Content Provider Conflict on the Modern Web

Terri Oda, Anil Somayaji, Tony White
School of Computer Science, Carleton University
Ottawa, Ontario, Canada
{toda,soma,arpwhite}@scs.carleton.ca

***Abstract*—Today many web pages include externally sourced content. Advertisements, video, blog "trackbacks," search—these and other features of the modern web are provided by third-party servers. Such external content is so popular that content is often incorporated from more than one source. In this paper we argue that such multiple inclusions are a significant security risk because of the potential for conflict between included elements. In particular, the use of JavaScript to provide external content means that providers can observe and interfere with each other. Financial incentives and competitive advantage provide motivation for such conflicts, both for criminals and for legitimate enterprises. To prevent users and web content providers from becoming collateral damage, we must develop and deploy practical techniques for isolating externally provided web content. This paper outlines the security threat posed by combining content from different providers and describes requirements for a solution.**

I. Introduction

From the beginning of the World Wide Web, HTML pages have been composite documents, incorporating elements from multiple sources. Early pages mostly used text and images from a single web server; modern web pages, however, include content from multiple organizations. Some of these inclusions provide functional enhancements such as search services, blog "trackback" links, and video players; others supply the advertisements that are the economic foundation of much of the web.

Standard HTML4 mechanisms for incorporating external content (such as the `img` and `embed` tags) restrict them to a portion of a page: they can only be displayed within a box within the page, they cannot observe the rest of the page, and they can only receive user input when mouse or keyboard events are directed to them. Many web developers, however, have found these mechanisms to be too restrictive for dynamic content, and so they have turned to another Web technology: JavaScript.

The most common mechanisms for including external content today require the web page author to incorporate a small fragment of boilerplate JavaScript code. This code will typically load more code from a third party server; this additional code is what provides the actual functionality. Unlike HTML-based inclusion mechanisms, included JavaScript has full access to a page: all of the content and all of the events.

Many have recognized that such inclusions could represent a security threat, particularly if the external JavaScript code is compromised (e.g., [3]). Others have recognized the particular dangers of web mashups—web applications that combine together ("mash up") two or more existing web applications or pages [12], [10], [7]. What has not been appreciated, however, is that the common case—inclusion of content from multiple providers in an "ordinary" web page—itself constitutes a security risk. The risk comes from the opportunities and incentives for conflicting code.

Specifically, the commercial agendas of external content providers seldom align; however, all JavaScript code is considered trusted within the confines of a page: each piece of JavaScript can access all of a page's code and data. Thus, it is possible for one content provider to manipulate the code and data included from another. This manipulation can be used to degrade service, divert advertising revenue, and conduct click fraud.

While current proposals for securing JavaScript in web mashups can help secure included content in certain circumstances, they break important uses such as context-sensitive advertisements (such as Google Adwords) while introducing usability issues for unsophisticated web developers. Thus, we believe that new solutions are needed for securing external web content.

This paper has two key contributions. The first is identifying the security threat caused inclusion of content from multiple content providers. This threat is made more dangerous by the assumption that content providers will interact only in safe ways (or not at all), as well as the assumption that most web pages are not in need of protections currently reserved for more complex web applications. Such assumptions can lead to inappropriate security decisions or design of systems which do not easily address the full scope of the problem. Because of the risks involved in such assumptions, the second contribution here is in outlining the requirements for a solution to this problem.

The rest of this paper proceeds as follows. In Section II, we explain in more detail how external content is included in web pages. Section III describes the standard security restrictions on JavaScript and their limitations. We explore the idea of content providers being adversaries in Section IV, including specific attack scenarios. Some requirements for a solution are discussed in Section V. In Section VI, we present related work in web security including work on web mashups. Section VII discusses the opportunities and challenges for better JavaScript isolation mechanisms and Section VIII concludes.

II. Web Page Composition Using Multiple Sources

Most webpages are constructed using information from several sources. Sites that have content they want others to

```
1 <object width="425" height="355">
2 <param name="movie" value="http://www.youtube.com/v/FiARsQSlzDc">
3 </param>
4 <param name="wmode" value="transparent"></param>
5 <embed src="http://www.youtube.com/v/FiARsQSlzDc"
6          type="application/x-shockwave-flash" wmode="transparent"
7          width="425" height="355">
8 </embed></object>
```

Listing 1. Code for including a video on a web page, as generated by YouTube. Note that the information about the URL for the video is repeated both as a parameter within the object tag (line 2) and inside the embed tag (line 5). This is to ensure compatibility with more browsers, as some use the object tag and others use embed.

Figure 1. Inclusion of an image into an HTML document results in a predictable webpage

include will often give fragments of HTML code that users can put in their page. Although browsers still vary in how they render a page, this is the easiest way to assure that anyone who wants to can include this content, be it an image, a video, or something else.

A. Including Static Content

In the most basic of HTML, there are many ways to include static content that will be the same every time the content is viewed. For example, video site YouTube generates code for people to embed video objects in their pages, as shown in Listing 1.

Here, the pertinent part is the `object` or `embed` tag which includes a flash video from YouTube into the page. Both tags are provided because some browsers only understand one or the other. The web browser reads the HTML, goes to get the video, and inserts this video into the page. It only inserts the video in where this tag was found. Images inserted with the `img` tag work the same way.

The path to content inclusion is shown in Figure 1. Here, you can see that the web page, combined with simple content such as the image shown, behaves in a predictable way, inserting the image where expected on the page.

B. Including JavaScript content

JavaScript is often used to generate content dynamically. Advertisements are a good example of this. Consider the code provided by Google for inserting an AdSense advertisement onto a web page, as described in Listing 2. Here, we have a small piece of JavaScript code which contains a few settings, followed by a link to more JavaScript code. This code then actually produces the advertisement which is to be placed on the page.

Note that there is no indication of where the advertisement should be placed. The very act of including this code allows it read and write access to the entire page. The included JavaScript code can choose to place the advertisement anywhere it deems suitable. In practice, it will place the content where the web page creator included the code, since this is the way things usually work with static content as described in Section II-A. However, this placement is not guaranteed—it is merely a convention.

Scripts are included as source, and often multiple scripts are included in the same page. Script sources are evaluated in the same context as the main page: the expectation is that the code of included scripts will not interfere with each other. Multiple inclusions work because developers respect conventions; the browser enforces no separation.

Figure 2 gives a visual representation of what could happen when JavaScript is included into a page. Unlike Figure 1, the result of this action is unpredictable. Figure 2a shows what one might expect the code to look like given JavaScript code from an advertiser: the code only adds an advertisement image into the box provided. However, we can see in Figure 2b that the JavaScript could be used to add content to a page, say to insert multiple advertisements. Finally, Figure 2c shows that JavaScript code could also be used to delete the contents of entire page. In practice, typical JavaScript from external parties does not make drastic changes to a page's appearance; however, as part of providing services such as visitor statistics and context-sensitive advertisements, included code commonly accesses virtually all parts of the including web page.

While there are clearly issues with giving external entities this level of control, there are limits placed on the functionality of JavaScript that address many security concerns. We discuss these features below.

```
<script type="text/javascript"><!--
Google_ad_client = "pub-6828282629126141";
/* 728x90, created 3/6/08 */
Google_ad_slot = "5248526188";
Google_ad_width = 728;
Google_ad_height = 90;
//-->
</script>
<script type="text/javascript"
        src="http://pagead2.googlesyndication.com/pagead/show_ads.js">
</script>
```

Listing 2. A sample advertisement inclusion (Google AdSense).

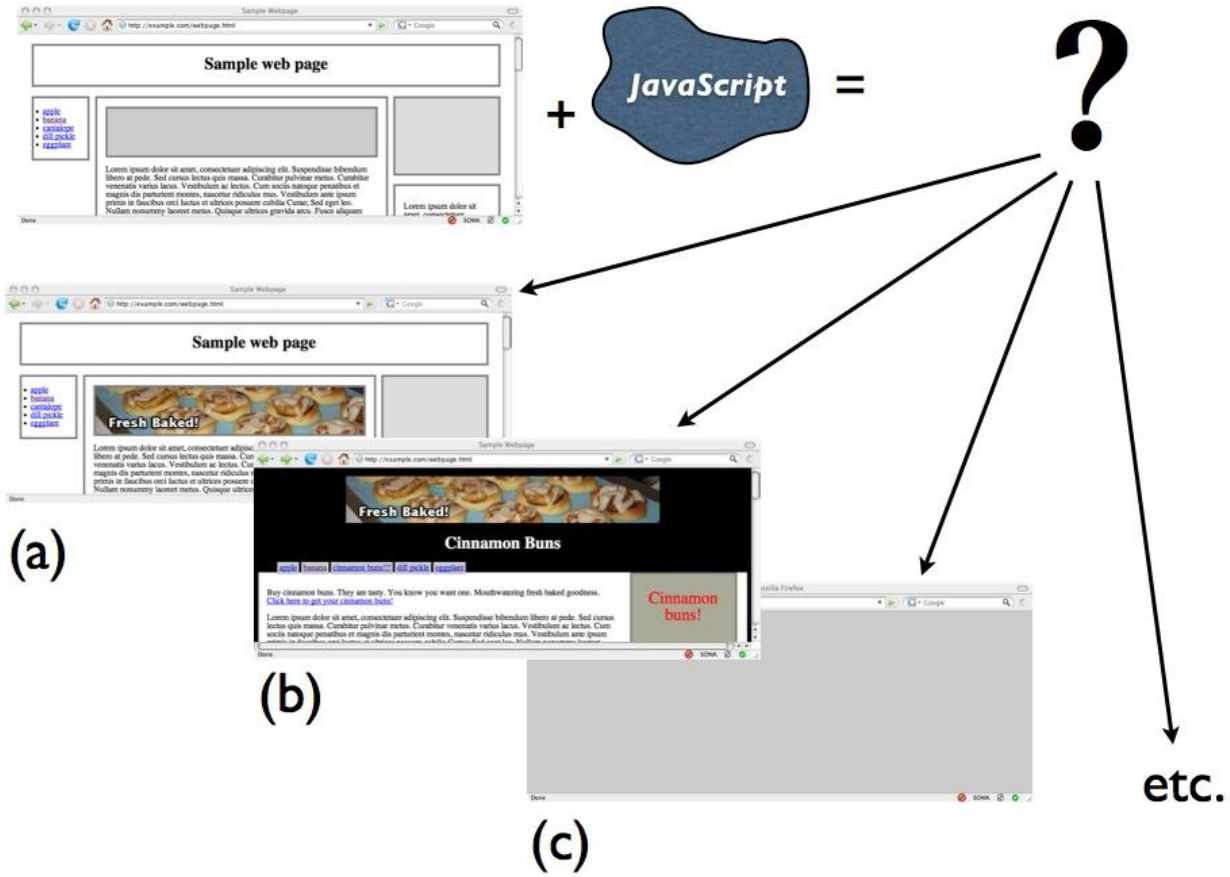

Figure 2. Inclusion of JavaScript in an HTML document leads to unpredictable results. (a) looks as one might expect given code from an advertiser: the code places an image advertisement in the box provided for the advertisement. (b) shows another possibility where the advertiser decides to modify the existing page, deleting segments, changing others to be more favorable to their advertisement. (c) shows a case where the JavaScript has replaced the page with a simple blank one.

III. JavaScript Security

JavaScript was designed to be a programming language for the web. Since web pages are provided by untrusted sources the goal was to create an environment such that code on a web page should not be able to harm web users, their computers, or other Internet hosts. To provide these guarantees, JavaScript enforces an execution sandbox. This sandbox is designed to isolate programs from each other and from the underlying operating system. Like Java applets (the first Web programming framework to employ a sandbox), the JavaScript sandbox prevents programs from accessing raw memory, the contents of other loaded web pages, and local files [4]. While elements can be incorporated from any remote resource that can be described by a URL, this content is also isolated to the including web page, thus preventing many forms of attack.

Elements from the same site residing on different pages or frames, however, often need to interact to exchange information. Thus, the JavaScript sandbox is relaxed in the case of documents originating from the same domain. Thus, JavaScript code in one document can manipulate the state of another document with the same origin. This same origin exception is routinely used to implement multi-pane interfaces, pop-up windows, and more complex AJAX-based sites such as Google Maps. While it is possible for the JavaScript sandbox to be subverted by exploiting browser flaws [1], to a large extent it succeeds in accomplishing its design goals. Our concern, however, is that malicious interactions can occur within a given page's sandbox.

The JavaScript supports powerful mechanisms for code and data separation, but these are undermined by unfettered access to the global environment and the Document Object Model (DOM). The DOM contains references to all document text, top-level functions, global variables and objects—essentially everything that is in a web page. DOM objects may be accessed through a number of global variables such as `document`. Any named node can be accessed using a call to `document.getElementByID()`. Each specific node may have its properties inspected, modified, or deleted.

All JavaScript code, including imported code, can read from and write to the global environment and, by extension, to all parts of the Document Object Model (DOM). Variables can be overwritten, functions substituted, and page elements can be read and changed arbitrarily. For example, a named link could have its color inspected, thus exposing whether or not that link had been previously visited by the user. Alternately, a node could be removed entirely from the document tree entirely, thereby compromising the document displayed.

The reason why imported code can access all global variables and functions is that all imported code and data is included in page's global JavaScript context (environment); there is so separation between imported elements and code embedded in the page (see Figure 3. This lack of separation is what allows content providers to come into conflict. For example, code from content-A.com can change attributes associated with `document` and these affect what is seen by

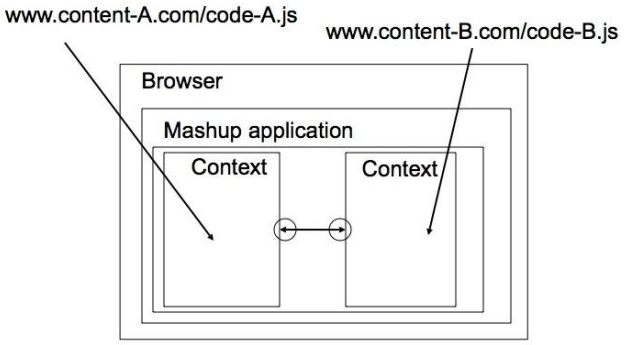

Figure 3. JavaScript included from external sources share the including application's global execution context (environment).

code included from content-B.com. Furthermore, a function named `foo` loaded from content-A.com will be overwritten by a function of the same name when content is loaded from content-B.com. This conflict arises because the top-level namespace is shared. To see the full extent of the problem, refer again to Figure 2b in which the outcome of code inclusion is undecidable. While it is expected that advertisers will only write in the box where the advertisement is intended to be displayed, there is nothing stopping them from doing other things to the page—including targeting other content providers.

To summarize, the current model for web document creation allows code and data to be included from several sources. The assumption is that all code is equally trusted and should be integrated with the same rights and privileges as the document that caused it to be included. Controlled interaction between sources—other than the same origin policy—is not provided. This inclusion of code and data can be thought of as a type of code mobility, a form of distributed computing. As with other code mobility systems, inappropriate trust relationships lead to a number of potential problems. We discuss these problems in the context of content provider conflict below.

IV. Content Providers as Adversaries

Much attention has been focused on the problem of cross site scripting (XSS), an attack in which someone injects malicious code, usually JavaScript, into a page [14]. XSS is usually accomplished by taking advantage of a bug in the input checking of a web application. Attackers who are trusted content providers, however, do not need to exploit software bugs to inject malicious JavaScript—they already have the required access.

However, why would one content providers want to interfere with another? Consider that many content providers are competitors. For example, they may both provide advertising services, they could both serve up video content, or they might both provide fast servers for accelerating delivery of web content. This inclusion of code from competitors happens on real pages. For example, CNN's website includes advertisements from advertisement.com (an AOL subsidiary), yet its search functionality is provided by Google. Salon includes

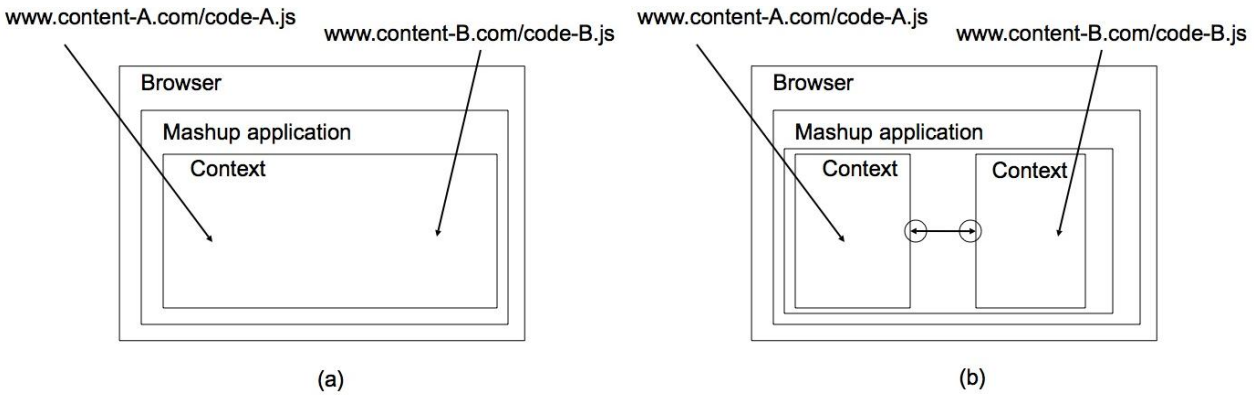

Figure 4. Current (a) and proposed (b) architectures for web mashup applications.

JavaScript code from Yahoo's Overture, Google Analytics, and other smaller advertisers and media providers. Indeed, it seems that virtually every major website that isn't owned by Google, Microsoft, or Yahoo includes content from competing organizations.

While it is likely that virtually all current external content providers are playing "nicely" with each other, we should not expect that those relationships will remain so civilized. Indeed, as the Sony rootkit debacle showed [20], major corporations are perfectly capable of using malicious software to further their interests.

Broadly speaking, there are three fundamental goals that can be achieved by manipulating JavaScript behavior within the sandbox: observation, content manipulation, and server manipulation. Here we expand on these attack strategies by exploring how a content provider A could target another provider B when they both have code resident on a page X.

A. Observation

One key task for any organization is to monitor its competition. By manipulating JavaScript, it is possible for one content provider to observe what the other is doing. For example, if A wanted to know what text advertisements B was presenting on a page, A could copy those advertisements into a separate variable (by walking the DOM tree) and then send them to another server using an HTTP GET or POST command.

Note that these are the same mechanisms used to generate customized advertisements and to gather statistics on site visitors. The only difference here is the subject of observation.

B. Content Manipulation

It would also be possible for provider A to directly manipulate B's code and data. Since there are no protection boundaries and every part of the global environment is accessible by all JavaScript code, all A needs to do is overwrite or override B's variables. One simple attack would be to delete B's content. This would deny B advertising revenue; however, it would also deny X's owner revenue as well. Another attack would be to rewrite displayed advertisements to make them less appealing, thus reducing B's click-through rate. A could also replace B's advertisements with A's.

A more sophisticated attack would be for A to adjust the attack based upon what code B has included. By walking the DOM, A's JavaScript can analyze any of B's code that was part the document. While imported JavaScript cannot be directly analyzed in this way, A's server can grab B's JavaScript file, analyze it, and return instructions on what variables to change to A's JavaScript in X—all by using a single GET request.

Of course, since there is no restriction on A's access to the page, A could choose to modify other parts of X. A could cause B to display inappropriate advertisements by inserting (hidden) content into the page. More sophisticated content-based attacks are also possible such as content censorship; to avoid detection, however, they would need to be very constrained and targeted.

C. Server Manipulation

Rather than manipulating B's JavaScript, A could instead send messages to B's servers using the state present in the web page. Note that A's JavaScript can do anything that B's JavaScript can do; thus, A can impersonate B's code to B's servers. A could do this to give false information about page X (e.g., misreport the content of X) to B, or to generate false clicks on B's advertisements (i.e., perpetrate click fraud).

V. Requirements

This section attempts to create a set of goals for web security solutions addressing the problem of content provider conflict. There are 4 goals: ease of use for all web page creators, clear adoption path, isolation between content providers, and flexibility for future innovation in web applications.

A. Ease of use for all web page creators

Few people would choose to design security solutions which are unusable, but sometimes it can be unclear who the intended users are and what skills they have. Early web mashup solutions such as Subspace [10] rely heavily upon skilled web programmers who could produce secure web pages. This makes sense when you are considering securing complex web applications, which are often created by skilled programmers. However, we have seen that there is risk any time code from

multiple providers is included on a page. Thus, any blog that contains cut and pasted ad code, videos, etc. could be at risk, and many people who use the simple cut and pasted code are not programmers at all.

The ideal solution to content provider conflict must take this common use case into account. For example, one could provide secured code which can be cut and pasted with the code fragments currently in use, or perhaps encourage tools which automatically generate more secure code. Or, perhaps, the solution lies in involving the page creator as little as possible since their skill set and preferred tools cannot be predicted.

B. Clear adoption path

Deployment of any solution is important, and a clear, feasible adoption path is needed to go past the research sphere into actual use on the web. The web is a distributed and heterogeneous environment, and it is the diversity that presents many problems to adoption. There are various types of web server and browser in use, controlled by many different individuals and organizations. Changing all of them at once is infeasible. Similarly, caution must be used when making changes to JavaScript or other web languages, and at least partial backwards compatibility or a way to deal with older websites could potential ease the pain of adoptions. The ideal solution would put some thought into these issues and examine ways in which deployment could be achieved.

C. Isolation between content providers

If the problem is that content providers have too much access to other content providers' code and content, then the solution is to limit this access by providing isolation between components. The ideal solution will block the three attacks described previously: observation, content manipulation, and server manipulation.

D. Flexibility for future innovation

Although it is difficult to predict the future, solutions should try to plan for it by giving flexibility and allowing for innovation in web applications. The current model, while it now seems overly permissive, has given us the ability to make applications that were unheard of when the web was created. The ideal solution would solve current use cases without limiting itself to only those known cases.

VI. Related Work

Although concerns about the security of JavaScript are as old as the language itself, only more recently have security researchers begun really exploring the kinds of attacks that JavaScript makes possible. While web security issues such as drive-by downloads [16], [15], cross-site scripting [13], and cross-site request forgery [11] do not require JavaScript, JavaScript does make these and other attacks more potent and easier to execute. Many security practitioners recommend that users disable JavaScript in their browsers entirely [2] or on a per-domain basis [8]; because so many pages require JavaScript to render correctly, such "solutions" are not practical for most users. Thus, even though such a solution would successfully block attacks, it can hardly be considered to be deployable.

JavaScript itself already has powerful mechanisms for code and data separation, as it has an environment-based lexical scoping model for variable and function binding. This model supports full closures, and thus provides very powerful mechanisms for code and data separation. These built-in abilities may be very helpful when it comes to finding a solution, but at the moment they are seldom-used. This is perhaps due to the fact that code is often written by those who have little understanding or interest in security. If things could be arranged so that using these mechanisms were the easiest route to writing JavaScript, it is possible that they would be utilized more effectively.

Some have argued for a more comprehensive approach to JavaScript security [18]; others have focused on scanning web pages for dangerous forms of JavaScript [19], [13], [11], [6], [21]. One limitation of code approaches is that they cannot restrict the regular behavior of external content providers such as ad servers even when they are potentially dangerous. Indeed some solutions must explicitly whitelist ad servers in order to achieve acceptable performance [21], [8]. As such, many techniques are unable to solve the problems of conflicting trusted content providers since they assume that all providers are trusted.

Currently there is an ongoing battle between advertisers and criminals. Advertisers regularly lose money to click fraud schemes in which criminals fake legitimate user behavior [9], and research indicates that some click fraud strategies can be extremely subtle and hard to detect [3]. Criminals are also taking advantage of the access ad servers have to regular users; indeed, even major companies such as DoubleClick [5] and Microsoft [17] have been tricked into distributing advertisements containing malware. In this environment it is clear that users have reason to be wary of ad servers. What we have argued here is that content providers such as ad servers need to be wary of each other as well.

A web mashup is an application that combines code and data from more than one source into a single integrated tool. In order to make a mashup application today, as shown in Figure 4a, JavaScript from multiple sites must be imported into a single environment. As explained in the previous section, this construction raises significant security issues.

Several researchers have proposed secure mechanisms for building web mashups. The goal of such systems is to achieve something equivalent to Figure 4b. Here, code and data from different sources are loaded into separate contexts; they neither share the same DOM objects, nor do they share the same namespace. Using the example from Section III, a function `foo` would be defined twice, once in the context from A and once in the context from B. For A and B to communicate, they must use well-defined communication channels using mutually agreed-upon protocols. Note that this architecture is virtually identical to that used by most distributed computing platforms.

Current secure web mashup proposals achieve separation and message passing either by requiring significant architectural changes to web applications [12], [10] or by using JavaScript language extensions that must be implemented in web browsers [7]. By themselves, these requirements make current secure mashup solutions unappealing for regular web pages. Even worse, these proposals would restrict the ability of content providers to access the body of the including page, thus breaking most popular site statistics services and context-sensitive advertisements. Thus, while secure web mashup mechanisms could be used to prevent content provider conflict, usability and functionality limitations would have to be addressed before they could be widely deployed.

VII. Discussion

When looking at the potential for conflict between external content providers, we do not mean to imply that we expect most providers to engage in open warfare on the web; instead, we are simply pointing out that there are conflicting agendas, and there is nothing to prevent those conflicts from manifesting as real attacks. While one type of content provider conflict, click fraud, is a significant problem, currently it is only perpetrated by known criminals. What happens, though, when legitimate businesses go bad?

The problem is somewhat analogous to that faced by companies that outsource to multiple partner companies. Each partner must be given access to the company's IT infrastructure; that access must be limited, however, in order to minimize the risks to the company. On the web, page authors outsource key parts of their "business" to outside parties, but do so without restricting their behavior. Social norms and legal measures can deter bad behavior to some degree; unfortunately, the global nature of the Internet means that we cannot rely upon individual societies or governments to enforce those norms. In operating systems, we long ago realized that memory protection made for more robust and secure systems. The question is, how do we bring analogous protections into the JavaScript sandbox?

As with most security problems, a point solution is not the answer; rather, layers of defense are required. Some defenses, such as code obfuscation and tamper resistance, could be deployed by the content providers themselves. Such measures, however, are partial at best; if conflict were to become common, an arms race would follow that would, at a minimum, degrade the experience of regular web users through broken and slowly executing web pages.

Work on more secure web mashups is an important step forward. However, the requirements for mashups and for content providers are different: where mashups require channels of communication between content in different frames, many content providers advertisements need to be able to analyze or even change the contents of a page in order to provide their services. We believe that protections from interactions between external content providers is an important area for future work.

One approach to providing such protections would be to adopt programming patterns and mechanisms to expose necessary content to external JavaScript without permitting unfettered access to the DOM. Another approach would be to enhance the browser such that it can automatically recognize and enforce appropriate boundaries between included JavaScript. Whatever the approach, the challenge is always to get the necessary buy-in from content providers, tool providers, web developers, and web users. Given the potential for problems, however, there may be sufficient motivation for a major change in the way web content is created and interpreted.

VIII. Conclusion

Including JavaScript code from multiple content providers is a potential source of security vulnerabilities. Such JavaScript can interact to allow surreptitious observation, content manipulation, and server manipulation. Although generally not a problem today, these interactions place external content providers at risk from both criminals and competitors. While there exist techniques for protecting against cross-site scripting, cross-site request forgery, and other web attacks, we lack mature methods for regulating interactions within the JavaScript sandbox. While work on protections for web mashups are an important step forward, further work is needed to find solutions that handle some very common use cases for the web, including context-sensitive ads and pages created by people who are simply pasting in code fragments provided by others (as opposed to created by skilled web programmers intending to create a complex web application). When creating these solutions, designers need to consider the needs of a wide range of web page creators, produce a clear adoption path so that their solution is not prohibitive to deploy, achieve separation between content providers who should not interact, and keep in mind not only current use cases, but also future innovation.

Acknowledgments

We thank the members of Carleton Computer Security Laboratory and the anonymous reviewers for their suggestions.

This work was supported by the Canada's National Sciences and Engineering Research Council (NSERC) through their Postgraduate Scholarship program (TO) and Discovery Grant program (AS & TW). In addition, Research in Motion (RIM) has provided support for our research in Web security.

References

[1] "Symantec internet security threat report," Symantec, Tech. Rep. XII, September 2007.

[2] CERT® Coordination Center, "Frequently asked questions about malicious web scripts redirected by web sites," CERT, Tech. Rep., 2004. [Online]. Available: http://www.cert.org/tech_tips/malicious_code_FAQ.html

[3] M. Gandhi, M. Jakobsson, and J. Ratkiewicz, "Badvertisements: Stealthy click-fraud with unwitting accessories," vol. 1, no. 2. Taylor & Francis, 2006, pp. 131–142.

[4] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2," in *USENIX Symposium on Internet Technologies and Systems*, 1997.

[5] D. Goodin, "Doubleclick caught supplying malware-tainted ads," *The Register*, November 13 2007.

[6] O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, 2005, pp. 85– 94.

[7] J. Howell, C. Jackson, H. Wang, and X. Fan, "Mashupos: Operating system abstractions for client mashups," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[8] InformAction, "Noscript." [Online]. Available: http://noscript.net/

[9] N. Ives, "Web marketers fearful of fraud in pay-per-click," *The New York Times*, March 3 2005.

[10] C. Jackson and H. J. Wang, "Subspace: Secure cross-domain communication for web mashups," in *Proceedings of the 16th International World Wide Web Conference (WWW2007)*, Banff, Alberta, May 8-12 2007.

[11] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *2nd IEEE Communications Society International Conference on Security and Privacy in Communication Networks (SecureComm)*. Baltimore, MD: IEEE Computer Society Press, August 2006.

[12] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama, "Smash: Secure cross-domain mashups on unmodified browsers," IBM Research, Tokyo Research Laboratory, IBM Japan, Ltd., Tech. Rep. RT0742, June 11 2007.

[13] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross site scripting attacks," in *The 21st ACM Symposium on Applied Computing (SAC 2006), Security Track*, Dijon, France, April 2006.

[14] T. Oda, G. Wurster, P. V. Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," School of Computer Science, Carleton University, Tech. Rep. TR-08-07, 2008.

[15] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser: Analysis of web-based malware," *Workshop on Hot Topics in Understanding Botnets (HotBots), April*, vol. 10, 2007.

[16] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us," Google, Tech. Rep. provos-2008a, February 4 2008.

[17] J. Reimer, "Microsoft apologizes for serving malware," *ars technica*, February 21 2007.

[18] C. Reis, S. Gribble, and H. Levy, "Architectural principles for safe web programs," in *Sixth Workshop on Hot Topics in Networks (HotNets) 2007*, 2007.

[19] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," in *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[20] B. Schneier, "Real story of the rogue rootkit," *Wired*, 2005.

[21] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, San Diego, CA, February 2007.