Oh, the Places You'll Go! Finding Our Way Back from the Web Platform's Ill-conceived Jaunts

Artur Janc
Google, Inc
aaj@google.com

Mike West
Google, Inc
mkwst@google.com

***Abstract*—In its transition from the original concept of a mesh of hypertext documents [2] into the world's most successful application ecosystem, the open web platform [3] has steadily, iteratively, accumulated a large number of unsafe features and behaviors. These features lead to vulnerabilities in web applications, enable attacks on web users, and often add significant complexity to developers' mental models of the web and to user-agent implementations.**

In this paper, we start from a scattered list of concrete grievances about the web platform based on informal discussions among browser– and web security engineers. After reviewing the details of these issues, we work towards a model of the root causes of the problems, categorizing them based on the type of risk they introduce to the platform. We then identify possible solutions for each class of issues, dividing them by the most effective approach to address it.

In the end, we arrive at a general blueprint for backing out of these dead ends. We propose a three-pronged approach which includes changing web browser defaults, creating a slew of features for web authors to opt out of dangerous behaviors, and adding new security primitives. We then show how this approach can be practically applied to address each of the individual problems, providing a conceptual framework for solving unsafe legacy web platform behaviors.

1. Introduction

OH! THE PLACES YOU'LL GO!

You'll be seeing great sights!
You'll join the high fliers
who soar to high heights.

You won't lag behind, because you'll have the speed.
You'll pass the whole gang and you'll soon take the lead.
Wherever you fly, you'll be best of the best.
Wherever you go, you will top all the rest.

Except when you don't.
Because, sometimes, you won't. [1]

The web is the world's most successful application ecosystem, universally admired by users and authors. It is also woefully inadequate from a security standpoint: after decades of growth and accumulating features in a seemingly haphazard manner, the platform finds itself at risk from legacy behaviors that threaten to undermine its basic security and privacy guarantees. The threats are exacerbated by the web's openness and composability – resulting in an outsized attack surface – and the high amount and sensitivity of data which users have entrusted to web applications.

In our attempt to find a solution to this problem, we start by enumerating a number of specific long-standing sharp edges of the web platform which contribute to a variety of security issues affecting web applications and users. After listing the dangerous behaviors in Section 2 we group these seemingly unrelated problems into three categories based on the type of threat caused by each dangerous behavior in Section 3.

Crucially, this categorization assists us in evaluating potential solutions to the problems, which we discuss in Section 3.2. Specifically, we analyze *where* a change could be made to address a given problem pattern, identifying three major locations for possible fixes: *disabling the behavior by default* in web browsers, *providing an opt-out toggle* for web applications, and *creating new security primitives* to offer more principled solutions.

The remainder of this paper expands upon this idea to build a simple framework that can help the web security community understand how to best remove each source of insecurity from the platform. In Section 4 we review past web security improvements and see where they fit within our model. Finally, we propose a concrete plan for using modern mechanisms including the Reporting API [51] and Document Policy [49] to present a blueprint for a collaborative process for browser vendors and application authors to address current and future web security problems.

2. A Laundry List of Web Security Problems

It is the authors' strong belief that starting with a neat model which systematizes and offers possible solutions to long-standing problems is a surefire way to alienate the reader, have them immediately suspect the end result, and begin harboring negative feelings towards the authors. This is a particular risk when attempting to make sense of a domain as complex as the web, which – almost as a rule – defies elegant solutions and simple models.

Let's try to avoid this pitfall by first embarking on a survey of specific web behaviors which, with the benefit of hindsight, can be considered mis-features or have otherwise been shown to be the cause of security headaches for authors and users. Since agreeing on the problems is easier than agreeing on the solutions, we hope to find

common ground with the reader by commiserating about a number of problematic APIs that we expect the reader, like us, has had the misfortune of dealing with.

This list in this section is certainly not objective, nor is it exhaustive. However, we expect that the reader, being well acquainted with the web platform's quirks, will intuitively empathize with these problems and hope these issues are a sufficient starting point for the discussion further in this paper. The high-level grouping in this section is provided for clarity only; we explain and expand this categorization in Section 3.

2.1. Vulnerability-prone APIs

2.1.1. `document.domain`. `document.domain` [4] allows documents to relax the same-origin policy [5] and enables direct DOM access between documents hosted on different subdomains within the same site.

Setting `document.domain` to the parent domain in any document exposes all data in a given origin to attackers with scripting access to a same-site origin, removing a fundamental web security boundary [6].

2.1.2. Maintaining state over HTTP. Documents delivered over an unencrypted channel (HTTP [9]) enjoy access to most web APIs which allow persistence: cookies, and all variants of local storage (e.g. `IndexedDB` or `localStorage`).

This allows developers to create full-featured applications with user sessions, which fail to enforce basic data confidentiality and integrity guarantees.

2.1.3. Cookie semantics. As the canonical web mechanism to persist state, cookies are surprisingly misaligned with other fundamental web security mechanisms: their scoping defies the same-origin policy in important ways [10].

Importantly, cookies are generally unsafe by default: they are available to HTTP origins even when set over HTTPS, they are available to client-side scripts, and they are attached by default even on cross-site requests [11].

2.1.4. Mixed content. Browsers have historically not enforced that secure contexts (HTTPS) load all of their subresources securely. This allows sensitive applications to mistakenly load resources which can be observed and modified by network-level attackers, leading to loss of confidentiality and integrity for users [17].

2.1.5. `javascript:` URIs. When a document initiates a navigation (e.g. by loading a frame or pop-up, or changing its own location), if the target of the navigation happens to be a `javascript:` URL [18], this will result in script execution in the context of the navigating document.

This is a common cause of cross-site scripting vulnerabilities in applications which accept user-controlled links without validating their protocols, and in cases where untrusted input is interpolated by the application inside a `javascript:` URL [19] (the authors dare the reader to identify the correct escaping required for this context).

2.2. Behaviors Enabling Attacks on Websites

2.2.1. MIME type sniffing. MIME type sniffing [7] (also known as *content sniffing*) is a compatibility feature which causes the browser to ignore the `Content-Type` value provided by the server in order to process a resource whose advertised MIME type would make it fail to load or display.

This is a particular concern when the browser can be forced to treat a server response with the attacker's data as HTML or JavaScript, often as a result of allowing users to upload files [8], leading to cross-site scripting vulnerabilities.

2.2.2. Plugins; `<embed>` and `<object>` elements. For unfortunate historical reasons, plugins [12] enjoyed a decade as the *de facto* cross-browser standard to provide "rich content" such as games, multimedia, and more powerful runtimes with capabilities not provided by web APIs.

This brought along a variety of security problems, including memory corruption bugs in plugin implementations [13], a proliferation of custom security principals incompatible with the same-origin policy [14], and extant strangeness in how browsers handle resource loads via the `<object>` element.

2.2.3. DOM clobbering. As a convenient shortcut for accessing HTML elements in client-side scripts, browsers create references to such DOM elements based on their `id` and `name` attributes [16] (for example `<a id=foo>` will create a reference to the element in the global `window.foo` variable).

This allows attacks which control a limited part of the DOM of the application (for example, markup processed by an HTML sanitizer, or injections mitigated by the Content Security Policy) to affect the behavior of client-side scripts, enabling a number of attacks [15].

2.2.4. Credentials in URL's `userinfo`. Browsers allow URLs to include credentials and supply these credentials to servers (e.g. via HTTP 'Basic' Authentication [20] or when connecting to non-web protocols such as FTP).

This enables social engineering attacks based on displaying attacker-controlled messages in a trusted browser UI, and gives attackers opportunities to create state in the context of another origin.

2.2.5. Site-based security boundaries. Most security rules are enforced by the browser at the level of individual origins [5]; however, there are important counterexamples where APIs are more broadly scoped and information may leak between different origins within the site boundary.

Examples include domain-scoped cookies, and – driven primarily by performance considerations – HTTP cache partitioning [21] and process isolation rules [22].

2.2.6. `window.frames` and frame tree access. While the browser prevents access to most DOM properties of cross-origin windows, an important exception are the `window.length` and `window.frames` properties.

These properties reveal the number of iframes embedded in any cross-origin document, and allow the attacker

to interact with individual iframes (for example, send them messages via `postMessage`). This enables cross-origin information leaks [23].

2.2.7. `window.history`. The `window.history` API [24] reveals the number of navigations that have previously been performed in an attacker-controlled window, and allows navigation to any document in the window's history. This enables cross-origin information leaks [23].

2.2.8. Unconstrained `<base>` URIs. The `<base>` element [26] allows applications to set a URL prefix against which all subresource loads are resolved. This allows limited HTML injections (for example, those mitigated by Content Security Policy, or in cases where the attacker controls a small number of characters) to modify the locations of resources loaded from relative paths.

This allows exploiting a subset of injection bugs [25].

2.2.9. Liberal parsing of malformed markup. In true spirit of Postel's law [27], browsers' HTML parsers are liberal when processing markup, even when it's malformed. This allows limited HTML injections to alter the meaning of subsequent markup, possibly allowing exfiltration of the contents of the page.

In the specific example of *dangling markup* attacks [28], an attacker-controlled element (such as a `<textarea>` or an `<img>` tag with an unterminated src attribute) can swallow the remaining parts of the document and include them in a request to an attacker-controlled destination.

2.2.10. Unlimited recursive `@import` in CSS. The `@import` rule [35] in CSS allows the loading of arbitrary styles, and can be used recursively to include subsequent stylesheets.

This allows a single stylesheet injection to reliably exfiltrate data from the DOM by using advanced CSS features [36].

2.2.11. Scrolling to URL fragments. When a document is navigated to a URL containing a *location fragment*, the browser will automatically scroll to an element whose `id` matches the fragment value [37]. For example `site.example/#foo` will scroll to an element with `id=foo` if it exists.

Because the consequences of a scroll can be inferred cross-origin (for example by using `IntersectionObserver` in any iframes loaded by the document, or by inspecting the state of the HTTP cache), this reveals whether an element with a given `id` is present on the page [38]. This enables cross-origin information leaks.

2.2.12. `load/error` events on external resources. Cross-origin requests by default attach cookies and the browser provides APIs that reveal both whether the request succeeded and the accurate time when the response was received.

This allows cross-origin timing attacks and reveals whether the user has access to an arbitrary cross-origin resource, allowing deanonimization attacks and information leaks [39].

2.3. Behaviors Enabling Attacks on Users

2.3.1. Styling of visited links in CSS. The CSS `:visited` pseudoclass allows any document to style a link present in the user's browsing history differently from other links, creating a distinction which can reveal the list of pages visited by the user [29].

Despite mitigations that prevent the page from directly reading the styles of links, multiple side-channels have been demonstrated to allow attackers to infer the contents of users' browsing histories [30] [31].

2.3.2. Non-partitioned HTTP cache. The fact that the HTTP cache has historically constituted global state, shared between cross-origin applications, creates opportunities for any origin to determine the state of other applications based on their cached resources [32].

The HTTP cache also reveals information about the sites previously visited by the user and may allow deanonymization attacks [33].

2.3.3. Requests to RFC1918 addresses. Because browsers lack a distinction between public and private destinations, any website visited by a user can issue HTTP requests to any destination, even if the target address is local to the user and unreachable externally [34].

Attackers can task the user's browser with making requests to the loopback address or the user's local network, using the browser as a proxy. This enables fingerprinting the user's network environment and exploiting vulnerabilities in services not reachable directly by the attacker.

2.3.4. Nested contexts control top-level browser UI. Subresources, such as iframes, can initiate actions which affect browser UI visible at the top level, which can be easily misattributed by the user as being generated by the embedding site.

This allows crafting convincing UI spoofing attacks by showing prompts (`alert()` or `prompt()`), initiating downloads or requesting permissions in windows where an attacker controls an iframe [40].

The reader has likely noticed that even this incomplete enumeration of sharp edges in the web platform contains problems that are varied in kind, scope and impact. Is there a way to think about them that, even if not demonstrably *correct*, is at least *useful*?

We attempt to answer this question in the next section.

3. Categorizing the Web's Faults

You'll look up and down streets. Look 'em over with care.
About some you will say, "I don't choose to go there.";
With your head full of brains and your shoes full of feet,
you're too smart to go down any not-so-good street.

And you may not find any
you'll want to go down.
In that case, of course,
you'll head straight out of town. [1]

In our attempt to create a useful model of the security shortcomings of the web, we will depart from general

approaches traditionally used for understanding software anti-patterns [41], vulnerability classification [42] and fault analysis [43]. Readers looking for references to the Bell-LaPadula model [44] throughout this paper will find themselves sorely disappointed.

Instead, true to the principles of the web, and eschewing rigor and theoretical purity, we will base our categorization on the simplest approach that has a chance of working; like the problems, our categorizations in this section are inherently web-specific.

3.1. A Summary of Problems

We posit that it is an interesting thought experiment to understand the fundamental nature of each of the problems listed above. In the previous section we outlined *what* patterns reduce security on the web; here we will attempt to understand *why* they do so.

3.1.1. Vulnerability-prone API. A fairly evident subset of the problems arises because the use of certain web APIs in an application is likely to introduce vulnerabilities. For example, setting `document.domain` is by itself very frequently a security bug; using cookies in their default configuration, or composing `javascript:` URIs with user input similarly leads to security bugs.

The existence of such APIs isn't *inherently* a security problem. However, in practice, application authors who use such APIs will inevitably do so unsafely and make their code vulnerable. We posit that this is *de facto* undesirable for the ecosystem.

3.1.2. Enabling attacks on websites. Another class of problems stems from the fact that some web behaviors provide attackers with capabilities to attack unrelated applications to which the user is logged in. The existence of MIME sniffing, DOM clobbering, cross-origin access to `window.frames` or the ability to scroll other documents to a URL fragment, all create opportunities to conduct injection attacks or leak cross-origin information.

This category of issues can manifest itself in the presence of a seemingly benign pattern in the vulnerable application; for example, an application that correctly sanitizes user-provided HTML but doesn't account for DOM clobbering may be vulnerable to XSS as a result. Other behaviors, such as the ability to traverse a cross-origin document's frames, affect any web application, even in the absence of any programming mistakes.

3.1.3. Enabling attacks on the user. Some web security problems are based on an untrusted website attacking the user directly, even if the user doesn't have any authenticated web sessions. Revealing web browsing history with the `:visited` selector or by querying the HTTP cache, or the ability to craft requests to the user's local network, all subvert users' security and privacy expectations.

As a result, even a user who never logs into websites and doesn't store any of their data on the web can be affected by these behaviors.

3.1.4. Bonus criterion: Complexity. In some cases, a web feature doesn't *directly* cause security problems, but its presence introduces complexity in user agents and adds cognitive load for applications and library authors, increasing the risk of implementation bugs.

For example, the ability to carry credentials in URLs increases the difficulty of parsing links, and the presence of `document.domain` significantly complicates browsers' same-origin logic.

TABLE 1. A PARTIAL LIST OF UNSAFE WEB FEATURES

Feature	Problem	Solution
`document.domain`*	Vulnerability-prone API	Site opt-out
State over HTTP	Vulnerability-prone API	New primitive
Cookie semantics*	Vulnerability-prone API	New primitive
Mixed content	Vulnerability-prone API	Default disable
`javascript:` URIs*	Vulnerability-prone API	Site opt-out
MIME sniffing*	Enables attacks on websites	Site opt-out
Plugins*	Enables attacks on websites	Default disable
DOM clobbering	Enables attacks on websites	Site opt-out
`userinfo` in URLs*	Enables attacks on websites	Default disable
Site-based boundaries	Enables attacks on websites	Default disable
`window.frames`	Enables attacks on websites	Site opt-out
`window.history`	Enables attacks on websites	Site opt-out
Unrestricted `<base>`*	Enables attacks on websites	Site opt-out
Dangling markup	Enables attacks on websites	Site opt-out
Liberal HTML parsing	Enables attacks on websites	Site opt-out
`@import` in CSS	Enables attacks on websites	Default disable
Fragment scrolling	Enables attacks on websites	Site opt-out
`onload`/`onerror`	Enables attacks on websites	Default disable
`:visited` in CSS	Enables attacks on users	Default disable
Global HTTP cache	Enables attacks on users	Default disable
RFC1918 requests	Enables attacks on users	Default disable
UI for nested contexts	Enables attacks on users	Default disable

*Complex features (as defined in Section 3.1.4) are marked with *.*

3.2. A Summary of Solutions

Having proposed a summary of root causes for the problems, let's now categorize them based on a different criterion: the nature of a potential solution. Specifically, let's consider *where* an ideal solution to the unsafety could lie. We outline the categories below and come back to a more detailed discussion in Section 4.

3.2.1. Disable by default in web browsers. In an ideal case, a given source of unsafety could be completely removed by web browsers. Addressing problems in the browser improves the security posture of the platform, protecting users without requiring action on part of web developers.

In practice, such security improvements must generally weigh the security benefit against other criteria, e.g. compatibility or performance concerns.

3.2.2. Provide an opt-out toggle for applications. When a certain feature cannot be removed from the platform, but is not necessary for most applications, browsers can provide developers to ability to opt their sites out of a given unsafe behavior. This allows sensitive applications to reduce the risk for their users, while allowing the behavior for sites with less strict security requirements.

3.2.3. Create new primitives. In practice, some unsafe behaviors cannot be removed either by default or on an opt-in basis: certain APIs are simply too ubiquitous and lack a suitable replacement.

In this case, the web platform can provide new security mechanisms that preserve the use of the feature while

mitigating its risk; we provide several examples of security features which fill this niche in Section 4.3. Alternatively, the platform can create APIs that serve the same use cases as the problematic feature in a safer way, and provide a well-lit path to help developers migrate their existing code.

3.2.4. Solutions as a spectrum. An astute reader will notice that there is nothing inherent to each problem that would naturally match it with the most appropriate location of the solution. After all, browser vendors could decide to remove all sources of unsafety by default (for example: disable cookies and disallow HTTP traffic) or provide opt-in features for developers to disable them for their origins.

We propose that an important distinction is the deployment and criticality of a given source of unsafety. On one end of the spectrum lie issues which can be addressed by default by the browser; the opposite of the spectrum contains mechanisms which are ubiquitous and have no suitable replacement, and require browsers to provide security primitives to reduce their risk. In between the two extremes lies the "opt-out territory" where features can be disabled by individual applications, but cannot be fully removed from the platform.

We also propose that, in time, features can shift along this spectrum from being critical for most sites, through being discouraged and allowing developers to disable them on their sites, to being removed from the platform by browser vendors. The work on HTTP and plugins is illustrative of this journey.

3.2.5. Other classifications. In addition to understanding the essence of the problem and the most effective location for the remedy, the patterns in Section 2 can be categorized along a number of other useful axes. For example, it could be instructive to assess each pattern in terms of its severity, the type of attack it allows (some vendors' threat models may prioritize protecting users against injections, others may prefer mitigating information leaks and attacks on user privacy), the web compatibility impact of removing the behavior, or the amount of community consensus for addressing the given risk.

This may be particularly useful as web browser vendors' threat models diverge, or as input towards the prioritization of web platform security work by each vendor.

4. Solving Web Problems: A Discussion

And when you're in a Slump,
you're not in for much fun.
Un-slumping yourself
is not easily done.

You will come to a place where the streets are not marked.
Some windows are lighted. But mostly they're darked.
A place you could sprain both your elbow and chin!
Do you dare to stay out? Do you dare to go in? [1]

We find that the location of the most effective solution outlined in the previous section is a particularly compelling axis alongside which we can evaluate potential approaches for a given problem. Its primary benefit is that it gives us a blueprint for structuring the web platform changes to address the problem; specifically:

- If the problem can be addressed directly by the browser, vendors should aim to solve it by default. This is discussed in more detail in Section 4.1.
- Issues solvable with application opt-outs depend on browsers to implement such opt-outs. They would benefit from common deployment and reporting infrastructure, described in Section 4.2.
- Problematic web features which are in widespread use and can neither be fixed by default, nor can they be realistically abandoned by web developers require new APIs to let developers achieve the same goals more safely. Section 4.3 goes into more detail about this approach.

4.1. Fixes in Web Browsers

Several elements of the web platform have an infamous track record of creating security headaches for implementers, web authors and users. Plugins and the global HTTP cache are two examples: the former has introduced an inordinate number of implementation vulnerabilities leading to remote code execution issues in browsers [13], and the latter has been a canonical information disclosure vector [32], allowing leaks of users' browsing habits and enabling the exploitation of a number of XS-leaks [23].

In such cases, we argue that the correct approach is to remove the feature from the platform. The two major criteria to evaluate are the risk posed by the feature, and the web compatibility impact of removing it (or enabling its safe subset where possible). Features whose complete removal would incur a substantial compatibility cost could be selectively re-enabled, either per-user (for example, by administrators in control of the browser's enterprise policy [45]) or per-site as a Reverse Origin Trial [46].

An informed reader may notice that such browser-side work has been in progress; for example, the work to remove the reliance on plugins [47] and to prevent loading mixed content [48] has advanced to the point that support for these (mis-)features may soon be removed in major browsers.

We express hope that the global HTTP cache and visited link styling can be similarly addressed in the near future.

4.2. Opt-outs in Web Applications

When a feature poses a danger to the security of the web platform, but cannot be completely removed for compatibility reasons, we propose that the platform should allow web authors to disable it in their applications.

This applies particularly to problems in the "vulnerability-prone API" and "allows attacks on websites" categories – an application which exempts itself from a given dangerous behavior will be protected from mistakenly using error-prone APIs and from the attackers abusing unsafe web defaults to launch attacks on the application.

This is not a novel idea: features to opt out of dangerous defaults have long been present in the platform; notable examples include cookie attributes (`Secure`,

`HttpOnly`, `SameSite`) and headers to prevent content sniffing (`X-Content-Type-Options`) and embedding (`X-Frame-Options`).

Because a lot of the problems listed in Table 1 are amenable to this approach, we review this idea in more detail below.

4.2.1. Opting Applications out of Unsafety: A Plan.

We suggest that addressing dangerous web behaviors on a per-application basis requires, for each feature:

- A web-facing toggle to disable the behavior (e.g. an HTTP response header)
- A reporting mode that will notify authors about incompatibilities in their applications.

While the presence of a toggle is a *sine qua non* to this approach, the more interesting element is the reporting capability, which is in practice the true enabler of application adoption. Building on the successful mode introduced by Content Security Policy, we propose for each toggle to be accompanied by a *report-only* and *report-while-enforcing* modes. This allows developers to both understand if their application can safely disable the feature and be alerted about potential regressions in the figure..

Two upcoming web platform features provide the infrastructure that browser vendors can build on to enable this approach. Document Policy [49], an offshoot of Permissions Policy [50] scoped to same-origin boundaries, is a general mechanism that controls the use of web APIs; we posit that adding features to disable unsafe behaviors would be a natural extension to this proposal. The Reporting API [51] is a mechanism which provides an abstraction for other browser features that allows them to notify the server of potential problems or feature-dependent error conditions. We propose that combining these features would be a promising approach for a variety of voluntary security restrictions.

An interesting aspect of this approach is that it offers immediate benefits for a subset of behaviors. As soon as one browser implements reporting for unsafe APIs, developers will be notified of any incompatibilities. For the *vulnerability-prone APIs* class of problems, this allows authors to identify and remove any uses of the offending pattern, improving security for users of all browsers, without requiring the use of polyfills.

In the long-term, we envision that browsers could simplify the disabling of unsafe features by bundling them together in simpler *security best practices* modes (for example, origin-wide via the Origin Policy [52]). We anticipate that browsers can foster the adoption of such modes by upgrading the UI of compatible sites, or enforcing the restrictions for privileged sites (e.g. those to which the user logs in, or which request permissions).

One more approach to help the ecosystem move forward and opt out of unsafe features could be similar to what was already applied for HTTP. Gating some powerful APIs like media access behind a *Secure Context* [53] helped speed up the adoption of HTTPS, so a similar concept of an even *Securer Context* could be introduced to leave some old and unsafe knobs behind.

4.3. New Security Primitives

Realistically, some elements of the web platform cannot be removed either by default or even on an opt-in basis: many authors cannot build their applications without common features such as cookies.

In such situations, we recommend an approach of building replacement APIs which will address authors' use cases in a safer way, or providing new security features which will protect applications from the sharp edges of these mechanisms.

Interestingly, many modern web security features have been implicitly following this approach:

- **Content Security Policy** [54] doesn't address the root causes of markup injections (mixing code and data), but mitigates their impact by introducing fine-grained controls over script execution.
- **Trusted Types** [55] allow developers to keep using potentially unsafe DOM APIs, but prevent injections by requiring their arguments to be created safely by central policy code.
- **SameSite cookies** and **Fetch Metadata Request Headers** don't change the web platform default of attaching cookies as ambient authority to cross-origin requests, but allow developers to change this behavior for individual cookies, or to make fine-grained per-endpoint decisions based on additional security information included in HTTP headers.
- **Cross-Origin Opener Policy** lets developers disable cross-origin interactions with their top-level windows.

We anticipate that new opt-in security features can help address several problems outlined in Section 2 as well as other emerging web platform risks. However, the authors want to caution that there are diminishing returns to compensating for legacy insecurity by engineering new features. Solving long-standing web problems will likely require an increase emphasis on fixing the root causes of insecurity, as outlined in the previous sections.

5. Future Work

We believe this paper to be only an initial exploration of an area that's arguably more practical than academic: the idea of initiating a concerted effort to remove patterns that have proven to undermine the security of the web ecosystem. As such, there are several natural extensions to the concepts discussed here.

First, the individual problems in Section 2 would benefit from examination and expansion by the security community, in order to more completely capture the set of known unsafe behaviors of the web platform.

Second, considering additional criteria for categorizing web security problems, as outlined in Section 3.2.5, could identify promising approaches for prioritizing specific improvements.

Third, the web is sorely lacking in meaningful measurement studies which would shed light on the real web compatibility impact of removing existing features. We strongly encourage the academic community to develop better models and infrastructure in this space.

Finally, we believe that the most meaningful test of the ideas presented in this paper is whether a discussion among browser vendors and the security community can lead to concrete actions resulting in addressing the issues discussed in Section 2. We plan to initiate discussion within the W3C Web Application Security working group to evaluate and improve upon our approach.

6. Conclusion

You'll get mixed up, of course,
as you already know.
You'll get mixed up
with many strange birds as you go.
So be sure when you step.
Step with care and great tact
and remember that Life's
a Great Balancing Act.
Just never foget to be dexterous and deft.
And never mix up your right foot with your left.

And will you succeed?
Yes! You will, indeed!
(98 and 3/4 percent guaranteed.) [1]

In this paper we started by curmudgeonly complaining about long-standing unsafe patterns in the web platform, and explaining the problems they lead to. This provided us with an opportunity to review the root causes of the problems and divide them into three different groups based on the most promising solution: issues that can be addressed by default by web browsers, problems that can be solved at the application level by opting out of unsafe platform behavior, and the middle ground of issues that require replacing unsafe APIs with new ones, or providing additional security mechanisms to reduce their risk.

We then proposed a set of practical approaches for each class of issues, which can serve as a starting point for active work to address them. Importantly, we outlined an approach for disabling legacy unsafe behaviors and APIs based on Document Policy and the Reporting API, and explained how browsers can help developers deploy these security improvements in their applications.

We believe that approach lays the groundwork for meaningful improvements that will allow the web platform out of the legacy dark corners it explored over the past three decades.

References

[1] Dr. Seuss, "Oh, the places you'll go!", 1990, New York, NY: Random House.

[2] T. Berners-Lee, "Information Management: A Proposal", 1989-1990. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/History/1989/proposal.html

[3] W3C, "Open Web Platform", 1989-1990. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/wiki/Open_Web_Platform

[4] WHATWG, "Relaxing the same-origin restriction", HTML Standard, June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://html.spec.whatwg.org/multipage/origin.html?relaxing-the-same-origin-restriction

[5] WHATWG, "Origin", HTML Standard, June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://html.spec.whatwg.org/multipage/origin.htmlorigin

[6] Web Security Academy, "DOM-based document-domain manipulation", June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://portswigger.net/web-security/dom-based/document-domain-manipulation

[7] WHATWG, "MIME Sniffing", April 23 2020. Accessed on: June 7, 2020. [Online]. Available: https://mimesniff.spec.whatwg.org/

[8] A. Barth, J. Caballero and D. Song, "Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves," 2009 30th IEEE Symposium on Security and Privacy, Berkeley, CA, 2009, pp. 360-371, doi: 10.1109/SP.2009.3.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", RFC2616.

[10] A. Barth, "HTTP State Management Mechanism", RFC6265.

[11] D. Johansson, "Cookie Security", OWASP London, November 30 2017.

[12] WHATWG, "Plugins", June 5 2020. Accessed on: June 7, 2002. [Online]. Available: https://html.spec.whatwg.org/multipage/system-state.htmlplugins-2

[13] Adobe, "Security updates for Adobe Flash Player", Accessed on: June 7, 2002. [Online]. Available: https://helpx.adobe.com/security/products/flash-player.html

[14] Adobe, "Cross-domain policy file specification ", Accessed on: June 7, 2020. [Online]. Available: https://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html

[15] Web Security Academy, "DOM clobbering", June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://portswigger.net/web-security/dom-based/dom-clobbering

[16] WHATWG, "Named access on the Window object", June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://html.spec.whatwg.org/named-access-on-the-window-object

[17] J. van Bergen, "What Is Mixed Content?", Web Fundamentals, Accessed on: June 7, 2020. [Online]. Available: https://developers.google.com/web/fundamentals/security/prevent-mixed-content/what-is-mixed-content

[18] B. Hoehrmann, "The 'javascript' resource identifier scheme", IETF Draft. Accessed on: June 7, 2002. [Online]. Available: https://tools.ietf.org/html/draft-hoehrmann-javascript-scheme-03

[19] Firing Range, "URL-based DOM XSS vulnerabilities". Accessed on: June 7, 2002. [Online]. Available: https://public-firing-range.appspot.com/urldom/index.html

[20] J. Reschke, "The 'Basic' HTTP Authentication Scheme", RFC7617.

[21] "Optionally partition cache to prevent using cache for tracking", WebKit Bugzilla, Accessed on: June 7, 2002. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=110269

[22] The Chromium Projects, "Site Isolation". Accessed on: June 7, 2002. [Online]. Available: https://www.chromium.org/Home/chromium-security/site-isolation

[23] XS-Leaks "Browser Side Channels". Accessed on: June 7, 2020. [Online]. Available: https://github.com/xsleaks/xsleaks/wiki/Browser-Side-Channels

[24] WHATWG, "The History Interface", HTML Standard, June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://html.spec.whatwg.org/multipage/history.htmlthe-history-interface

[25] M. West, "Nonce Retargeting", Content Security Policy Level 3, October 15 2018. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/TR/CSP3/security-nonce-retargeting

[26] WHATWG, "The 'base' element", HTML Standard, June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://html.spec.whatwg.org/the-base-element

[27] J. Postel, "Transmission Control Protocol", RFC761.

[28] Web Security Academy, "Dangling markup injection", June 5 2020. Accessed on: June 7, 2020. [Online]. Available: https://portswigger.net/web-security/cross-site-scripting/dangling-markup

[29] A. Janc, L.Olejnik, “Feasibility and real-world implications of web browser history detection”, Proceedings of W2SP 2010.

[30] Z. Weinberg, E. Y. Chen, P. R. Jayaraman and C. Jackson, “I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks,” 2011 IEEE Symposium on Security and Privacy, Berkeley, CA, 2011, pp. 147-161, doi: 10.1109/SP.2011.23.

[31] M. Zalewski, “Some harmless, old-fashioned fun with CSS,” May 4 2013. Accessed on: June 7, 2020. [Online]. Available: https://lcamtuf.blogspot.com/2013/05/some-harmless-old-fashioned-fun-with-css.html

[32] E. Felten, M. Schneider. “Timing attacks on Web privacy”. In Proceedings of the 7th ACM conference on Computer and Communications Security (CCS ’00). Association for Computing Machinery, New York, NY, USA, 25–32. DOI:https://doi.org/10.1145/352600.352606

[33] G. Wondracek, T. Holz, E. Kirda and C. Kruegel, ”A Practical Attack to De-anonymize Social Network Users,” 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, 2010, pp. 223-238, doi: 10.1109/SP.2010.21.

[34] M. West, “CORS and RFC1918”, August 8 2017. Accessed on: June 7, 2020. [Online]. Available: https://wicg.github.io/cors-rfc1918/goals

[35] CSSWG, “Importing Style Sheets: the @import rule”, June 28, 2019. Accessed on: June 7, 2020. [Online]. Available: https://drafts.csswg.org/css-cascade-3/at-ruledef-import

[36] x-c3ll, “CSS Injection Primitives”. Accessed on: June 7, 2020. [Online]. Available: https://x-c3ll.github.io/posts/CSS-Injection-Primitives/

[37] W3C, “Introduction to links and anchors”. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/TR/REC-html40/struct/links.htmlh-12.2.3

[38] D. Bokan, “Possible side-channel information leak using IntersectionObserver”. Accessed on: June 7, 2020. [Online]. Available: https://github.com/WICG/scroll-to-text-fragment/issues/79

[39] A. Sudhodanan, S. Khodayari, J. Caballero “Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks.”, 2019. arXiv:1908.02204v2.

[40] M. West “Play safely in sandboxed IFrames.”, 2013. Accessed on: June 7, 2020. [Online]. Available: https://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/

[41] W. Brown, R. Malveau, H. McCormick, and T. Mowbray. 1998. “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis (1st. ed.)”. John Wiley Sons, Inc., USA.

[42] MITRE “Common Weakness Enumeration”, Accessed on: June 7, 2020. [Online]. Available: https://cwe.mitre.org/

[43] P. Brooke, R. Paige “Fault trees for security system design and analysis.” Comput. Secur. 22, 3 (April, 2003), 256–264. DOI:https://doi.org/10.1016/S0167-4048(03)00313-4

[44] D.E. Bell, L.J. LaPadula “Secure computer systems: Mathematical foundations”, MITRE. Accessed on: June 7, 2020. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/770768.pdf

[45] Google “Understand Chrome policy management”, Accessed on: June 7, 2020. [Online]. Available: https://support.google.com/chrome/a/answer/9037717

[46] The Chromium Projects “Origin Trials”, Accessed on: June 7, 2020. [Online]. Available: https://github.com/GoogleChrome/OriginTrials

[47] The Chromium Projects “Flash Roadmap”, Accessed on: June 7, 2020. [Online]. Available: https://www.chromium.org/flash-roadmap

[48] E. Stark, M. West “Mixed Content Level 2”, February 10, 2020. Accessed on: June 7, 2020. [Online]. Available: https://w3c.github.io/webappsec-mixed-content/level2.html

[49] I. Clelland “Document Policy”, May 27, 2020. Accessed on: June 7, 2020. [Online]. Available: https://w3c.github.io/webappsec-feature-policy/document-policy.html

[50] I. Clelland “Permissions Policy”, May 27, 2020. Accessed on: June 7, 2020. [Online]. Available: https://w3c.github.io/webappsec-feature-policy/

[51] D. Creager, I. Grigorik, P. Meyer, M. West “Reporting API”, September 25, 2018. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/TR/reporting/

[52] D. Denicola “Origin Policy”, March 16, 2020. Accessed on: June 7, 2020. [Online]. Available: https://wicg.github.io/origin-policy/

[53] M. West “Secure Contexts”, September 15, 2016. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/TR/secure-contexts/

[54] M. West, “Content Security Policy Level 3”, October 15 2018. Accessed on: June 7, 2020. [Online]. Available: https://www.w3.org/TR/CSP3/

[55] K. Kotowicz, M. West, “Trusted Types,” June 3, 2020. Accessed on: June 7, 2020. [Online]. Available: https://w3c.github.io/webappsec-trusted-types/dist/spec/