Architectural Principles for Safe Web Programs

Charles Reis, Steven D. Gribble, and Henry M. Levy
{creis, gribble, levy}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington

Abstract

Web content is migrating away from simple hyperlinked documents towards a diverse set of programs that execute within the web browser. Unfortunately, modern browsers do not provide a safe environment for running these web programs. In this paper, we show how current web security threats are symptoms of four key problems in supporting web programs: vague program boundaries, unwanted code, poor isolation, and inconsistent security policies. In response, we introduce abstractions for web programs and program instances, and we present a set of architectural principles to address these fundamental problems.

1 Introduction

Modern browsers must contend with a complex and hazardous web. Web content has been migrating away from simple hyperlinked documents towards a diverse set of programs designed to execute within the browser. Thanks to new mechanisms and coding techniques, these programs increasingly resemble their desktop counterparts: they have interactive user interfaces, they compose with other programs within the browser, and they communicate with remote servers to exchange data.

This shift demands a change in the way we think about web browsers, which must now act more like operating systems than document renderers. The focus of browser architecture should not only be on how to transport and render documents efficiently, but also on how to execute *web programs* safely.

In an ideal world, web browsers would provide a *safe* platform for running instances of web programs, each clearly defined and easy to manage. Like operating systems, web browsers would provide well-designed facilities for running program code, loading libraries, and obtaining data from authorized sources. Browsers would keep program instances separate from each other and from the rest of the client's resources. Threats to this platform may change over time, so browsers would enforce extensible policies on the behavior of web programs.

Today's web browsers are far from this ideal vision, however, and they do not provide a safe environment for running code from the web. There has been much recent attention to web security threats, from browser exploits [17, 18] to cross-site scripting attacks [15] to DNS rebinding [13]. These threats are symptomatic of four fundamental problems that become clear when viewing the web as a collection of programs:

- *Can't identify program boundaries* — A web program comprises more than a single document, yet it is difficult to say where its boundaries are. Current browsers use the notion of *origin*, which we argue is both inappropriate and insecure.
- *Can't prevent unwanted code* — It is currently difficult to distinguish web program code from data. As a consequence, web developers who integrate content from multiple sources risk giving adversaries control over their programs.
- *Can't isolate programs in the browser* — Separate web programs can interfere with each other in critical ways, making it unsafe to visit trusted and untrusted sites in the same web browser.
- *Can't apply uniform security policies* — There are many types of executable code that run in browsers. Each type defines its own security policies, potentially creating inconsistencies across types. The situation is exacerbated by browser extensions that can breach these policies.

The individual symptoms of these problems may be alleviated by point solutions, but the problems themselves will continue to cause new issues unless we rethink aspects of the web. Finding solutions requires looking at the web with a program-centric perspective, to improve the ways web programs are defined and change the ways browsers are architected. By addressing these fundamental problems, we can provide a much safer environment for browsing the web.

In the rest of this paper, we discuss architectural principles for web programs that can help resolve these issues. In Section 2, we discuss the fundamental problems and their symptoms. We identify a need for web program and program instance abstractions in Section 3, and we present principles for supporting these abstractions and show they can be achieved in incremental ways. We conclude in Section 4.

2 Web Programs are Unsafe

Current web browsers were built with documents in mind, not programs. Supporting active code within documents was an afterthought, and as a result, browsers continue to have inadequate mechanisms for safe code execution. In this section, we describe how the resulting security threats are symptoms of our set of four basic problems.

2.1 Can't identify program boundaries

Web browsers are expected to isolate the code and data of unrelated documents, yet it is difficult for browsers to determine the boundaries separating web programs from each other. This leads to many difficulties in managing and isolating web programs.

In today's browsers a web document is the basic abstraction, but web programs often comprise more than a single document. Sets of documents in the browser can fully access and modify each other through their Document Object Model (DOM) trees, exchange stored data through cookies, and receive data from particular servers. Documents with full access to each other are effectively part of the same program, while other documents are kept separate.

Current web browsers use the Same Origin Policy [19] to govern a document's access rights: two documents can access each other only if they arrive from the same origin. Yet the origin of a document is a mutable property; a document from `foo.bar.com` can change its origin to a suffix of that name (e.g., `bar.com`) by modifying its own `document.domain` property. This confounds reasoning about the boundaries of a web program, as they may change as the program executes. At best, one can conservatively define a program's boundaries by the second level domain name of the documents' origin.

At a deeper level, however, origin itself is a largely inappropriate and insecure way to define program boundaries. *First, the notion of origin may not match the desired boundaries of the program.* A single origin may be too narrow: pages often include code directly from a third party's server, mashups obtain data from multiple origins, and a secure session with a bank or similar party may span documents from multiple origins. Conversely, a single origin may be too broad: some social networking services host pages with user-supplied content that might be considered separate programs, yet providing a separate origin for each program may not always be feasible.

Second, the notion of origin can be abused and compromised. Recent studies of DNS rebinding attacks reveal that browsers can be tricked into believing two documents from different origins belong to the same origin [13]. This allows an adversary to cross the boundaries of any program defined by its origin.

These limitations make it difficult to identify a web program's boundaries in practice. Without clear boundaries, any notions of authorization, isolation, and security policies are difficult to enforce.

2.2 Can't prevent unwanted code

The current design of web programs makes it very difficult to distinguish between active code and passive data. Additionally, modern web programs integrate content from multiple sources, ranging from user input to data to code libraries from third parties. As a result, it is difficult to restrict such content to prevent it from gaining control over the web program itself.

Cross-site scripting (XSS) attacks are one widely acknowledged symptom of this problem. XSS vulnerabilities allow adversaries to inject scripts into another party's web program via user input, effectively gaining control of the program. Worse, attacks like the Samy XSS worm [1] are able to bypass many server-based security filters using syntactically incorrect script code, which slips through filters but runs in many browsers anyway. This makes XSS attacks difficult to prevent without support from the browser. Script injection can also occur via third party code libraries that change without warning (e.g., via a compromised server [12]) or ISPs that inject advertisement scripts into a page while it is in transit [2].

Mashups are also affected by this fundamental problem. The Same Origin Policy in browsers prevents web programs from retrieving *data* from third party servers, but it places no such restrictions on retrieving *code*. To integrate data from other sources, many mashups retrieve data in the JSON format [5], which is executable code presumed to only contain data. This is analogous to a traditional program loading input data by linking against a code library. As a result, adversarial data servers could easily include other code statements in JSON files to gain control of mashup web programs. Instead, safe mechanisms for obtaining data are necessary to support mashups without the risk of unwanted code.

These threats are indicative of a need to explicitly authorize the code for a web program, and a need to place limits on other code that might intentionally or unintentionally appear.

2.3 Can't isolate programs in the browser

The Same Origin Policy in current browsers is intended to prevent web programs from interfering with each other. Even if origins were adequate to infer program boundaries, though, there are still ample opportunities for web programs to interfere with each other. This interference can exhibit itself in three ways: leaking private information, taking control of another program, and causing disruptions or the loss of potentially unrecoverable state.

First, studies by Felten et al. [9] and Jackson et al. [14] show that adversarial web programs can learn about a client's browsing history using covert channels. For example, a web program might measure the time to load an object from another program to see if it has been cached, or it might query the color of links to another program to see if the latter has been visited. Such problems can leak information from a trusted web program to an untrusted one.

Second, browsers allow untrusted web programs to take control of other web programs by misusing the client's credentials. In an attack known as cross-site request forgery (CSRF) [25], an adversary's web program sends an action-inducing HTTP request to a trusted server, such as a bid request to an auction site. If the client has authenticated itself to the auction site already, the browser will send the client's credentials with the adversary's request. In this example, the auction site can use workarounds to mitigate this threat, but the underlying problem is a lack of isolation between the adversary's and the auction site's programs in the browser.

Third, interference between web programs can occur due to cross-program failures and resource contention. Current web browsers run multiple web programs within the same operating system process. As a result, any bug that causes a crash in the browser or its plug-ins will lead to the failure of all web programs running in that process. In this way, a failure in a poorly implemented or malicious web program could cause the loss of an unrecoverable session, such as a partially completed flight reservation. Resource contention can lead to similar interference, including CPU starvation or memory leaks that are difficult to attribute.

In all of these ways, web programs are less isolated than traditional programs in a modern operating system. If the web is to continue its progression towards supporting richer programs, achieving better isolation and robustness must be a high priority.

2.4 Can't apply uniform security policies

The code in web programs is often in the form of JavaScript embedded in HTML. However, browser plug-ins offer an increasing variety of code formats, where each plug-in defines its own security policies. Furthermore, extensions to the browser's user interface are not subject to the browser's security policies. There is no uniform interface for monitoring web program behavior, so it is difficult to reason about what web programs can and cannot do. Such an interface is important for applying security policies and reacting to newly discovered threats.

The widening range of possible content types shows one symptom of this problem. Flash movies, Java applets, and new formats such as Microsoft Silverlight each provide their own runtime environment for code embedded in web programs, and each has its own security model. For example, Flash movies have different rules than JavaScript for fetching data from another origin, and Java applets maintain their own DNS-pinning database independent of the browser's database [13]. Not only must each environment provide a sufficient security model on its own, but the combination of environments must not give adversarial programs unexpected capabilities.

Another symptom arises with unnecessarily powerful browser extensions. For example, Firefox extensions may contain arbitrary binary code, access all browser resources, and communicate across origins. Even worse, they may inadvertently allow web programs to gain these abilities, as in early versions of the popular Greasemonkey extension [26].

It is also difficult to introduce new security policies into this environment, due to the lack of an interposition layer on web program behavior. Work on BrowserShield [18] shows one possible use for new security policies: defending against initially unforeseen threats such as browser vulnerabilities. Introducing such policies into the browser itself is challenging without a way to uniformly interpose between web programs and the browser's runtime environments.

2.5 Summary

Overall, the lack of boundaries, presence of unwanted code, poor isolation, and inconsistent policies in web programs leave browsers hampered with safety concerns. The symptoms of these problems are now surfacing in many popular applications, from XSS attacks on MySpace [1] and Yahoo Mail [3] to CSRF vulnerabilities on Gmail [23]. Unfortunately, the current fixes tend to be mere band-aids, such as strengthening input validation on a web server. These point fixes do not address the fundamental problems and further symptoms will continue to arise. In the next section, we discuss ways to resolve the fundamental problems by changing how web programs are defined and how web browsers are architected.

3 Principles for Web Programs

Today's browsers are architected with a *server-based* point of view in mind: browsers merely provide a remote "terminal" into programs running on servers. For example, the Same Origin Policy restricts what programs can do based on their originating server, and recent proposals to allow cross-origin reads [6, 22] reinforce this view by allowing a web program to read a network resource only if its origin server is explicitly authorized to access it.

However, this server-based view of web programs is increasingly inaccurate. First, a web program is no longer controlled by a single server, but instead is a composition of code and data from multiple servers. Second, modern web programs contain substantial amounts of client-side code; by visiting a web page, a browser will instantiate and execute this code on behalf of the program. The server-based view overlooks these realities, conflating the notions of *program*, *program instance*, and *origin*.

We claim that today's web demands a new perspective — one that is based on *programs* and *program instances* as first class abstractions. A web program is an entity defined by one or more servers, and a program can have one or more instances running within a browser. These abstractions are not the same as an origin, and they can be used to solve the problems we discuss in Section 2.

In the rest of this section we discuss the following four architectural principles, which we derive from our list of problems:

- Web programs and program instances must have clear boundaries on the network and in the browser.
- It must be easy to specify which code is authorized to run in a web program and to impose limitations on this code.
- Instances of programs must be isolated in the browser, to prevent interference at the client.
- The behavior of program instances must be governed by browser-level policies, independent of content types and browser extensions.

By adhering to these principles, we believe browsers can become a safe platform for web programs.

3.1 Program Boundaries

We argue that a solution to the web's current safety problems must include abstractions for web programs and instances of such programs. These abstractions are essential for defining boundaries around resources in the browser, and they provide building blocks for solving the remaining safety problems.

A *web program* must be a named entity that can be controlled by its authors. It is comprised of documents, code, data, and embedded objects from one or more servers. Browsers must be able to identify which resources belong to which program, and whether to fork a new program for any given request. Program boundaries can be derived by attributing the name of the program to each document associated with it. Within those documents, programs may embed code and objects from any source to which the program has access rights.

A *web program instance* must be an entity managed by the browser, containing resources related to a single web program and possessing an identifier similar to an OS process ID. An instance is created when the user independently navigates to a new program's document or when a program instance requests a document from another program. The instance then comprises all resources requested from within the instance. Importantly, all requests initiated by a web program instance must be marked with the program name and instance identifier, to ensure that servers can control access to their resources appropriately.

An appropriate system for *naming* programs is an open challenge. We have seen that origins are inadequate, but other techniques are possible. One potential solution is to equate program authorship with knowledge of a secret, such as an asymmetric key pair. The program author could generate a key pair, in which the public key names the program and the private key provides proof of membership. Such a scheme could support a single program with documents from multiple origins (by sharing keys) or multiple programs within the same origin (by generating multiple key pairs). No public key infrastructure would be required: keys merely indicate membership in a given program.

Importantly, such a solution can be deployed incrementally. That is, browsers can fall back to the

Same Origin Policy for those sites that do not define explicit boundaries for their web programs.

3.2 Authorized Code

A web program must be free to compose code, data, and other objects from multiple sources, but it is critical that the program's authors have the ability to authorize and place limits on any code that runs in the program. Specifically, web programs must have the ability to specify an exclusive set of code, so that adversaries cannot inject new code to gain control of the program. Moreover, web programs that compose code from multiple sources must have the ability to limit the access rights of such code toward the programs' resources. Finally, web programs must have the ability to request data from multiple sources safely, without resorting to code formats like JSON.

Recent research has begun to explore these requirements. To identify which scripts in a document are authorized, BEEP proposes that browsers ignore any script whose checksum does not appear in the document's whitelist [15]. Such a whitelist could block all injected scripts, even those inserted while a page is in transit. Additional authorization mechanisms are possible: a program could require all code fragments to contain proof of authorization, such as a signature from the program's key pair.

Researchers have begun to explore limitations for web program code as well. Including code libraries from multiple sources is now a popular practice (e.g., mashups often include code from Google Maps). To prevent such embedded libraries from usurping control of the program, MashupOS has proposed ways to restrict code, using sandboxes, separate address spaces, and messaging primitives [24].

Such restriction mechanisms could take many forms, such as limiting access to DOM subtrees or restricting specific access rights (e.g., network access). Checksums and versioning may also prove useful for library code, to ensure that publishers can test against a known version without fear of errors if the library changes. To specify restrictions on code, web programs could include attributes on script tags, new types of tags [7], or even program-level manifest files [4]. The most effective techniques for restricting web program code remains an open problem.

Importantly, web programs must have the ability to safely access data from multiple sources, without resorting to the use of code formats to exchange data. Both the JSONRequest proposal [6] and a Mozilla-supported W3C draft [22] offer mechanisms for retrieving non-executable data from other origins. To prevent malicious programs from accessing private client resources, these proposals force the data provider to acknowledge the origin of the request. Rather than relying on origins for access control, however, we would require such requests to be marked with the program name and instance identifier.

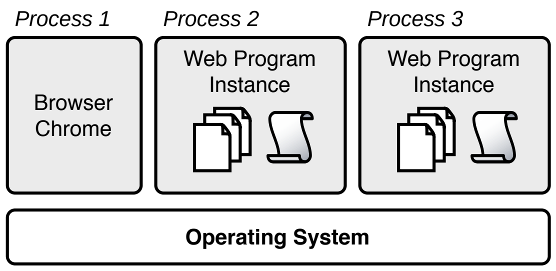

Figure 1: Browsers should isolate web program instances with OS processes, while keeping the browser user interface ("chrome") in its own process.

Overall, common practices on the web indicate that code authorization, code restrictions, and safe data requests are important features for supporting modern web programs. Like program boundaries, they can be incrementally deployed: browsers can continue to permissively run script code on pages that do not define restrictions, while web programs that employ the new mechanisms can be safer from attack.

3.3 Program Isolation

To safely run web programs in the browser, the client-side effects of each program instance must be independent. Having an explicit *program instance* abstraction allows us to support more comprehensive isolation than current browsers, both for security and robustness. This must include isolating at least *(1)* the information stored by each program, *(2)* the credentials acquired by each program instance, and *(3)* the OS resources used by each program instance.

For isolating information between web programs, Felten et al. and Jackson et al. propose specific fixes for the covert channels they discover, isolating the browser's cache and visited link information between web programs [9, 14]. Jackson offers a further invariant for this isolation: "only the site that stores some information in the browser may later read or modify that information" [14]. We support this broader invariant, though we equate "site" in this proposal with a *web program.* Enforcing such isolation between web programs is important to preserve the user's privacy, though this stored information can be safely shared between instances of the same program.

Credentials must be isolated between *program instances* to prevent their misuse by malicious web programs. For example, many current web programs use session cookies (i.e., cookies that expire when the browser exits) to authenticate clients. Browsers pass such cookies on every request to the server that created them, even on requests made by other web programs. To prevent CSRF attacks, we claim that browsers must isolate session cookies within each program instance. In recent work, RequestRodeo uses similar ideas to restrict the flow of credentials [16], though it lacks the notion of program instance.

Furthermore, browsers must better isolate web programs to improve their robustness. We argue this can be achieved using support from operating system services that already exist. In particular, each *program instance* in the browser should map to an OS process. As with traditional programs, processes can prevent crashes or failures in one web program instance from affecting other instances. Processes can also ensure that one instance cannot starve others of CPU time, and they allow users to easily attribute memory leaks to the offending instance. To achieve this, the browser's user interface and navigational features should reside in one process, while each instance's runtime environment should reside in a process of its own, as shown in Figure 1. Processes might not securely isolate web programs in the case of compromised browsers, however, so stronger sandbox mechanisms may also be desirable [4, 10, 11].

With such isolation in place, browsers would provide a far safer environment for running trusted and untrusted web programs side by side.

3.4 Security Policies

Browsers must uniformly apply policies to web program instances and the code they run, regardless of the content type of the code. This is necessary for reasoning about what programs can and cannot do in the browser. To achieve this, browsers must provide appropriate interfaces for interposing on web program behavior. They must also allow extensible security policies to be registered with these interfaces, to support new policies that react to unforeseen threats.

As shown in Figure 2, we envision an interposition layer in the browser that can govern a program's access to the DOM, the network, other web program instances, and other browser resources. Such a layer must apply uniformly to each content type, so that a single set of policies can govern JavaScript, Flash, and other types of code. Global policies might range from preventing access to LAN resources [21] to blocking exploits of recently discovered vulnerabilities [18]. Program-specific policies might restrict or modify certain program instances [8]. Such policies would be easy to apply if program instances were properly isolated, as we propose in Section 3.3.

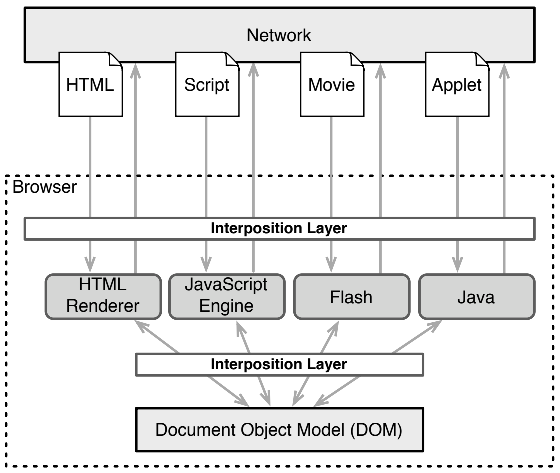

Figure 2: Browsers should provide interfaces to interpose on a web program's behavior and its access to the DOM and network, regardless of content type.

Most browser extensions should also be subject to policies via the interposition layer. Some extensions may legitimately require elevated privileges to accomplish their goals, but we propose that the principle of least privilege [20] be applied to the vast majority of extensions. For example, a weather report widget in the browser chrome need not access the DOM trees of all web programs, while ad blockers need not modify the browser's address bar. We propose creating a class of "safe" browser extensions that either gain no additional privileges over the web programs they modify, or that declare the privileges they need.

With a proper interposition layer and sufficient restrictions on browser extensions, end users could safely reason about the effects of visiting any given web program in their browsers. To reach this goal, browsers might sandbox or wrap existing plug-ins like Flash and Silverlight. This would allow interposition on their calls to both browser APIs and the operating system, much like Janus [11] or Ostia [10].

4 Conclusion

Content on the web has changed substantially, from documents into interactive programs. In this paper, we have argued that the numerous threats to safe web browsing are the result of inadequate support for these *web programs*. Unclear program boundaries, intermingled code and data, poor isolation between programs, and inconsistent security policies have all contributed to an environment rife with dangers.

We have argued that these fundamental issues must be addressed, rather than continuing to provide point fixes for their symptoms. Proper notions of programs and program instances should replace the Same Origin Policy. Code should be clearly delimited from data, and browsers should support facilities for authorizing code, restricting libraries, and exchanging data. Browser architectures should be refined to fully isolate web program instances, and browsers should offer uniform interposition on the behavior of web programs, regardless of content type.

With these changes, the behavior of web programs can be more predictable and manageable. The current symptoms can be resolved, and users can have a safer environment for running both trusted and untrusted web programs in their browsers.

Acknowledgments

This research was supported in part by the National Science Foundation under grants CNS-0132817 and CNS-0614975, by the Torode Family Endowed Career Development Professorship, and by the Wissner-Slivka Chair.

References

[1] Technical explanation of the MySpace worm. `http://namb.la/popular/tech.html`, 2005.

[2] NebuAd / Service Providers. `http://www.nebuad.com/providers/providers.php`, Aug. 2007.

[3] C. Babcock. Yahoo Mail Worm May Be First Of Many As Ajax Proliferates. `http://www.informationweek.com/security/showArticle.jhtml?articleID=189400799`, 2006.

[4] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[5] D. Crockford. Introducing JSON. `http://www.json.org/`, 2006.

[6] D. Crockford. JSONRequest. `http://www.json.org/JSONRequest.html`, 2006.

[7] D. Crockford. The module Tag. `http://www.json.org/module.html`, 2006.

[8] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-End Web Application Security. In *HotOS XI*, 2007.

[9] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *ACM Conference on Computer and Communications Security*, pages 25–32, 2000.

[10] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *USENIX Security Symposium*, July 1996.

[12] J. Grossman. Cross-site scripting worms and viruses. `http://www.whitehatsec.com/downloads/WHXSSThreats.pdf`, Apr. 2006.

[13] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *ACM CCS*, Oct. 2007.

[14] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting Browser State from Web Privacy Attacks. In *WWW*, May 2006.

[15] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW*, May 2007.

[16] M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In *OWASP Europe Conference*, May 2006.

[17] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. SpyProxy: On-the-fly Protection from Malicious Web Content. In *Usenix Security*, Aug. 2007.

[18] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *OSDI*, Nov. 2006.

[19] J. Ruderman. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[20] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Communications of the ACM*, 17(7), 1974.

[21] SPI Labs. Detecting, Analyzing, and Exploiting Intranet Applications using JavaScript. `http://www.spidynamics.com/assets/documents/JSportscan.pdf`, 2006.

[22] A. van Kesteren. W3C: Enabling Read Access for Web Resources. `http://www.w3.org/TR/access-control/`, June 2007.

[23] J. Walker. CSRF Attacks or How to avoid exposing your GMail contacts. `http://getahead.org/blog/joe/2007/01/01/csrf_attacks_or_how_to_avoid_exposing_your_gmail_contacts.html`, Jan. 2007.

[24] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *SOSP*, Oct. 2007.

[25] P. Watkins. Cross-Site Request Forgeries. `http://www.tux.org/~peterw/csrf.txt`, 2001.

[26] S. Willison. Understanding the Greasemonkey vulnerability. `http://simonwillison.net/2005/Jul/20/vulnerability/`, July 2005.