent, child, and sibling nodes and innerHTML are rewritten to point to permission checks. (If the node is beyond the untrusted `div` or the innerHTML includes new JavaScript, appropriate action is taken.) Rewriting rules are also applied to external scripts that are imported by an untrusted block.

We have built a Firefox extension that prototypes the `untrusted` tag and JavaScript rewriting. In the prototype, `<div>` tags with a `class` of `untrusted` are considered untrusted. The extension intercepts a Firefox page load event and passes the page through Mozilla's built-in SAX parser. Untrusted DOM elements are renamed, and we create a global `checker` object. The `checker` object provides methods for runtime rule enforcement. The `<script>` tags are sent to Mozilla Narcissus's JavaScript lexer and parser; we edit the resulting abstract syntax tree to add prefixes and runtime checks. As our extension finishes altering sections of the document, the buffered XHTML is sent to Firefox for a regular page load. Runtime checks point to our global `checker` object. For example, `foo(b)` becomes `checker.func("foo",b)`.

## 5. RELATED WORK

Several proposals to add new tags that disallow JavaScript have been put forth to prevent cross-site scripting (Section 5.1). More closely related, `Sandbox` and `Module` tags have been suggested as more flexible alternatives to iframes (Section 5.2). Section 5.3 surveys other techniques for preventing cross-site scripting.

### 5.1 Disallowing JavaScript

Three new browser features aim to prevent the execution of JavaScript in untrusted static content. In Internet Explorer 6+, the parent document of a frame can set its `security` attribute to `restricted` to prevent the frame from running scripts [15]. Brendan Eich (author of JavaScript and CTO of the Mozilla Corporation) has suggested a `<jail>` tag that would disable JavaScript for the jail's contents [4]. Browser-Enforced Embedded Policies (BEEP) allow Web developers to define a whitelist of scripts that may run in a page [13]. These proposals are targeted at isolating wholly-static content and do not apply to interactive mashup applications such as social network platforms.

### 5.2 iframe Alternatives

Two iframe alternatives, `Sandbox` [19] and `Module` [2], have been proposed. The `<Sandbox>` and `<OpenSandbox>` tags would let the mashup integrator isolate frame content regardless of its origin. Additionally, the `<OpenSandbox>` tag allows one-way visibility for the mashup integrator to see into the container [19]. The `<Module>` tag would enforce full DOM isolation regardless of page origin while permitting the passing of JSON text objects [2].

Unlike our proposal, these tags both require an extra page load, similar to an `<iframe>`. Highly trafficked sites often are designed to use proxying to reduce load time and evenly distribute bandwidth. Additionally, proprietary tags that ease development (like those included in FBML) require proxying. Our design would let sites retain the benefits of proxying while adding new security features.

### 5.3 XSS Prevention

Server-side static and dynamic analysis techniques help prevent cross-site scripting vulnerabilities when the input is expected to be static [3, 14, 12]. An anomaly-based intrusion detection system analyzes Web server logs and compares the profile to incoming requests [14]. These approaches all assume that *no* executable content should be permitted, and therefore are not helpful for preventing attacks on partially-executable content.

Client-side dynamic taint analysis can be used to monitor the flow of sensitive information (cookies, history, form values) in browsers [18]. JavaScript instrumentation has been previously used to enforce user-specified policies (e.g., no pop-up windows, do not send cookies to another domain) [21]. Both ideas mitigate the effects of untrusted JavaScript, but neither one directly addresses the design problem. Our `untrusted` tag lets Web developers explicitly identify untrusted code so that standard browser origin policies may be applied.

## 6. CONCLUSION

Current browsers are ill-equipped to provide security for third-party scripts and proxied content. Either no security is provided at all (imported scripts), or the mashup integrator becomes responsible for complex server-side filtering (proxied content). Not only is server-side filtering resource-intensive, but it is also complex and difficult to do correctly.

We propose the addition of a new `untrusted` tag that will identify third-party scripts and proxied content so that policies may be applied to them. Imported scripts and proxied content should be subject to one-way visibility rules so that the mashup integrator's code can access them but they cannot access outer proprietary content. The browser will always be correct about what is executable JavaScript and what is not, since the browser is responsible for executing scripts. Mashup aggregators and their users would thereby be protected by browser-enforced policies.

Our prototype Firefox extension implements this new tag and security policy with client-side rewriting techniques. The untrusted JavaScript is statically and dynamically altered to enforce a namespace. The extension's design could be migrated into browsers with minimal changes to browser architecture.

## ACKNOWLEDGMENTS

This work was partially funded by the NSF through the CyberTrust program, award CNS 0627523.

## REFERENCES

[1] M. Broersma. Cross-site scripting the top security risk. Technical report, `http://www.networkworld.com/news/2006/091806-cross-site-scripting-the-top-security.html`, Sep 2006.

[2] D. Crockford. The module Tag. Technical report, `http://www.json.org/module.html`, Oct 2006.

[3] G. A. DiLucca, A. R. Fasolino, M. Mastoianni, and wP. Tramontana. Identifying Cross Site Scripting Vulnerabilities in Web Applications. In , 2004.

[4] B. Eich. JavaScript: Mobility and Ubiquity. Technical report, `http://kathrin.dagstuhl.de/files/Materials/07/07091/07091.EichBrendan.Slides.pdf`, Sep 2007.

[5] EveryBlock Incorporated. *EveryBlock*. `http://chicago.everyblock.com/`.

[6] Facebook. *Facebook Application Directory*. `http://uva.facebook.com/apps/`.

[7] Facebook. *Facebook Platform Developer Guide*. `http://developers.facebook.com/`.

[8] A. Felt. The Facebook Chronicles. Technical report, `http://www.cs.virginia.edu/felt/fbook/`, Aug 2007.

[9] Flickr. *Create your own Flickr badge*. `http://www.flickr.com/badge.gne`.

[10] Google. *What is the Google Maps API?* `http://code.google.com/apis/maps/`.

[11] Google. Google Launches OpenSocial to Spread Social Applications Across The Web. Technical report, `http://www.google.com/intl/en/press/pressrel/opensocial.html`, Nov 2007.

[12] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web*, 2004.

[13] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610, 2007.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 2006 Symposium on Security and Privacy*, 2006.

[15] Microsoft. SECURITY Attribute (FRAME, IFRAME). Technical report, `http://msdn2.microsoft.com/en-us/library/ms534622(VS.85).aspx`.

[16] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[17] Samy. Technical explanation of The MySpace Worm. Technical report, `http://namb.la/popular/tech.html`, Oct 2005.

[18] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Conference*, 2007.

[19] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[20] Yahoo! *ADsafe*. `http://adsafe.org`.

[21] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.