Towards Improving the Deprecation Process of Web Features through Progressive Web Security

Tom Van Goethem
imec-DistriNet, KU Leuven
tom.vangoethem@kuleuven.be

Wouter Joosen
imec-DistriNet, KU Leuven
wouter.joosen@kuleuven.be

***Abstract*—To keep up with the continuous modernization of web applications and to facilitate their development, a large number of new features are introduced to the web platform every year. Although new web features typically undergo a security review, issues affecting the privacy and security of users could still surface at a later stage, requiring the deprecation and removal of affected APIs. Furthermore, as the web evolves, so do the expectations in terms of security and privacy, and legacy features might need to be replaced with improved alternatives. Currently, this process of deprecating and removing features is an ad-hoc effort that is largely uncoordinated between the different browser vendors. This causes a discrepancy in terms of compatibility and could eventually lead to the deterrence of the removal of an API, prolonging potential security threats. In this paper we propose a progressive security mechanism that aims to facilitate and standardize the deprecation and removal of features that pose a risk to users' security, and the introduction of features that aim to provide additional security guarantees.**

I. Introduction

Every year several hundreds of new features are added to the web platform, each of which go through a standardization track before an implementation lands in the browser. This ensures that when a developer designs a website using standardized features, it will be compatible in all browsers. When a feature has gone through the standardization track, which typically also includes a review of the security implications, an implementation in the various browsers typically follows. At this point, web developers can start using the new features, and will typically rely on a continued support of these features. However, in practice, it may happen that features need to be removed from the web platform. There are several reasons why this may occur. For instance, a certain feature may not achieve the adoption rate that was originally intended, making it infeasible for browser vendors to continue support and development of an underused API. Another, more severe scenario is that a feature has been found to either introduce security or privacy threats, or blocks the design of a more secure web. In this case the feature may pose a threat to web users and should thus be removed as fast as possible.

In contrast to the introduction of new features, the deprecation and removal of features in the current state of the web is done in an ad-hoc and mostly uncoordinated fashion. This provides web developers with an unclear overview of the deprecation status of features as this may differ depending on the browser. Furthermore, the incoherent timeline of feature removal may cause compatibility issues causing a website to function in one browser but not an other one that removed support for a feature with a nefarious security impact. A possible side-effect of this, is that this may trigger users to switch to a browser that retained support for a deprecated feature, despite the additional security risk this poses. This in turn could deter browser vendors to be the first to remove a feature, resulting in an extended timeframe during which a feature with nefarious security implication remains available in the browser.

In an effort to improve the process of deprecation and removal of features, in particular those that have an impact on the security or privacy of users or websites, we propose the progressive security mechanism. This mechanism aims to provide a streamlined and standardized approach to remove features from the web platform and provide a unified way for web developers to configure their applications. The design of the mechanism is based on incrementing versions, where each version gradually improves the security of the browser. By periodically increasing the minimally acceptable version, the provided security guarantees progressively advance. Our proposed mechanism is based on a qualitative analysis of past and ongoing deprecation and removal efforts, as well as the enabling of security mechanisms by default. We show that this mechanism provides a similar level of granularity for temporarily opting in to deprecated features, or opting out to new security mechanisms, compared to the current ad-hoc removal approaches.

Through this paper we intend to initiate a discussion on how the deprecation and removal process of features with an adverse security impact can be improved, and consider the proposed progressive security mechanism as an initial step in that direction.

In summary, we make the following contributions:

- We perform a quantitative analysis on the deprecation status of browser features, and find that approximately 10% of features are marked as deprecated, of which only a small fraction (4.3-16.8%) have been removed from the web platform.
- We analyze three ways in which features affect the improvement of security on the web: certain features may be used to launch cross-site attacks, improving the architecture of the browser might be impeded by requiring support of legacy features, and security mechanisms can

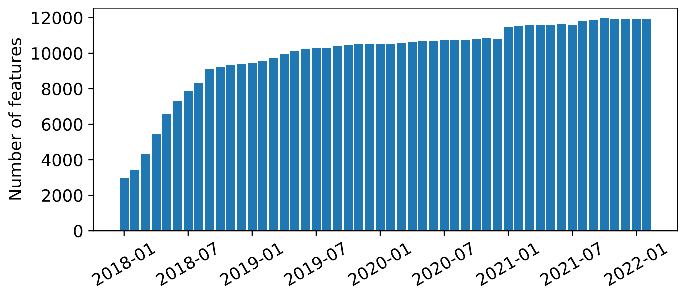


Fig. 1. Number of web platform features tracked by MDN over time.

be enabled by default.

- We propose a progressive security mechanism that aims to standardize the deprecation and removal of features with a negative security impact, and support the introduction of security mechanisms.

II. Web platform features

A. Overview

Ever since the inception of the first web browser, new features have continuously been added to the web platform. These new technologies allow developers to create performant and highly interactive web applications that are typically made up out of HTML, CSS and JavaScript. Some of the major evolutions in the browser include the interplay with peripherals, e.g. camera, microphone, accelerometer and gyroscope, access to a virtual or augmented reality through WebXR, native execution speed of binary code using Web Assembly, and background processes – referred to as service workers – that allow a website to be accessed even when the user is offline. At the time of writing, a total of 527 browser mechanisms are listed by the CanIUse project [1].

Each browser mechanism can have several features: for instance, service workers can be used to serve content from the cache but can intercept push messages from the server. This more detailed information is tracked by the Mozilla Developer Network (MDN), which captures detailed information about the different features, in the form of Web APIs, CSS features, HTTP headers, HTML element, JavaScript language features, and XML-based markup mechanisms, namely MathML and SVG. In Figure 1 we show the evolution of the number of features that are tracked by MDN over time, according to the publicly available data on the browser compatibility repository [2]. Note that this data is incomplete, and that efforts have been made since 2018 to capture information about all features. As of February 2022, a total of 11,912 features (excluding those of the Web Driver and Web Extensions components) are tracked by this dataset.

In Figure 2 we show the number of features that have been added or removed for each browser version, for the three main browser engines: Chromium (Chrome), Gecko (Firefox), and WebKit (Safari). Interestingly, with every browser release tens to hundreds of new features are introduced; for Firefox and Chrome a new version is released approximately every month,

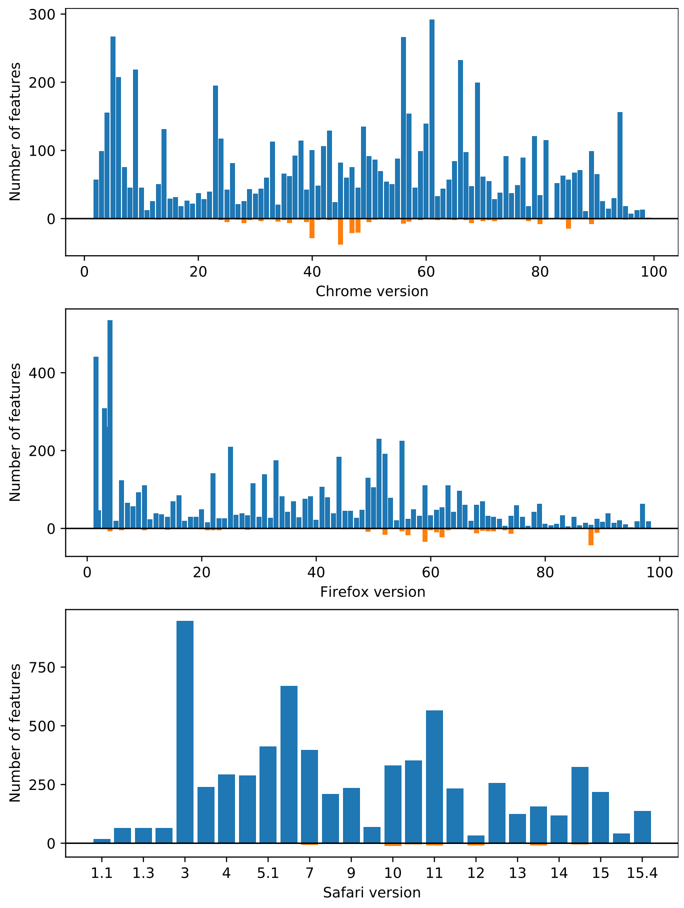


Fig. 2. Number of features added/removed per browser version.

for Safari this is between 3 and 6 months. Furthermore, a limited number of features are removed from the browser. This can be the result of various reasons, as we will discuss in the remainder of this section.

B. Lifetime of a browser feature

Before a new feature lands in the browser, it will typically have gone through a standardization track to ensure compatibility among different browsers. As such, most features share a common lifetime, which can be summarized as follows:

Initiation. The first step in the lifetime of a browser feature is its initiation. Typically, this is the result of a growing need of web developers to implement certain functionality that is either not possible or highly cumbersome with existing browser features. Alternatively, new features may be introduced to give web developers ways to protect against new attacks (e.g. using a Cross-Origin Opener Policy), or that improve existing features (e.g. the Service Worker API that overrides Application Cache). The new feature can then be proposed in a common forum such as the Web Incubator Community Group (WICG), in the form of a specification. As of February 2022, a total of 127 features have been drafted and are being discussed in the WICG.

Consensus. Once a draft specification has been submitted to WICG, it is publicly discussed among community members and browser vendors, possibly leading to changes to the proposed mechanism. The specification editors then aim to achieve consensus among the different stakeholders, after which the draft is finalized. At this point, the specification can be incorporated by the World Wide Web Consortium (W3C), or become part of the Web Hypertext Application Technology Working Group (WHATWG) living standard.

Implementation. Once a standardized specification is available, browsers can start implementing the feature. Initially, as the implementation is still gaining maturity, features are typically only available behind a configuration flag that can be enabled by the end-users. Alternatively, Chromium-based browsers have a concept of "Origin Trials", which allows developers to experiment with new web platform features [3]. Similarly, Firefox has a similar mechanism named "site permission add-ons", which allows individual sites to request a user's permission to a add-on-gated feature [4].

Adoption. When the implementation of a new browser feature has sufficiently matured, browsers will enabled it by default, and websites can start adopting it. If possible, a polyfill implementation that provides the same functionality albeit at a likely worse performance is provided to ensure backward compatibility for users with older browser versions.

Deprecation. As the web platform is constantly changing, it might be that after some time a feature is superseded by a newer mechanism, or that the initial expected support for the feature did not match the reality and it was never adopted. Additionally, in worse cases, it could be that the feature introduces new web vulnerabilities that were not considered when the feature was initially introduced. An example of such a case is the browser's reflected XSS detection mechanism, which could be controlled through the `X-XSS-Protection` header. When it was found that this feature would introduce various cross-site information leaks, and could be circumvented, it was deprecated and eventually removed from the web platform.

Removal. As features need to be maintained and extend the threat surface, deprecated features should be removed from the web platform. However, when a feature is already in use by a number of websites, which may rely on its presence, the removal of a feature may be arduous as it could cause existing websites to break in a particular browser. For some features, it may take several years before it can be completely removed from the web platform. This is especially worrisome for features that introduce vulnerabilities, or that inhibit the development of a more secure browser design. In this paper we propose a new mechanism to expedite the removal of features, or limit the nefarious side-effects of its presence.

C. *Deprecation of web features*

Next, we explore the deprecation of web platform features in more depth, and again leverage the browser compatibility dataset by MDN [2]. In Figure 3 we show the total number of deprecated features over time, along with the percentage they make up out of the total number of featured. On the bottom graph we show the number of features that became

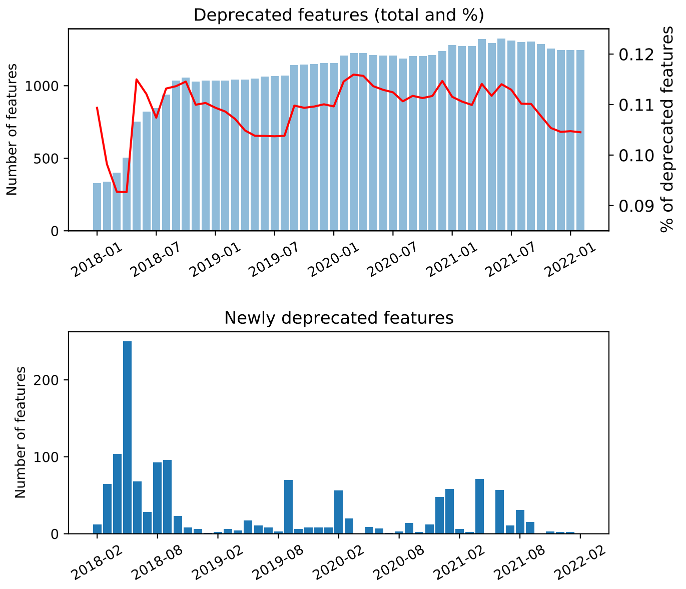


Fig. 3. Total number of deprecated features over time, as tracked by MDN. The top graph shows the total number of deprecated features over time (blue), and the percentage of features that have been marked as deprecated (red). The bottom graph shows the number of features that were marked as deprecated in a particular month.

TABLE I
NUMBER OF DEPRECATED FEATURES THAT HAVE BEEN ENABLED OR REMOVED FROM THE THREE MAIN BROWSER ENGINES.

Browser	Deprecated and enabled	Deprecated and removed	Total removed
Chrome	1,093	152	234
Firefox	1,036	209	306
Safari	1,191	54	68

deprecated in a granularity of one month. We note that the large increase in deprecated features in early-2018 is due to a large number of new entries being added to the dataset, and thus do not accurately reflect the deprecation rate. Overall, we find that more than 10% of web platform features are marked as deprecated, despite the large number of new features being introduced (see Figure 1). This percentage has remained fairly stable over time.

As of February 2022, we find that a total of 1,245 (10.45%) features have been marked as deprecated. In Table I we show the number of these deprecated features that are still supported by the three main browser engines. This data clearly shows that the vast majority of deprecated features still remain supported by most browsers; e.g. 1,093 deprecated features are still supported by the Chrome browser. In fact, we find that features may remain in the deprecated status for several years, clearly indicating that completely removing features from the web platform is highly challenging. Finally, some features were also removed from the browser that were previously not marked as deprecated. These include features that have been removed for a long time, and for which the deprecated status was thus not captured in the dataset, and features that were

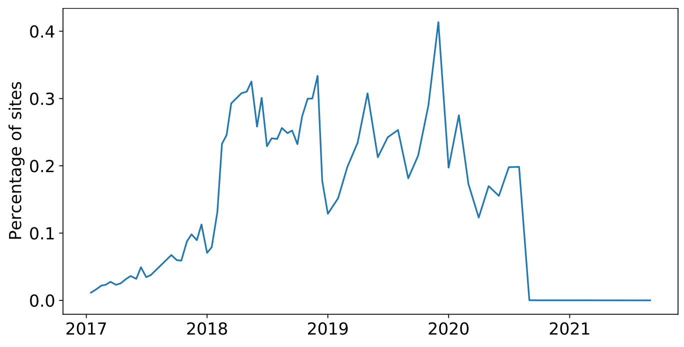


Fig. 4. Percentage of page loads using AppCache in a secure context, based on data from Chrome Platform Status metrics.

experimental on which web developers should not rely, such as the `X-XSS-Protection` header. However, even considering all the features that have been removed from the web platform, there are still 3-4 times as many deprecated features that are still enabled, extending the threat surface of the browser.

III. Improving security of the web platform

As the web platform is evolving, also its security needs to evolve: new attacks are discovered, or existing security issues become more prevalent due to the increased complexity of web applications. We can distinguish improvements to the security of the web platform in three different classes, which we will discuss in more detail throughout this section. A first class of security improvements is to minimize the attack surface of the browser and limit the features that an adversary could leverage as a gadget in their attacks. Second, the architecture of browsers might have to change over time in order to accommodate for new attacks that are discovered, such as Spectre. In order to support certain features, the improvements of the browser architecture cannot be made, and thus these features may impede the security of the browser. Finally, as there exist many legacy applications which may not be frequently updated, the security guarantees that the browser provides by default should be as strong as possible.

A. Thwarting attacker gadgets

Every browser feature that interacts with a cross-site resource, whether it is fetching it, parsing it or rendering it, has the potential of revealing information about this (potentially sensitive) response. As new features are proposed, these typically undergo a review to assess whether these introduce new ways to leak information. However, this assessment might not have been performed on features that were introduced at an earlier stage, and that are still enabled in browsers. Moreover, the security assessment might have missed certain attack vectors, or because of newly discovered vulnerability classes, the security impact of a feature may change over time. This is particularly undesirable for features that are deprecated and will be removed from the web platform: the longer that these features remain enabled, the longer these pose a threat.

An example of a deprecated feature that could be abused to launch attacks, is Application Cache (AppCache). Several years after the introduction of AppCache, at a time that transport security (SSL/TLS) had become more prevalent, it was apparent that the feature posed a significant threat when the page was loaded over an insecure context. More specifically, when the page was loaded over an unencrypted connection, a MitM attacker could install a malicious Application Cache. As a result, this would give the adversary permanent access to the website, even when the user is no longer connected to the network compromised by the adversary. As a reaction, browser vendors deprecated the use of AppCache in insecure contexts. A detailed timeline of the deprecation for the main browser engines can be found in Figure 5. Firefox was the first to deprecate AppCache (September 2015), which was followed by Chrome, where the use in insecure contexts was deprecated in February 2016. Safari deprecated the use of AppCache in January 2018. For Firefox and Chrome it was almost three years later that the use of AppCache was restricted to secure contexts (September and October 2018 respectively).

Later on, it became apparent that AppCache was the source of other security issues as well. For instance, it could be abused to leak the response status, as reported by Lee et al. in 2015 [5]. More recently, Herrera found that the pattern matching feature of AppCache could be abused to determine the URL of a redirect, which could be used to leak access tokens or determine the identity of the user [6]. Other issues that were reported to the browser vendors include a circumvention of Chrome's Cross-Origin Resource Blocking (CORB) mechanism [7], [8], a leak of information about the size of responses [9], or even a use-after-free vulnerability that resulted in a remote code execution [10]. These issues led to the general deprecation of AppCache: in Firefox it was already deprecated in September 2015; Chrome announced its intent to deprecate in August 2018, which was approved more than a year later in September 2019; Safari deprecated the feature in January 2018.

Similar to the process of disabling the use of AppCache in insecure contexts, the complete removal of the feature also took several years since its deprecation. In Safari, it is even still supported at the time of writing (February 2022). This means that the continued support for a single feature, which at its peak was only used by approximately 0.4% of sites according to the HTTP Archive dataset, expanded the browser's threat surface and allowed adversaries to launch attacks that were otherwise not possible, for several years. Interestingly, when looking at the prevalence of AppCache used in a secure context, an increase can be identified starting from approximately February 2018, several years after the deprecation of the feature in Firefox and just after its deprecation in Safari. It was only when the feature was removed in the Chrome browser by default that the number of sites using it dropped significantly, as can be seen in Figure 4, which shows the usage of AppCache in a secure context over time based

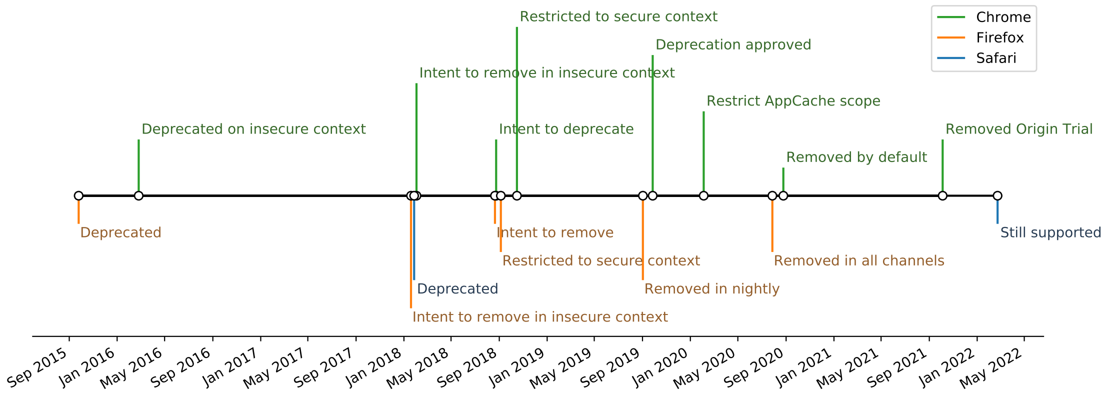


Fig. 5. Timeline of the deprecation and removal of Application Cache in Chrome, Firefox and Safari.

on data from Chrome Platform Status metrics[1].

Although the feature could still be used maliciously when it was only available through an Origin Trial, the browser vendor retained more control over which sites were using it, and the token could be revoked in case of abuse. However, this would most likely not prevent targeted attacks.

B. *Improving browser architecture*

A second class of implications that deprecated features may have, is that they may impede the implementation of mechanisms that improve the security of the browser and its general architecture. A concrete example of such a deprecated feature is the `document.domain` property, which can be used to relax the same-origin policy. More specifically, by setting this property, two pages that are cross-origin but same-site can access each other directly. For example, a page on `https://foo.example.com` can access `https://bar.example.com` when the `document.domain` property is set to `https://example.com` on both pages. Not only does this feature diminish the protections provided by the same-origin policy, it also prevents the adoption of origin-based process isolation [11]. As other, more secure mechanisms, such as postMessage and MessageChannel, can be used for cross-origin communication, the `document.domain` feature has been deprecated.

Although it has been generally known that it has unfavorable side-effects, it was only deprecated in the WHATWG standard in July 2020[2] and the official deprecation in Chrome is only planned for approximately September 2022[3]. Most likely, the reason for this is that the usage of the feature has been relatively high, with over 10% of page visits setting the `document.domain` property[4]. Nevertheless, only in a small fraction of the cases (0.3% to 0.5% of page visits) is the relaxation of the same-origin policy actually used[5].

Because the browser does not know in advance whether the `document.domain` property will be set, it cannot know whether the browser process of the web page can be isolated according to origin (host, scheme and port). Hence, browser processes can currently only be isolated at the granularity of a site, i.e. eTLD+1 and scheme [11]. As the deprecation and removal of the `document.domain` is expected to be a painstaking experience, a new mechanism named origin-keyed agent clusters was introduced [12] and added to the HTML standard. The feature, which is based on a response header (`Origin-Agent-Cluster`) can be used by web developers to indicate whether or not a relaxation of the same-origin policy through the `document.domain` setter should be disabled. When disabled, which browser vendors aim to make the default, the isolation of processes can be made at the level of origin, providing better protection against Spectre attacks [13].

C. *Security by default*

A third way in which features affect the improvement of the security of browsers, is by introducing new features, typically in the form of response headers, that can be used to control the potentially nefarious effects of legacy mechanisms. A concrete example of such a case is the embedding of cross-origin pages, which is known to be the source of a myriad of security issues [14], [15]. To give web developers the possibility to defend against these attacks, the `X-Frame-Options` header and `frame-ancestors` directive of CSP has been introduced. However, because in an earlier era of the web, framing content was a very prominent way of creating web applications, its overall usage is still pervasive, especially on legacy applications. Consequently, deprecating this feature would not be “web compatible”. Instead, a current proposition is to require an explicit opt-in for a web page to be embedded in a cross-origin context [16].

[1]https://chromestatus.com/metrics/feature/timeline/popularity/1248
[2]https://web.archive.org/web/20200715094921/https://html.spec.whatwg.org/multipage/origin.html#relaxing-the-same-origin-restriction
[3]https://chromestatus.com/feature/5428079583297536
[4]https://chromestatus.com/metrics/feature/timeline/popularity/2026
[5]https://chromestatus.com/metrics/feature/timeline/popularity/2544

Another response header that has been introduced to mitigate issues that are caused by a legacy feature, is the Cross-Origin Opener Policy (COOP). This mechanism can be used by web pages to instruct the browser that cross-origin pages cannot retain a reference to it, e.g. via the `window.opener` property or when the page was opened via `window.open()`. When the header is set, the page is protected against various XS-Leak attacks, such as counting the number of frames [17], and is the practicality of other attacks severely limited [18]. Similarly to embedding cross-origin pages, there is a small but significant portion of web applications that rely on using this mechanism, and therefor it is unlikely that it can be deprecated or removed as this would likely cause a backlash from web developers. Instead, it has been proposed that COOP should be enabled by default whilst allowing sites to explicitly opt out [19]. This would ensure that websites are secure by default, which is a significant improvement over the current state where the security of the vast majority of web applications is sacrificed to retain support of a legacy feature that is used by a limited number of websites.

IV. Progressively improving web security

In this section we propose a new mechanism that aims to facilitate a progressive improvement of the security of the web platform with regards to the deprecation and removal of features that inhibit this improvement. First, we determine the different requirements for such a mechanism. Based on these requirements and a qualitative analysis of the current state of deprecation and removal of web features and the techniques that are used to improve security on the web, we introduce a mechanism that meets these requirements.

A. Goals and requirements

Although there are several ways through which new features can be introduced to the platform, there is no synchronized way to remove them if they lack adoption or, even worse, introduce security issues. A first goal of our proposed mechanism is that the way that features are deprecated and eventually removed from the browser is **synchronized**. This has several advantages: when all browser vendors agree upon a fixed date when features are deprecated or removed, the behavior of those features will be the same in all browsers. Consequently, consequences of the removed support will be the same in all browsers, and, in case of backlash from web developers, browser vendors will not be singled out. This has been indicated as the cause of delaying the process of deprecation and feature removal [20].

A second goal is that the mechanism should aim to **gradually improve security**. Due to the dynamic aspects of how features are adopted and used, it is practically infeasible to instantly switch to a more secure version of the web. Instead, this change should be gradual, and be focused on initially reducing the adoption rate of a feature, e.g. through deprecation, whilst providing more secure alternative solutions. In a second phase, the support for security-sensitive features can be either removed, or required to explicitly be opt-in. This allows websites that rely on the feature to still use it with the additional effort of setting a response header, at the expense of their own security.

For the mechanism we introduce to progressively improving security, we aim to provide a **tailored approach** for handling features depending on their impact on security. For instance, the deprecation and removal of features that can be used by adversaries to launch attacks on websites should be treated differently than features whose usage could affect the security of a website. Although an opt-out in the latter case would only affect the security of a single site, if it would be possible to still use a feature that can be used to attack other sites, its remained support affects the security of all sites.

Furthermore, the aim of the progressive security mechanism should be to eventually guarantee **security by default**. Concretely, regardless of which web APIs are used, a web application should be protected against attacks. In the current state of the web, this is not the case. For instance, if a website does not explicitly prevent framing, it currently is susceptible to a series of attacks such as clickjacking, cross-origin leaks, or even certain cross-site scripting attacks [15].

Another requirement for the progressive security mechanism is that it should be **straightforward to implement** in order to facilitate deployment. However, it should also provide a sufficient level of **granularity** to allow web developers to customize their configuration to match the requirements of their application. We believe that a strategy that is used by current (ad-hoc) deprecation and removal efforts is most suitable: initially deprecating a feature to raise awareness that a certain feature is intended to be removed, followed by the by-default removal of the feature, possibly providing an explicit opt-in to indicate that the feature should still be enabled. Finally, the presence and settings of the progressive security mechanism should be **communicated as soon as possible** such that the browser can take appropriate actions before the web page is rendered.

B. Progressive security

Our proposed mechanism that aims to gradually improve the security of the web platform by facilitating the deprecation and removal of browser features, is based on a monotonic increasing version system. With each new version number, features can be moved to either the deprecated list, the unsupported list, or the list of (security) features that are enabled by default, where the composition of each of these lists are synchronized. That is, the web community agrees upon which features need to be deprecated, removed or enabled by default. As the main focus of this mechanism is to improve the security of the web platform, we believe that the scope of the features that should be considered are only those that have a security impact. In their study on feature deprecation in Chrome, Mirian et al. categorized 118 deprecated features, and found that 26 of those were due to security concerns [20].

With this version system, it is possible to achieve the same granularity of actions compared to most of the current individual deprecation and removal efforts. In the most straightforward scenario, a feature could be marked as *deprecated* in version n, and eventually *removed* in version $n + 1$ or $n + 2$. However, an alternative option is also feasible where the feature is first deprecated in version n, which may be followed by a security mechanism that is *enabled by default* and mitigates the issues caused by the deprecated feature. In this case, the majority of web sites will be protected whereas it is still possible for websites to opt out of the security mechanism, at their own risk. Ideally, the enabled-by-default state is only temporary, and the feature that causes a security impact is eventually removed from the web platform.

The last set of features that need to be defined by the web community is those that are *unsupported*, but that can be re-enabled by explicitly opting in to it. For example, once a feature such as Application Cache has been placed on the unsupported list, it will not be available by default. However, if a website explicitly declares that it wants to use it, it will be re-enabled depending on the current progressive security version of the browser. For example, if a feature was added to the unsupported list in version n, then it would only be possible to explicitly opt in to it as long as the progressive security version that is considered the default in the browser is less than or equal to $n + 2$. The mechanism of this explicit opt-in is similar to what web developers can achieve with the reverse origin trial (as was the case for Application Cache), In the case of AppCache, there were only a very limited number of sites that continued using the feature after it required an explicit opt-in. As such, we believe that this provides a good trade-off between retaining support for legacy applications and guaranteeing security for most web users.

By allowing an explicit opt-in for features that are considered unsupported, it is important to distinguish between features that pose a security threat to the site itself, or that can be used to attack other sites. While the former only affects the security of the single site, which knowingly agrees to this threat, in the latter case the feature could be used launch cross-site attacks on any other website. As such, we propose that in case a website wants to opt in to using a feature that can possibly harm other sites in attacks, this would require explicit permission from the user. This permission could be granted in a similar fashion as to how permission needs to be granted for access to the microphone or camera. Although users could be tricked in providing this permission, any possible attack leveraging the deprecated feature would still require a manual interaction from the targeted user, thereby hindering or impeding the adversary.

Finally, in order to gradually improve security, the web community will determine at what time, i.e. for which browser releases, the minimal progressive security version will become required. Ideally, it can be agreed upon by the web vendors that the new browser versions requiring a new progressive security version are in quick succession. This will ensure that changes occur globally at a predefined time, and thus features will stop working at the same time in all browsers. To further facilitate testing the impact of new progressive security features, the beta or nightly versions of the browser could require a higher minimal progressive security version.

C. Implementation

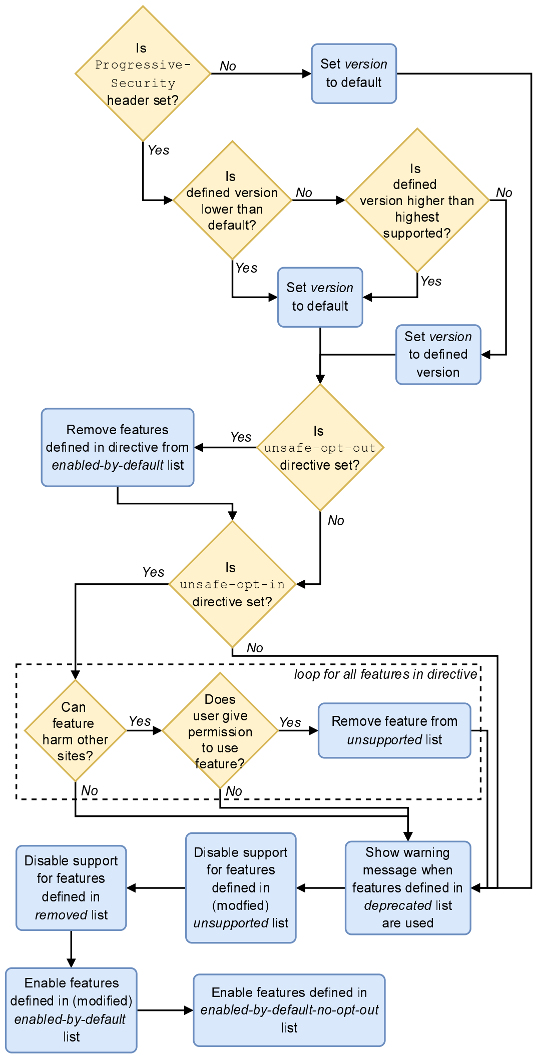
When the set of features for the different feature lists (deprecated, unsupported, removed, enabled-by-default, and enabled-by-default-no-opt-out) has been defined and agreed upon for the current progressive security versions, this still needs to be enforced by the browser. The default policy that will be enforced by the browser is based on the progressive security version that is considered the minimal required version. Web developers can opt to deviate from this behavior by specifying a `Progressive-Security` response header. This header is defined as follows:

```
Progressive-Security:
  version;
  [unsafe-opt-out=(feature_list);]
  [unsafe-opt-in=(feature_list)]
```

As such, the header has one required value, namely the progressive security version. The browser will ensure that this is capped to the available versions, i.e., the enforced version will be at least as high as the default browser policy and not higher than any of the currently supported versions. This means that with the version parameter, websites can only opt in to stronger security guarantees compared to the default. Furthermore, if a website owner sets the header with a certain version number and then forgets to update the value for an extended amount of time, the site will still have the same security level as if it would not set the header. By capping to the maximum supported version, websites could set the value to a very high integer and always be opted in to the maximum security levels that are available. This may however come at the cost of compatibility, as a new progressive security version could cause a browser feature on which the site relies to be removed. Instead, in the ideal use-case, web develops would first verify the compatibility of their website with a new progressive security version by using a beta or nightly build of the browser and then explicitly opt in to it.

The first optional directive is `unsafe-opt-out`, which can be used to opt out of security mechanisms that are enabled by default, i.e. those that are on the *enabled-by-default* list. As this could negatively affect the security of the website, it is prefixed with the `unsafe` keyword, similar to the `unsafe-inline` and `unsafe-eval` keywords in CSP. The aim of this is to discourage web developers from choosing to opt out of security features, as in a future progressive security version it may no longer be possible to do so, i.e. when the security feature is placed on the *enabled-by-default-no-opt-out* list.

The second optional directive is `unsafe-opt-in`, which can be used to re-enable features that are considered as *unsupported*. For features that are defined in this directive, a distinction needs to be made whether these are features that can only cause harm to the current website, or that could be used to attack other sites. An example of the former is the setting of the `document.domain` property: this relaxes



Is Progressive-Security header set?
No
Set *version* to default
Yes
Is defined version lower than default?
No
Is defined version higher than highest supported?
No
Yes
Set *version* to default
Yes
Set *version* to defined version
Remove features defined in directive from *enabled-by-default* list
Yes
Is unsafe-opt-out directive set?
No
Yes
Is unsafe-opt-in directive set?
No
loop for all features in directive
Can feature harm other sites?
Yes
Does user give permission to use feature?
Yes
Remove feature from *unsupported* list
No
No
Disable support for features defined in *removed* list
Disable support for features defined in (modfied) *unsupported* list
Show warning message when features defined in *deprecated* list are used
Enable features defined in (modified) *enabled-by-default* list
Enable features defined in *enabled-by-default-no-opt-out* list

Fig. 6. Flowchart of how the `Progressive-Security` header is parsed and enforced.

the same-origin policy for the website and prevents process-isolation at the origin-level, thereby only affecting the security of the website that chooses to still opt in to the feature. On the other hand, the Application Cache feature can be used to perform cross-site attacks, and should thus be handled differently. Concretely, we propose that the user should explicitly give permission to the website to use a legacy feature.

Summary. In Figure 6 we show a flowchart that reflects how the browsers builds the policy that needs to be enforced, and the actions it takes to enforce it. In essence, the browser will perform three main actions to enforce the policy:

1) Show warning messages when a deprecated feature is used.
2) Disable the features that are marked as either unsupported or removed. Web developers can only choose to re-enable features of the former set.
3) Enable security mechanisms that aim to protect against issues caused by deprecated features. Web developers can only opt out of a predefined set of security mechanisms.

D. Examples

Next, we analyze two use-cases of features that have been deprecated and removed from the web platform or that are in this process, and evaluate how this could have been done via the proposed progressive security mechanism. We look at Application Cache, a feature that was found cause various security issues and that could be leveraged to launch cross-origin attacks. We also consider the evolution towards origin-based process isolation, which is still ongoing at the moment. We opted for these examples to show the diversity and granularity of the proposed mechanism, as the deprecation and removal of these features are multi-step processes. For certain other features, this process is straightforward. For instance, the client-side XSS detection mechanism that could be controlled via the `X-XSS-Protection` header was simply removed from the web platform as it did not cause any compatibility issues and was not supported by all browsers. For the proposed progressive security mechanism, this coincides with adding the feature to the *removed* list.

In Section III-A we described the deprecation and removal process of application cache in detail, with a timeline of this process shown in Figure 5. In short, the use of AppCache was first deprecated in an insecure context, followed by a general deprecation and direct removal (without the option of opt-in) in insecure contexts. Next, the feature was disabled by default, but could still be re-enabled in Chromium by using the Origin Trial mechanism. Finally, the Origin Trial disappeared and support for AppCache was completely removed. In Table II we show how this process would have been if the proposed progressive security mechanism was used. By placing the `appcache-insecure` and `appcache` features on the different lists, it is possible to approximately replicate the deprecation and removal process.

One exception is that before the ultimate removal, Chrome opted to restrict the scope of AppCache based on the path on which AppCache was installed. Modifications of how the feature is implemented, is not supported via the proposed progressive security mechanism. Based on the deprecated and removed features that we considered, we find that such modifications are uncommon in practice. Another difference is that for the explicit opt-in, the Origin Trial mechanism was used, which gives the browser vendor detailed information on which sites still rely on the feature. For the progressive security mechanism, this opt-in occurs via setting a header, and this insight may thus be lost. This information could still be captured via large-scale crawls as performed by HTTP Archive[6] or through mechanisms such that Chrome's Real User Metrics[7].

Process isolation at the origin level is a feature that, at the time of this writing, is still under development. It is mainly blocked by the presence of two legacy features, namely the relaxation of the same-origin policy through the `document.domain` property, and cross-origin module shar-

[6]https://httparchive.org/
[7]https://developers.google.com/web/tools/chrome-user-experience-report

TABLE II
EVOLUTION OF THE APPLICATION CACHE AND ORIGIN ISOLATION FEATURES ACCORDING TO THE PROGRESSIVE SECURITY MECHANISM.

AppCache	Origin isolation
	`-- version 1 --`
`-- version 1 --`	`# deprecated`
`# deprecated`	`  + document-domain`
`  + appcache-insecure`	`  + cross-origin-wasm`
`-- version 2 --`	`-- version 2 --`
`# deprecated`	`# enabled-by-default`
`  + appcache`	`  + origin-agent-cluster`
	`# unsupported`
`-- version 3 --`	`  + document-domain`
`# removed`	`  + cross-origin-wasm`
`  + appcache-insecure`	
	`-- version 3 --`
`-- version 4 --`	`# unsupported`
`# unsupported`	`  - document-domain`
`  + appcache`	`  - cross-origin-wasm`
	`# removed`
`-- version 5 --`	`  + document-domain`
`# unsupported`	`  + cross-origin-wasm`
`  - appcache`	`# enabled-by-default`
`# removed`	`  - origin-agent-cluster`
`  + appcache`	`# enabled-by-default-no-opt-out`
	`  + origin-agent-cluster`

ing in Web Assembly. So far, these two features have been marked as deprecated in Chrome [21], [22] and will likely be marked as deprecated in the specification [23]. In a second stage, the browser intends to implement origin isolation, which can be controlled through the `Origin-Agent-Cluster` response header, and enable it by default [22]. This mechanism ensures that processes will be isolated at the origin level, and will thus prevent relaxation of the same-origin policy. To achieve the same effect with the progressive security mechanism, the two legacy features can be placed on the *unsupported* list, and the origin-agent-cluster security feature is placed on the *enabled-by-default* list. This allows web developers to still opt out of the origin isolation, and use either of the deprecated features. Note that this provides more granularity as the current process, as the features could be re-enabled independently.

In a final phase, the support for the legacy features might need to be completely removed, and the origin-agent-cluster security mechanism could then be enabled by default, without allowing an opt-out. This can be achieved by moving the legacy features from the *unsupported* list to the *removed* list, and moving the origin-agent-cluster feature to the *enabled-by-default-no-opt-out* list. Although we indicate that these changes occur in consecutive versions of the progressive security mechanism, in reality the timeline may differ, and might also depend on the effects on feature adoption at the time of deprecation. Nevertheless, these aspects should be agreed upon by the larger web community as part of an (ongoing) standardization effort.

V. DISCUSSION

A. Comparison to current state

The main difference with the way that deprecation and removal of web features is currently performed, is that this occurs through a synchronization effort, cooperatively between the different browser vendors and web developers. Currently, the intent to deprecate or remove a feature is announced via a public mailing list for Firefox[8] and Chromium[9]. Safari announces changes via release notes[10]. Although other browser vendors are typically polled on their stance of deprecating or removing a feature, each browser vendor mainly decides for themselves whether and when to deprecate or remove a feature. This two key disadvantages:

- **Compatibility.** If browsers disable support for features at different times, in an uncoordinated fashion, a situation may be created where a website become incompatible in one browser, but remains compatible with a different browser. From an end-user perspective, it may not be clear that this is due the removal of an outdated feature.
- **Deterrence effect.** A side-effect of the varying deprecation and removal schedules that lead to differences in website compatibility is a deterrence of removing support for features. More specifically, a browser vendor might be less inclined to remove a feature if this would result in certain (features of) websites no longer being available in their browser, possibly encouraging users to switch browsers. Consequently, legacy features that have a negative security impact may remain present in the web platform for an extended time.

With the proposed progressive security mechanism, features should be removed at approximately the same time in the different browser engines, depending on their release cycle.

Another benefit of the proposed progressive security mechanism is that it provides a unified framework for the deprecation and removal of web features. In the current state of the web, this process is mostly ad-hoc, possibly unique for the different features, and thus it might not be clear for web developers which exact actions they can take in order to remain compatible, or which time schedule they are provided with (which may differ depending on the browser). Despite the advantages of this unified approach, it comes with one key limitation, namely that it may limit the options for which actions can be taken in the deprecation process. For example, adjusting specific aspects of an API is not possible with the current mechanism. The only instance that we are aware of where this occurred, was in the removal process of Application Cache by Chrome, where the scope for the mechanism was limited, which improved security while remaining compatible with websites that relied on the feature.

B. Agreement

The synchronization of the entries on the progressive security list largely depends on the agreement among the different browser vendors. As these may have different priorities and goals, it could be that a single browser vendor wants to postpone the removal of a certain feature. To deal with such cases of disagreement, the browser vendors would ideally commit to follow the majority vote, accepting that this may

[8]https://groups.google.com/a/mozilla.org/g/dev-platform/
[9]https://groups.google.com/a/chromium.org/g/blink-dev/
[10]https://webkit.org/blog/

not always be in line with their development plans. In case the browser vendor would not follow the majority vote, this would be a deviation from best-practice with a known security impact, possibly affecting the reputation of the browser vendor. We believe that this could incentivize browser vendors to come to an agreement. In the worst case, when it is not possible to synchronize the progressive security list among browser vendors, the proposed mechanism would only provide a standardized way of deprecating and removing features, and would only minimally improve upon the current status quo.

C. Selection of features

Although the progressive security mechanism can be used for the deprecation, removal or introduction process of all web features, we believe that limiting the set of features to those that have a direct effect on security, either positive or negative, has several advantages. If the number of features tracked by the mechanism that are in the process of being removed remains limited, this provides a clear overview for web developers. Furthermore, by limiting the number features, this provides more flexibility in the timeline of when new progressive security versions are launched. We believe that this reduction of complexity outweighs the extension of the increased threat surface of deprecated features that have no direct security impact caused by the lack of a streamlined deprecation mechanism.

D. Feature removal

Depending on the feature, there are several ways in which it can be removed from the web platform. For features that have a very limited usage, and thus where removal would not cause significant compatibility problems, the relevant API can simply be omitted. However, for features that have a higher adoption rate, the removal of the API could possibly cause syntax errors, resulting in the script failing unexpectedly. Instead, the API could be altered to no longer have any effect; which will ensure that scripts that use it will not break. In general, although the exact way of removal of a feature from the web platform is not considered in the progressive security process, we believe that it is important for reasons of consistency and compatibility that this aspect is also standardized, ideally in the specification of the relevant API or the HTML specification.

VI. Related work

Deprecation and subsequent removal of features from the web platform is a largely uncharted research area. To date, only a single study on the deprecation of browser features has been conducted: in their work, Mirian et al. [20] perform an analysis of web deprecations and explore how the process can be improved to minimize the burden for users and web developers. In total, the authors analyze 117 deprecated features, and find that 25 of those were due to a security flaw or concern. In our study, we specifically consider features that have a security impact, as an improved deprecation and removal process will additionally safeguard users and websites.

Prior work on deprecation of APIs has mainly focused on studying its effect in various ecosystems. Zhou and Walker how API deprecation is performed in 26 open-source Java frameworks, and may often lead to outdated coding examples on platforms such as StackOverflow [24]. Another analysis of deprecation of Java APIs is performed by Sawant et al., who analyzed the impact of the deprecation of five popular APIs on over 25,000 clients [25]. In a later study, Sawant et al. study how developers react to deprecation of APIs, and find that the vast majority of API consumers in fact do not react to deprecated features [26]. Based on a user study of developers, the authors found that one of the main reasons why deprecations were not reacted to, was that the associated cost was not worth it, as the API would still continue to work. We believe that a more unified and standardized deprecation and removal process could provide more clarity to web developers on the concrete consequences and timeline.

Another related line of research explores how the threat surface of applications can be minimized by reducing, or debloating, the features that are available or accessible. In the context of web browsers, Snyder et al. performed a cost-benefit analysis to the various Web APIs that are offered compared to how frequently these are used [27]. The authors find that exposing all Web APIs poses a considerable security and privacy risk while providing little benefit for legitimate use-cases. Other work focuses on reducing the attack surface of applications by minimizing the portions of the code that is actually used. Qian, Hu et al. propose a system named RAZOR that does this for binary applications, and used it on the Firefox browser, leading to a reduction of code of approximately 60% [28]. In the context of web applications, Azad et al. show how the code can be significantly reduced, removing code associated with known vulnerabilities or that could be leveraged as a gadget [29].

VII. Conclusion

With an aim to standardize and improve the process of deprecation and removal of features that affect the security of users, browsers or websites, we propose a progressive security mechanism. By gradually advancing the minimally required version of security enhancements, the web platform can move towards improving the current state of security. The proposed versioned mechanism is based on past and ongoing ad-hoc deprecation and removal processes, and provides a similar granularity of actions that can be taken. It allows web vendors to provide a grace period during which web developers can choose to temporarily opt in to deprecated features, or opt out of new security features that will be enabled by default. Furthermore, it makes a distinction between legacy features that could only harm the website opting in to it, and those that can be used to launch cross-origin attacks. By standardizing and unifying the deprecation and removal process among different browser vendors, we believe that the deterrence effect of removing features could be reduced, and that this could facilitate the overall process for web developers.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity.

References

[1] A. Deveria, "Can I use," Feb. 2022. [Online]. Available: https://caniuse.com/

[2] Mozilla Developer Network (MDN), "MDN browser compat data," Feb. 2022. [Online]. Available: https://github.com/mdn/browser-compat-data

[3] Google Chrome, "Origin trials," 2022. [Online]. Available: https://googlechrome.github.io/OriginTrials/

[4] Mozilla.org contributors, "Site permission add-ons," 2022. [Online]. Available: https://extensionworkshop.com/documentation/publish/site-permission-add-on/

[5] S. Lee, H. Kim, and J. Kim, "Identifying cross-origin resource status using application cache." in *NDSS*, 2015.

[6] L. Herrera, "AppCache's forgotten tales," May 2021. [Online]. Available: https://blog.lbherrera.me/posts/appcache-forgotten-tales/

[7] L. Anforowicz, "In presence of NetworkService, AppCache may be used to bypass CORB," Nov. 2018. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=910210

[8] ——, "AppCache may be used to bypass CORB," Jan. 2019. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=927471

[9] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and conquer: Exposing cross-origin resource size," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 447–462.

[10] N. Williamson and N. Baumstark, "Security: UaF in Appcache," Sep. 2018. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=888926

[11] D. Vogelheim and M. West, "Origin isolation and deprecating document.domain," Nov. 2021. [Online]. Available: https://github.com/mikewest/deprecating-document-domain

[12] WICG, "Origin-keyed agent clusters explainer," Dec. 2020. [Online]. Available: https://github.com/WICG/origin-agent-cluster

[13] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[14] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, no. 6, 2010.

[24] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 266–277.

[15] F. Braun and M. Heiderich, "X-Frame-Options: All about clickjacking?" Dec. 2013. [Online]. Available: https://frederik-braun.com/xfo-clickjacking.pdf

[16] M. West, "Embedding should require explicit opt-in," Nov. 2020. [Online]. Available: https://github.com/mikewest/embedding-requires-opt-in

[17] XS-Leaks wiki, "Frame Counting," Oct. 2020. [Online]. Available: https://xsleaks.dev/docs/attacks/frame-counting/

[18] ——, "Cross-Origin-Opener-Policy," Oct. 2020. [Online]. Available: https://xsleaks.dev/docs/defenses/opt-in/coop/

[19] M. West, "COOP by default," Oct. 2020. [Online]. Available: https://github.com/mikewest/coop-by-default

[20] A. Mirian, N. Bhagat, C. Sadowski, A. P. Felt, S. Savage, and G. M. Voelker, "Web feature deprecation: a case study for chrome," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 302–311.

[21] L. Vahl, "Intent to deprecate: WebAssembly cross-origin module sharing," Jul. 2021. [Online]. Available: https://groups.google.com/a/chromium.org/g/blink-dev/c/nhmP8A61xk8/m/VuJXK8HQAwAJ

[22] D. Vogelheim, "Intent to ship: Origin isolation by default / deprecate document.domain," Dec. 2021. [Online]. Available: https://groups.google.com/a/chromium.org/g/blink-dev/c/_oRc19PjpFo/m/xUsWtryQAgAJ

[23] C. Lamy, "Deprecate cross-origin module sharing," Apr. 2021. [Online]. Available: https://github.com/WebAssembly/spec/issues/1303

[25] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 400–410.

[26] ——, "To react, or not to react: Patterns of reaction to api deprecation," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3824–3870, 2019.

[27] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 179–194.

[28] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1733–1750.

[29] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1697–1714.