The Most Dangerous Code in the Browser

Stefan Heule[1] Devon Rifkin[1] Alejandro Russo*[2] Deian Stefan[1]
[1]Stanford University [2]Chalmers University of Technology

ABSTRACT

Browser extensions are ubiquitous. Yet, in today's browsers, extensions are the most dangerous code to user privacy. Extensions are third-party code, like web applications, but run with elevated privileges. Even worse, existing browser extension systems give users a false sense of security by considering extensions to be more trustworthy than web applications. This is because the user typically has to explicitly grant the extension a series of permissions it requests, e.g., to access the current tab or a particular website. Unfortunately, extensions developers do not request minimum privileges and users have become desensitized to install-time warnings. Furthermore, permissions offered by popular browsers are very broad and vague. For example, over 71% of the top-500 Chrome extensions can trivially leak the user's data from *any* site. In this paper, we argue for new extension system design, based on *mandatory access control*, that protects the user's privacy from malicious extensions. A system employing this design can enable a range of common extensions to be considered *safe*, i.e., they do not require user permissions and can be ensured to not leak information, while allowing the user to share information when desired. Importantly, such a design can make permission requests a rarity and thus more meaningful.

1 INTRODUCTION

The modern web browser is one of the most popular application platforms. This is, in part, because building and deploying web applications is remarkably easy and, in other part, because using such applications is even easier: a user simply needs to type in a URL to run sophisticated applications, such as document editors, email clients, or video players. Unlike venerable desktop applications, these *apps* run on many different devices without imposing painstaking installation procedures or forcing users to be concerned with security—e.g., the weather app stealing their banking data or locally-stored photos.

As the web evolved to address different application demands, it did so in a somewhat security-concious fashion. In particular, when adding a new feature (e.g., offline caching [22]), web browsers have been careful to ensure that the feature was confined to the browser, i.e., it did not unsafely expose underlying OS resources, and that it could not be used to violate the *same-origin policy (SOP)* [5, 29]. The SOP roughly dictates that an app from one origin can only read and write content from the same origin. This ensures that one app cannot interfere with another—it is the reason the weather app cannot read data from the tab running the banking app.

Unfortunately, the web platform has some natural limitations. Despite prioritizing "users over [app] authors [27]," a user's experience on the web is largely dictated by the app author. For example, the web does not provide users with a means for removing advertisements served by an app. Similarly, the user cannot directly share content from one app with another app of their choosing without the app author offering such a service. Of course, it is unrealistic to demand that app authors provide such features since they may be at odds with the authors' goals (e.g., to serve ads).

To address the limitations of the web platform, most modern browsers provide users with *extensions*. Extensions are typically used to modify and extend web application behavior, content, and display (style). For example, Adblock Plus [1], one of the most-widely used extensions, modifies apps by blocking network requests and hiding different page elements to provide ad-free browsing. However, extensions can also be used to implement completely new functionality. For instance, LastPass [2] allows users to store and retrieve credentials for arbitrary apps, in the cloud. And, in some cases, extensions even modify and extend the browser itself.

Unlike web applications, which are bound by the SOP, extensions can access the page contents of different-origin apps, perform arbitrary network communication, inspect and modify browser history, etc. Misusing such privileged APIs is a serious security concern. In light of this, browsers vendors have imposed various restrictions. For example, Chrome—which has the most comprehensive extension security system—makes it difficult to install extensions that are not distributed through its official Chrome Web Store (CWS), requires users to grant extensions access to use privileged APIs, and employs various mechanisms to prevent privilege-escalation attacks [6, 8, 21].

Unfortunately, even Chrome's extension system has fundamental shortcomings. For example, Chrome's attacker model assumes that extensions are not malicious, but rather that they are *benign-but-buggy* [6]. As a consequence, Chrome's security mechanisms were designed to prevent attacks wherein malicious app pages try to exploit vulnerable extensions. However, the system does

*Work conducted while at Stanford University.

not provide a way for protecting sensitive app data from extensions—a malicious extension can easily leak data. And the premise for placing more trust on extension code over web app code is unfounded: both are provided by third-party developers, while the former runs with elevated privileges. Unlike other privileged code in the browser (e.g., plugins), these JavaScript-based extensions are made available to users without a code review process. It is of no surprise that roughly 5% of the users visiting Google have at least one malicious extension installed, as showed by a recent study [7, 26].

Unfortunately, in the current extensions system, even trustworthy but vulnerable extensions can be exploited by malicious pages to leak sensitive data from cross-origin apps [20]. While Chrome's mechanisms limit an attacker to abusing the privileges held by the vulnerable extension, developer incentives have led many extensions request broad privileges. Similarly, many users have become desensitized to the install-time warning accompanying these extensions [12]. For example, of the 500 most popular Chrome extensions, over 71% request the privilege to "read and change all your data on the websites you visit." Since these extensions also retain their privileges throughout lifetime of the extension, this makes them especially attractive targets to attackers that wish to steal user-sensitive data.

In today's browsers, extensions are arguably the most dangerous code to user privacy. Yet, this need not be the case. This position paper argues for new extension system designs that can address user privacy without giving up on desired extension functionality.

What might such an extension system look like? Given that apps handle sensitive data such as banking information and that extensions are written by potentially untrusted third-party developers, it is clear that apps need to be protected from extensions. At the same time, it is important to keep protecting extensions from apps, as extensions may run with higher privileges. Mandatory access control (MAC)-based confinement [19, 25] naturally fits this scenario of mutually distrusting parties, where apps and extensions can be protected from one another.

But, MAC alone is not enough. While MAC-based confinement can prevent an extension from leaking sensitive app data even after it has access to it, for many extensions, this is overly restricting. For example, the Google Dictionary extension [3] needs to read text from the page and communicate with the network when looking up a word—its functionality relies on the ability to "leak" data. Hence, the extension system should allow users to explicitly share app data with extensions, which may further share the data with a remote server. Similarly, it should provide robust APIs that common extensions can use to operate on sensitive information without being confined. Together with MAC-based confinement this can alleviate the need for permissions altogether for a broad range of *safe* extensions: many extensions only read sensitive data and provide useful features to the user, but never disseminate the data without user intent.

Of course, leveraging user actions to share data is not possible in all cases; user-approved permission may still be necessary. However, these permissions should be fine-grained and content-specific. Since many extensions are safe and do not rely on special permissions, it would be possible for the extension system to give users more meaningful messages and warn them appropriately about installing dangerous extensions.

In the rest of the paper we give a brief overview of Chrome's extension system and its limitations (§2). We then expand on the design goals of an extensions system that addresses Chrome's limitations (§3) and describe a preliminary system design that satisfies these goals (§4). Finally, we conclude (§5).

2 Chrome's Extension Model

In this paper, we focus on the Chrome extension model, whose security system is widely regarded as being more advanced than those implemented in other browsers [8, 16]. More specifically, we focus on JavaScript-based extensions; we do not consider plugins, which can additionally execute native code.[1] Below we describe the extension system's security model, evaluate the use of permissions in this ecosystem, and highlight its key limitations.

2.1 Security Model

The Chrome attacker model assumes that extensions are trustworthy, but vulnerable to attacks carried out by apps [6]. Hence, Chrome's extension security system is designed to protect extensions from apps. Chrome requires developers to *privilege-separate* [21] extensions into a *content script* and a *core extension*. Content scripts interact directly with web pages (e.g., by reading the page's cookies or modifying its DOM),[2] but do not have access to any privileged APIs. To perform privileged operations, content scripts use message-passing to communicate with core extension scripts, which have access to the privileged APIs needed to perform the actions.

To mitigate the impact of exploits that compromise vulnerable content scripts, in addition to privilege separation, Chrome also follows the *principle of least privilege* [23]. Specifically, Chrome implements a permission system that can be used to limit the privileges available to core script extensions. By limiting the privileges of an

[1] Plugins make up only a small fraction of the space, require a code review before being put on the CWS, and are widely-accepted to be dangerous. We do not discuss them further.

[2] Actually, Chrome employs *isolated worlds* [6] to separate the JavaScript heaps of the content script and page. This prevents attacks where a malicious page redefines functions (e.g., `getElementById`) that are commonly used by extensions.

extension, the damages that can be caused from exploits is also more limited.

To this end, Chrome requires extension authors to statically declare, in a *manifest*, what kind of permissions the extension requires. In turn, the user must approve these permissions when installing the extension. Since the compromise of an overly-privileged extension can cause serious harm (e.g., leaking user's banking information), Chrome encourages developers to only request minimal privileges. Below, we report the results of our study evaluating permission usage in Chrome extensions.

2.2 Permission Study

We surveyed the permissions used by the 500 most popular Chrome extensions [14] by inspecting their manifest files.[3] Most extensions are widely deployed: the most popular extension is used by more than 10 million users; the 500th extension is used by more than 76,000 users. In Table 1, we list the permissions most often requested. The most widely required permission is `tabs`, which among other abilities, allows an extension to retrieve URLs as they are navigated to. More concerning is the prevalence of permissions such as `http://*/*`, `https://*/*` and `<all_urls>`, which allow an extension to make requests to any origin (over HTTP, HTTPS, or both, respectively). Upon installing any extension that requires one of these permissions (or several other similarly high-privilege permissions), the user is warned that the extension can "read and change all [their] data on the websites [they] visit." These permissions can easily be used maliciously, for example, to retrieve a sensitive webpage (using the cookies stored in the browser) and forward its contents to the attacker's own server. Despite this danger, permissions triggering this warning are widely used. In our study, we found more than 71% of the top 500 extensions display this "read and change..." warning at installation-time. For users installing popular Chrome extensions, the norm is to allow for such high privilege requests. In fact, the more popular extensions are more likely to show this warning: 74% of the top 250 extensions display this warning, 82% of the top 100, and 88% of the top 50. We did not investigate how many of these extensions actually needed or exercised their requested permissions.

2.3 Pitfalls

Chrome assumes extensions to be benign-but-buggy [6]. Unfortunately, this trust in extensions is amiss, as extensions are written by potentially untrusted developers. For example, Kapravelos et al. [18] report on 140 malicious extension in the CWS. While taking malicious extensions out of the CWS is an appropriate response, this weak-attacker model has unfortunately led to the design of security mechanisms that do not explicitly protect web app data. This is particularly disconcerting because, as our study shows, most extensions can access highly sensitive data and communicate with the network—vulnerabilities in such extensions can be used to leak user data [9, 20]. A single compromised or malicious extension is enough to put the users privacy at risk.

[3]The manifests in this study were fetched on April 20, 2015.

Permission	Count	Permission	Count
`tabs`	*75.6%*	`webRequestBlocking`	*25.6%*
`storage`	*38.4%*	`cookies`	*24.6%*
`http://*/*`	*37.8%*	`unlimitedStorage`	*20.4%*
`https://*/*`	*36.4%*	`<all_urls>`	*19.2%*
`contextMenus`	*36.0%*	`webNavigation`	*16.6%*
`webRequest`	*32.2%*	`management`	*14.6%*
`notifications`	*30.4%*	`history`	*10.4%*

Table 1: The 14 most prevalent permissions as required by the top 500 Chrome extensions. A single extension may request any number of permissions. A full list explaining what each permission grants is available in [13].

Chrome provides a permission system that is meant to implement least privilege. Unfortunately, the explanations accompanying the permissions are broad and content-independent. Moreover, they do not convey to the user why such permissions are justified. Permissions must be accepted at install time, before a user has acquired context from using the extension.[4] Because the vast majority of extensions require many broad permissions, users have grown desensitized and accustomed to accepting most permission requests.

The other pitfall of the extension system is that it makes it difficult even for security conscious extension developers to request minimal privileges. The permissions are coarse grained and Chrome does not provide a way for requesting finer-grained access. Even worse, developers are incentivized to ask for more permission than they actually need. For example, if an extension update requires additional permissions (e.g., because of a new feature in the extension), Chrome automatically disables the extension until the user approves the new permissions. Since users get irritated by such prompts, developers often ask for more permissions than necessary up front thereby eliminating the risk of removal.

3 Design Goals

In this section, we outline a series of design goals that a modern browser extension system should strive for in order to protect user privacy and avoid Chrome's pitfalls.

1. **Handle mutually distrusting code** Extensions and web apps may be written by mutually distrusting parties. In addition to protecting extensions from untrusted app

[4]Chrome more recently added *optional permissions*, which, while still declared statically, only demand the user's approval at run-time, e.g., right before the extension uses the privileged API. Unfortunately optional permission warnings fall victim to the coarseness of the system and often ask for far more expansive abilities than required.

code, an extension system should provide mechanisms for protecting sensitive user (app) data from untrusted extensions without giving up on functionality. We assume a relatively strong attacker model where an extension executes attacker-provided code in attempt to leak user data via the extension system APIs. We consider leaks via covert channels to be out of scope.

2. **Leverage user intent** An extensions system should leverage user intent for security decisions. The system should provide APIs and trusted UIs for making security decisions part of the user's work-flow. For example, browsers can use user intent to make sharing of app data with an extension explicit via a sharing-menu API. A challenge with this goal is designing APIs and UIs that are not susceptible to *confused deputy attacks* [17].

3. **Provide a meaningful permission system** Most common extensions should not need to request user permission to perform their tasks. In the rare case that an extension requires to leak sensitive data without explicit user intent, the permissions available should be fine-grained and content-specific. Furthermore, the system should provide the user with specific-enough information necessary to make an educated decision. This could happen, for instance, by asking for permission at runtime when the leaked content can be shown and the user has an idea of what the extension is about to do (in contrast to install-time permissions).

4. **Incentivize safety** The incentives of developers and the security model should align such that most common extensions are safe, i.e., they run without requiring user approval for permissions. The extension system should reward developers that implement these and other least-privileged extensions and penalize overly-privileged ones. For example, extensions that require permissions should require a security audit before being allowed to be installed. This ensures that APIs that leverage user intent for disclosing data are prioritized and that security warnings remain meaningful.

4 Preliminary Design

In this section we propose a new extension system designed to meet the aforementioned goals. We observe that, for an extension to be useful, it typically needs to have access to sensitive data such as the current app's URL or different parts of the page. However, if this information cannot be arbitrarily disseminated (within or external to the confines of the browser), then the user's privacy is not at risk: it is entirely safe for an extension to read sensitive data as long as it does not write it to an end-point that is not permitted by the SOP.

This idea of allowing code to compute on sensitive data, but restrict where it can subsequently write it, is endemic to MAC-based confinement systems (e.g., HiStar [28] and COWL [25]). In such systems, the sensitivity of information is tracked throughout the system and the security mechanism ensures that leaks due to data- and control-flow cannot occur.

We propose to extend the Chrome architecture to use a coarse-grained confinement system, similar to COWL [25]. As in Chrome, to achieve isolation and protect an extension from an untrusted app, every app and extension runs in a separate execution context. However, and unlike Chrome, our proposed extension system additionally protects app user data from an untrusted extension by ensuring that whenever the extension accesses sensitive data, its context gets "tainted" with the app's origin—we consider any data in the page to be sensitive. In turn, the origins with which the extension can subsequently communicate with is restricted by this taint—e.g., the extension cannot perform arbitrary network requests once it has read sensitive data.

With confinement, extensions that only read sensitive information can be implemented securely and without requiring any permissions. For instance, consider the Chrome extension Google Mail Checker [15], which displays an icon in the browser with the number of unread emails in Gmail. Confinement allows this extensions to connect to Gmail using the users credentials. Once it does so, however, the execution context is tainted with `mail.google.com` and thus cannot communicate with, for instance, `evil.com`. However, the extension can safely do its job and show an unread count to the user. We remark that such a system satisfies our goal of protecting user data against malicious extensions—even if malicious, the extension cannot leak the user's emails.

Of course, not all extensions are this simple, and a real extension system must provide extension developers with APIs to carry out common tasks. Below we describe some of these APIs, and in particular focus on APIs that make the confinement system more flexible or address our design goals directly.

Page access Some extensions read and modify page contents. Our system provides content script extensions with APIs for reading and writing the DOM of a page, much like COWL's labeled DOM workers [25]. Importantly, when accessing the DOM of a page, the content script is tainted with the origin of the page and its functionality is subsequently restricted to ensure that the read information is not leaked. (Of course, an extension can create such labeled content scripts at run time, to avoid over-tainting [25].) To ensure that extensions cannot leak through the page's DOM, we argue that extensions should instead write to a *shadow-copy* of the page DOM—any content loading as a result of modifying the shadow runs with the privilege of the extension and not the page. This ensures that the extension's changes to the page are isolated from that of the page, while giving the appearance of a single layout.

Explicit sharing Of course, some functionality requires sensitive data to be "leaked." For instance, Evernote Web Clipper [10] offers the functionality to save part of the page (e.g., the current selection) to `evernote.com`. Sharing page contents with arbitrary origins violates confinement—the page may contain sensitive information (e.g., a bank statement).

However, in some cases, the user may wish specific information to be sent to `evernote.com`, e.g., to save a recipe the user saw online. Confinement systems typically require *declassification* to allow such controlled leaks. However, extensions cannot be trusted to declassify data on their own. Our key insight is that information sharing typically follows a user action (e.g., clicking a "Save to Evernote" button), and therefore the intent of the user can be used to declassify the data. Concretely, we propose a sharing API that extensions can use to received data from the user and trusted browser UIs that users can employ to share specific content with these extensions, e.g., by means of a "Share with…" context menu entry. With this API, extensions like Evernote or Google Dictionary can be implemented without requiring specific declassification permissions. Of course, they can only leak data the user shares explicitly.

Encrypted sharing Since credentials are usually treated with more care than other data, our sharing API does not allow extensions to receive credentials without restrictions. Concretely, when sharing credentials, our sharing API provides extensions with *labeled blobs* [25], which the extension can only observe by tainting its context. However, it is often useful to allow extensions to synchronize and store such sensitive data. For this, we propose an API that takes a blob labeled with `a.com` and returns an unlabeled encrypted blob. This directly allows the extension to send the encrypted data to the cloud and synchronize it to another devices. There, a similar extension can use our API to decrypt the data to `a.com`-labeled credentials, which can then be used to, for example, fill in a `a.com` login form. This API makes our MAC system more flexible and directly allows the implementation of an extension to manage user passwords similar to LastPass. Unlike LasPass, however, the encryption algorithms and parameters are provided by the browser, only relying on the user to supply a master key.

Privileged content sharing Some extensions need to read content from the page and communicate with the network without user interaction. For example, the Reddit Enhancement Suite (RES) [4] fetches images that are linked in a post as to display them inline. Unfortunately, the page access API is insufficient when implementing such extensions since the code cannot communicate with arbitrary domains, as to fetch images, once it traverses the DOM to find the image links. Instead, we provide APIs that can be used to retrieve content from the page without imposing confinement restrictions. In particular, extension developers can request to access different kinds of elements on the page, e.g., URLs, or the current origin, etc. Our extension system would, in turn, ask the user to consent to the request at run time when the extension requests the data, applying the lessons and techniques of [11] to avoid desensitization (e.g., use different icons and colors to signify the severity of the request). Unlike Chrome's permissions requests, we envision providing users with content-specific choices (e.g., "RES wishes to see *all* the *links* on this page."), which they can also deny while continuing to use the extension—extensions should gracefully handle exceptions from these APIs or risk removal from the platform. Besides content-specific messages, other HCI techniques would be employed to make permissions more meaningful and to refrain users from blindly consenting to security prompts [24].

We remark that other APIs (e.g., a network API or declarative CSS replacement API) share many similarities with the above: they are fine-grained, content-driven, and abide by MAC. More interestingly, we note that our MAC-based approach encourages safe extensions, i.e., extensions that do not rely on privileges and raise alarms, but, rather, rely on sharing menu- and crypto-APIs to get user data. But in cases where permissions are required, our system presents security decisions at run-time and in terms of data—by reasoning about the content being disclosed users can make more educated decisions.

5 Summary

We identify extensions as some of the most dangerous code in the browser and show the pitfalls of modern extension security systems. For this reason, new extension security models that protect user privacy are in need. We outlined the goals of such as system and proposed a preliminary system design to this end. Our proposal relies on MAC-based confinement to prevent sensitive information from being arbitrarily shared. We also outlined several APIs that can be used to safely share data and make such a system flexible enough to handle a large class of common extensions, while keeping developer incentives aligned with security. We hope this encourages browser vendors to rethink extensibility.

Acknowledgements

We thank James Mickens, Petr Marchenko, Adrienne Porter Felt, and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, by a gift from Mozilla, by the Swedish research agency VR and the Barbro Oshers Pro Suecia Foundation.

REFERENCES

[1] Adblock Plus – surf the web without annoying ads! https://adblockplus.org/, 2012. Visited April 21, 2015.

[2] LastPass password manager. https://lastpass.com/, 2012. Visited April 21, 2015.

[3] Google dictionary. https://chrome.google.com/webstore/detail/google-dictionary-by-goog/mgijmajocgfcbeboacabfgobmjgjcoja, 2015. Visited April 21, 2015.

[4] Reddit enhancement suite. http://redditenhancementsuite.com/, 2015. Visited April 21, 2015.

[5] Adam Barth. The web origin concept. https://tools.ietf.org/html/rfc6454, 2011. Visited April 21, 2015.

[6] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.

[7] BBC. Google purges bad extensions from Chrome. http://www.bbc.com/news/technology-32206511, 2015. Visited April 21, 2015.

[8] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Security*. USENIX, 2012.

[9] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC*, 2009.

[10] Evernote. Evernote web clipper. https://chrome.google.com/webstore/detail/evernote-web-clipper/pioclpoplcdbaefihamjohnefbikjilc, 2015. Visited April 21, 2015.

[11] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David Wagner, et al. How to ask for permission. In *HotSec*. USENIX, 2012.

[12] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *WebApps'11*. USENIX, 2011.

[13] Google. Declare permissions. https://developer.chrome.com/extensions/declare_permissions, 2014. Visited April 21, 2015.

[14] Google. Chrome Web Store - Extensions. https://chrome.google.com/webstore/category/extensions?_sort=1, 2015. Visited April 21, 2015.

[15] Google. Google mail checker. https://chrome.google.com/webstore/detail/google-mail-checker/mihcahmgecmbnbcchbopgniflfhgnkff, 2015. Visited April 21, 2015.

[16] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Security and Privacy*. IEEE, 2011.

[17] Norm Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS OS Review*, 22(4):36–38, 1988.

[18] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Security*. USENIX, 2014.

[19] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[20] Petr Marchenko, Ulfar Erlingsson, and Brad Karp. Keeping sensitive data in browsers safe with ScriptPolice. Technical Report RN/13/02, UCL, January 2013.

[21] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Security*. USENIX, 2003.

[22] Alex Russell and Jungkee Song. Service workers. http://www.w3.org/TR/service-workers/, 2014. Visited April 21, 2015.

[23] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *IEEE*, 63(9), 1975.

[24] S. W. Smith. Humans in the loop: Human-computer interaction and security. *IEEE Security and Privacy*, 1(3), May 2003.

[25] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *OSDI*. USENIX, 2014.

[26] Kurt Thomas, Elie Bursztein, Chris Grierand Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Security and Privacy*. IEEE, 2015. To appear.

[27] Anne van Kesteren and Maciej Stachowiak. HTML design principles. `http://www.w3.org/TR/html-design-principles`, 2007. Visited April 21, 2015.

[28] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*. USENIX, 2006.

[29] Michal Zelwski. Browser security handbook, part 2. `http://code.google.com/p/browsersec/wiki/Part2`, 2009. Visited April 21, 2015.