A Security Analysis of Emerging Web Standards - Extended Version

HTML5 and Friends, from Specification to Implementation

Philippe De Ryck
Lieven Desmet
Frank Piessens
Wouter Joosen

Report CW 622, May 2012

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Security Analysis of Emerging Web Standards - Extended Version

HTML5 and Friends, from Specification to Implementation

Philippe De Ryck
Lieven Desmet
Frank Piessens
Wouter Joosen
Report CW 622, May 2012

Department of Computer Science, K.U.Leuven

Abstract

Over the past few years, a significant effort went into the development of a new generation of web standards, centered around the HTML5 specification. Given the importance of the web in our society, it is essential that these new standards are scrutinized for potential security problems.

This paper reports on a systematic analysis of ten important, recent specifications with respect to two generic security goals: (1) new web mechanisms should not break the security of existing web applications, and (2) different newly proposed mechanisms should interact with each other gracefully. Our analysis reveals several violations of these security goals, which should be addressed by the specifications.

In addition, we analyze the implementations of some of these standards in major browsers, and show that (1) even explicit security considerations in the standards are not consistently implemented, and (2) ambiguities and underspecification in the standards lead to suboptimal browser implementations.

In total, the security analysis reveals 45 issues, of which 12 are violations of the proposed security goals and 31 issues concern vague and ambiguous features in the specifications. Additionally, we found that 6 out of 11 explicit security considerations have been overlooked/overruled in major browsers, leaving secure specifications vulnerable in the end.

Keywords : HTML5, web application security, standards, specification.
CR Subject Classification : D4.6

1 Introduction

The past few years, web applications have known a significant growth in popularity. With billions of users, applications like Facebook, Twitter, Foursquare or anything provided by Google are omnipresent in most internet users' daily life. One constant in the evolution of web pages or web applications has always been the drive towards more and better client-side functionality, up to the level where web applications obtain privileges previously only available to desktop applications. Examples of the latter are client-side storage or easy access to peripheral devices (webcams, microphones, sensors etc).

The client-side functionality available to early web applications was limited to dynamic page construction, without any explicit support for features like client/server communication, interaction between browsing contexts or local storage of data. The introduction of an explicit client/server communication mechanism, XMLHttpRequest [33], meant a major extension of client-side functionality for web applications. The *Asynchronous JavaScript And XML (AJAX)* development method, which allows a web application to submit information in the background, and only modify or change a selected part of the web page, revealed the true potential of the XMLHttpRequest extension. Additionally, the recently proposed Web Messaging API [15] enables communication between browsing contexts, a feature long-desired by mashup developers. Currently, a vast amount of new, exciting features are appearing in browsers as standards are being developed or are nearing the W3C recommendation status. These features include embedding media elements, registering remote content handlers in the browser, an iframe-based sandbox, media capture APIs, several client-side storage APIs, access to a device's location as well as various system properties (battery level, CPU properties, sensors, etc).

Unfortunately, an extensive set of client-side features poses an attractive target for attackers, making the user into a potential victim. For example, a media capture API is interesting to include captured media in a web application, but can perhaps be abused to steal captures without the user's knowledge? Remote handlers can be useful to streamline the process of opening text documents with an online office suite, but could they also be used by an attacker to hijack documents?

In this paper, we report on a structured security analysis of a large subset of these emerging web standards, where we not only analyze each specification in isolation, but also investigate potential interactions between co-existing features of multiple specifications. We have studied the following list of ten W3C specifications:

1. HTML5 [14]
2. Media Capture API [30]
3. Web Messaging [15]
4. XMLHttpRequest Level 1 [33]
5. XMLHttpRequest Level 2 [34]
6. Cross-Origin Resource Sharing [32]
7. Uniform Messaging Policy [6]
8. Web Storage [16]
9. Geolocation API [24]
10. System Information API [29]

A key question for any security analysis is: what constitutes a security violation? In other words, what are the security goals we are analyzing against? Obviously, different mechanisms will have different security goals. For instance, the sandboxing mechanism specified in the HTML5 specification has the goal of *isolating* content in the sandbox, whereas the media capture specification might have the completely different goal of appropriately *guarding access* to the API to capture audio or video. It is *not* our objective to analyze each mechanism for compliance with these mechanism-specific security goals. Whether each proposed mechanism correctly satisfies its mechanism-specific security goals is an interesting question, but this question should be addressed separately for each mechanism, most likely using mechanism-specific methods. Instead, we identify two generally applicable security goals, that apply to all proposed mechanisms.

The first security goal states that new mechanisms introduced in the web platform should not weaken the security of existing (legacy) web applications. For example, the security of existing web applications might rely on the enforcement of the same-origin-policy in the browser. The introduction of new cross-origin communication mechanisms should not break the security of these existing applications. Akhawe et al. [3] identify the same generic security goal, and call it the *preservation of security invariants*.

The second security goal states that separately specified mechanisms meant to co-exist and to be used together should interact gracefully. The introduction of one newly proposed mechanism should not break or weaken the security of another newly proposed mechanism. We call this second goal the *graceful interaction of co-existing features*. For example, a newly proposed mechanism for cross-origin communication should be able to handle the introduction of new kinds of origins (such as the *unique* origin in sandboxes).

From the security analysis of the ten standards, we conclude that these new web standards already achieve a high level of security. This is most probably due to the fact that the emerging standards explicitly endorse the *secure-by-design* design principle [14], taking security into account from the beginning, and not as an afterthought. As such, the analysis does not reveal major unconditional security issues. We do however identify several violations of one of the stated security goals under certain application-specific circumstances. We describe several examples in the paper, and provide suggestions on how to deal with them.

Of course, standards are only part of the story. Even if the specification of the standard is secure, implementations in browsers might violate security. We investigate the compliance of major browsers, and we show that even if the standards make explicit (mandatory) security considerations, implementations are often not compliant. Finally, we also identify cases where ambiguity or under-specification in the standard results in inconsistent and suboptimal implementations in browsers.

In summary, the contributions of this paper are:

- A security analysis of ten secure-by-design web standards, based on two important security goals: the *preservation of security invariants* and the *graceful interaction of co-existing features*.
- A validation of compliance of mainstream implementations to explicit (mandatory) security considerations stated by the specifications.
- An analysis of the impact of vague and ambiguous features in specifications on mainstream implementations.

In the remainder of this paper, we first present the approach and the process of the security analysis and give a summary of the results (Section 2). Next, we discuss these results in more detail: a representative sample of violations of the security goals (Section 3), several examples of non-compliance issues in major browsers with respect to explicit security considerations (Section 4), and examples of under-specification issues that lead to inconsistent and suboptimal implementations (Section 5). Section 6 presents related work, and in Section 7, we discuss the results of the analysis and propose guidelines to improve the future state of web security. We conclude in Section 8.

2 Overview of the Security Analysis

The broad set of new and emerging web standards covers an immense range of functionality and features, from detailed HTML parsing rules to rich JavaScript APIs and security design guidelines. In the analysis, we focus on the newly-added core features of HTML5 [14] (e.g. offline application caching and sandboxed iframes) as well as the new features provided by JavaScript APIs (e.g. client-side storage and cross-origin communication). These JavaScript APIs are available to any client-side script embedded in a website, either directly coming from the website owner, or integrated from a (potentially less trusted) third-party service provider, as is the case for online advertisements, web analytic frameworks, mashups compositions and many more.

2.1 Scope

The ten W3C specifications studied form a representative subset of the emerging web standards, and are per category depicted in an abstract model of modern web standards (Figure 1). Central in the abstract model is the window object, which provides access to the page (via the document), as well as all JavaScript functionality. Within the window are event handlers and the DOM tree, which we do not include explicitly, unless used by other parts of the model. The window can be restricted by a sandbox, which is part of HTML5. The building blocks around the window offer a tremendous amount of functionality to a web application, such as communication between components, embedding and capturing media (audio, video, images), requesting information from the device or its sensors, communicating with external services, storing data on the client-side and caching an entire application or component. Note that for the "Client-side storage" building block we focus on the Web Storage specification [16], which contains

most of the complexity with regard to storage areas and their access rules, compared to other client-side storage specifications [20, 13].

In order to keep the analysis focused, we considered the UI building block to be out of scope (with the exception of end-user interactions for user consent: these are in scope). Similarly, we discarded parts of the HTML5 specifications describing the HTML parsing and rendering, usability guidelines and API development guidelines.

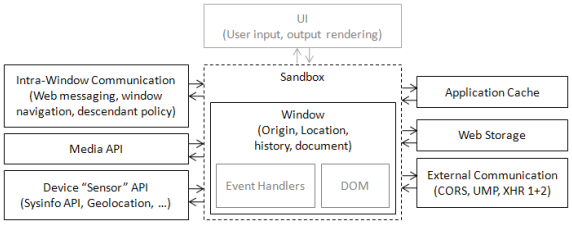

Figure 1: The abstract model of emerging web standards

2.2 Attacker model

For this security analysis we consider both the *web attacker* and the *gadget attacker* to be valid attacker models [4]. As defined by Barth et al., a web attacker is a malicious principal who owns one or more machines on the network where the user gets and renders content from, whereas the gadget attacker has one additional ability: the integrator embeds a gadget of the attacker's choice.

Concretely, we assume that a user can visit honest sites, honest sites unknowingly incorporating malicious content (in case of the gadget attacker) or malicious sites (in case of the web attacker), using an honest, specification-compliant browser. Both attackers aim to exploit weaknesses in the specifications to compromise one of following assets:

1. Application-specific data and functionality (page contents, DOM elements, JavaScript code, etc.)

2. End-user data and identity (cookies, credentials, locally stored data, user/media input, etc.)

3. Contextual data (device/sensor information, origin, etc.)

2.3 Analysis Process

Since the security analysis covers ten standards[1] (approximately 1000 pages of specification), we applied an efficient, but systematic and repeatable analysis approach. The approach consists of two manual steps (the *distillation step* and the *security assessment step*), and was applied by a small team in multiple iterations to gradually go from a broad sweep over the specifications to more detailed analysis results. In order to preserve consistency, the team remained stable during the entire analysis.

The main purpose of the *distillation step* is to extract useful information from the full specification text. For each specification, we explicitly distill (a) the *functional capabilities* offered by the specifications, (b) the *user involvement*, which describes how the specification involves the user for consent, and (c) the *security/privacy considerations*, either explicitly listed in the specification or implicitly assumed to hold. For the most complex capabilities, we detailed the algorithms using flowcharts and/or clarified complex access control systems using permission tables.

To illustrate the distillation process, we show a partial extract of the analysis of the Geolocation API [24]. Some of the functional capabilities offered by the specification are:

C1. One-time permitted access to the current location using one-shot access or a monitoring function

C2. Continuous permitted access to the current location using one-shot access or a monitoring function

The Geolocation API involves the user as follows:

U1. Give consent for an origin to access location information

U2. Manage long-term permissions for access to location information

Finally, some of the explicit (E) and implicit (I) security/privacy considerations made by the specification are:

E1. Accessing the location requires explicit consent, where the UI must show the URI of the document's origin

E2. The permission UI must show the URI of the document's origin

I1. All sites within an origin have the same trust level

I2. The user unambiguously knows which part of the page wants to know the location (e.g. a framed document)

[1] We used the specifications available in June 2011 for the security analysis

As part of the *security assessment step*, we have investigated each specification in isolation and in co-existence with other specifications, based on the distilled information. The goal was to detect and identify weaknesses in the specifications leading to an increase of attack surface, either by (1) newly added features not preserving security invariants for existing applications, or by (2) ungraceful interaction of co-existing features. An example of the former is Cross-Origin Resource Sharing enabling POST requests with an arbitrary body format (see Section 3, Example 2). An example of the latter is the lack of sender information for Web Messaging when used in a sandbox (see Section 3, Example 4).

In addition, we have assessed the quality of the specification in terms of well-definedness, i.e. does the specification clearly define every security-relevant design decision, or does the specification leave some room for arbitrary and/or ambiguous interpretation. Similarly, are the design decisions consistent across the various specifications.

Finally, we have verified the compliance of major browsers with the explicit security considerations, mentioned in the specifications and enlisted as part of our analysis. Since no test cases for these listed security considerations were available, we explicitly implemented a test case for each security consideration and executed them in each subjected browser.

2.4 Results of the Security Analysis

Performing the security analysis as described before has revealed 45 security issues, distributed over the 10 specifications as shown in Table 1. The identified issues are presented in four categories, respectively (a) is the functionality well-defined and secure on its own, (b) does it respect isolation properties and restricted origins, (c) is the specification consistent with similar specifications, and (d) does the specification have unrealistic dependencies on user involvement to achieve security. The majority of these 45 issues, i.e. 31 issues, relates to under-specification and inconsistencies across the specifications. A smaller fraction, i.e. 12 issues, increases the attack surface by breaking one of our two security goals, namely the *preservation of security invariants* or the *graceful interaction of co-existing features*. This paper focuses on the most significant issues, but the full analysis report covers each issue in more detail [9]. As part of the analysis, we have reported each of these issues to the appropriate standardization bodies.

Note however that we did not discover any unconditional security violations in either of the specifications, despite the thorough analysis approach. This finding seems to be consistent with the rigorous development approach and the secure-by-design principle, and is a very positive result for modern web standards. We did however discover several conditional violations, i.e. weaknesses in the specifications that, depending on the specific application, potentially increase the attack surface. These violations are discussed in detail in Section 3.

Since the specifications are carefully designed and pay special attention to security, for example by explicitly listing important security considerations, we also investigated various implementations of the specifications, to see whether these guidelines are actually followed and thus result in a secure implementation. Section 4 discusses this investigation of compliance with explicit security considerations, revealing 6 cases of non-compliance, out of 11 explicitly tested security considerations. We are currently in the process of reporting each of these non-compliance cases to the respective vendors.

Several specifications contain ambiguities or lack details about interaction of co-existing features from different specifications. In Section 5 we discuss these under-specification issues, as well as the consequences they have on the implementations.

3 Increased Attack Surface

Introducing new features and functionality in the web application ecosystem not only greatly benefits newly developed applications, but also impacts legacy applications, which are not immediately updated to cope with the new features. Any of these new features can lead to a violation of the *preservation of security invariants* or *graceful interaction of co-existing features* security goal, resulting in an increase of attack surface for newly developed or legacy applications.

In the remainder of this section, we discuss 5 examples where newly introduced features and functionality violate one of the security goals.

[Ex. #1] HTML5: Injection of Submission Controls

The HTML5 specification [14] enables the use of form controls outside of a `form` element, to overcome the lack

	Well-Defined/Secure	Isolation Properties	Consistency	User Involvement
HTML5	8	3	2	2
Web Messaging		1	2	
XHR 1 + 2	1			
CORS	2	1		
UMP				
Web Storage	3	1	1	
Geolocation API	5	1	1	1
Media Capture API			3	
System Information API	3	1	1	2
Totaal	22	8	10	5

Table 1: An overview of the identified issues, categorized according to the criteria in the ENISA report [9]

of support for nested forms. Such a form-associated element can appear anywhere, but is associated with a form within the document. The associated form is either the nearest ancestor form, or the form with the ID explicitly referenced by the control using the `form` attribute (see Listing 1, line 6). Additionally, the specification allows submission controls to provide attributes that change the form behavior when this specific submission control is used. Examples are `novalidate` that disables form validation and `formaction` that changes a form's destination (see Listing 1, line 4 and 7).

```
1 <form id="one">...</form>
2 <form id="two">...</form>
3 <!-- Associated with nearest form "two" -->
4 <input type="submit" ... novalidate />
5 <!-- Associated with form with id "one" -->
6 <input form="one"
7      type="submit" ... formaction="http://..." />
```

Listing 1: Use of the newly introduced form attributes

The combination of both new features creates a new attack vector that allows an attacker to inject a form submission control, associated with an existing form on the page. By changing the form's destination using the `formaction` attribute, an attacker can trick the user into submitting a form located anywhere on the page to an attacker-controlled destination. After having captured the form's parameters, the attacker can automatically resubmit the form to the original destination, making the attack fairly stealthy.

One simple attack scenario is the stealing of an auto-completed form, such as a login form, for which the browser automatically enters the appropriate values, even if the user had no intention of logging in to the site. If the attacker can trick the user into clicking the injected submission control, the form will be submitted to the attacker's website. Tricking the user into clicking an oddly placed button may be difficult, but submission controls can also be styled with custom images, allowing full integration into the victim website. A second, more complex attack scenario involves the stealing of hidden security tokens from a form. The de facto countermeasure against cross-site request forgery (CSRF) attacks is the use of a unique token to authenticate a future request [38]. The security model of this countermeasure depends on the attacker not being able to extract such tokens from the page of the victim site, due to the same-origin policy separation, an assumption violated in this attack scenario.

Note that these attack scenarios do not require to execute any injected JavaScript, but only require the possibility to inject a previously harmless form control. This means that sites without cross-site scripting (XSS) vulnerabilities or sites deploying explicit defenses against injection attacks, such as Content Security Policy [26], can still be vulnerable. Additionally, traditional form stealing using HTML injection [37] either depends on the position of the vulnerability within the page or the capability to inject a form element in front of another form. The attack presented here is not dependent on script execution nor on a specific position of the injection vulnerability.

Web developers can protect themselves against this attack vector by making sure that input validation filters prevent the injection of submission controls. It suffices to prevent the use of the behavior-changing attributes on user-supplied form controls, although it is wise to prevent the use of the form-association attribute as well.

[Ex. #2] CORS: Arbitrary Body Format

The Cross-Origin Resource Sharing [32] and XMLHttpRequest Level 2 specifications [34] enable cross-origin requests from JavaScript, where previously only same-origin requests were allowed. The main idea of CORS is that the client provides the server the necessary information to make a decision about whether an origin can fetch a resource or not. The decision is communicated to the client by means of response headers, where it is enforced. If an origin is not allowed to fetch the resource, the requesting script will not have access to the response data.

Technically, the algorithms elaborating on this idea distinguish between *simple requests* and *actual requests*. Simple requests are requests that could previously be sent with other HTML elements, such as GET requests or POST requests using forms. A simple CORS request is sent to the server, and based on the CORS headers in the response, the browsers decides whether the script can access the response contents. Actual requests are new types of requests, such as requests with custom headers or cross-origin PUT or DELETE requests, and can not be sent cross-origin with traditional techniques. This design decision prevents introducing new vulnerabilities in legacy applications.

The treatment of simple CORS requests is based on the assumption that these cross-origin requests could previously be made using commonly available HTML elements. However, one important difference between a POST request resulting from a form submission and a scripted POST request is the format of the request body. A form body is either in the text/plain or application/x-www-form-urlencoded format, which is restricted to *key*=*value* data separated by & (Listing 2), or the multipart/form-data format, which contains multiple sections separated by a self-defined boundary [14]. As we identified during the analysis, a simple cross-origin POST request sent from JavaScript is still restricted to these content types in its header, but can provide a body in an arbitrary format, since compliance of the body format to the specified content type is never checked. This means that an attacker is able to send POST requests with an arbitrary body format to any origin, but will most likely not be able to access the response, since the appropriate response headers allowing the requesting origin access to the resource will be missing.

```
1 username=Homer&password=beeranddonuts
```

Listing 2: POST body in the text/plain or application/x-www-form-urlencoded format

A concrete attack scenario is an application offering an authenticated API, for example using the JSON format (Listing 3) . Where previously only pages within the origin of the application were able to access the API, CORS enables all origins to make requests to the API. Even though the API will not add the appropriate headers to the response, the request will still be processed, potentially resulting in unintended server-side changes.

```
1 { username: "Homer",
2   password: "beeranddonuts" }
```

Listing 3: POST body in the JSON format

Before the introduction of CORS, the body format of cross-origin POST requests was subject to strict restrictions. With simple CORS requests, explicitly aimed at not enabling more features than existing HTML elements offer, cross-origin POST requests can use an arbitrary body format, opening up existing applications to cross-origin requests, where previously only same-origin requests were possible. This is a violation of the first security goal, since a new attack vector for legacy applications is introduced.

Concretely, web developers can protect against this attack vector by validating the format of incoming requests against the format specified in the header.

[Ex. #3] CORS: Sandboxed Sender

As discussed with the previous violation, the HTML5 specification offers the possibility to sandbox an iframe and impose restrictions on the content. One important restriction is the possibility to have the browser assign a unique origin to the content of the iframe. This ensures that the content of the iframe is separated using the same origin policy [36], allowing strict isolation of content, even within the same origin. An important nuance of this unique origin is that the serialization to a string equals *null* [14], meaning that if the origin is ever used for checks or in a header, the value *null* will be used instead.

The effect of a unique origin being serialized to *null* can be observed with the use of CORS in a sandboxed iframe. A CORS request contains an origin header defining the origin where the request originated from. In a sandboxed iframe with a unique origin, the value of the origin header will be *null*. Even though the server has no concrete origin information about the request, CORS specifically supports granting the *null* origin access to the resource by adding the appropriate

response headers. This behavior enables the use of the value *null* as a wildcard, since any document can sandbox itself in an iframe and send requests with an origin-value of *null*. The main difference with the already available wildcard in CORS (∗) is that the *null* origin allows the use of credentials, a feature that is explicitly forbidden by the explicitly defined wildcard.

A concrete scenario where the use of CORS from a sandbox poses a problem is a legitimate site that wants to expose an API to a selected set of origins that use a sandboxed iframe for additional security. Since the site offering the API no longer receives origin information, it is unable to enforce access control restrictions and can either disallow the use of the API or allow the use of the API from a *null* origin. The former prevents the sites using the API from deploying a sandboxed iframe, the latter forces the site offering the API to open up its API to all origins. The combination the sandbox attribute and the CORS specification is a violation of the second security goal, stating that co-existing security mechanisms should interact gracefully.

Web developers should never allow the use of a *null* origin in a CORS request, unless they explicitly intend to provide wildcard-accessible authenticated APIs. Explicitly checking for the *null* value and immediately returning a response is the safest approach. The wildcard (∗) offered by CORS can be used to grant any origin (unauthenticated) access to a specific part of the application.

[Ex. #4] Web Messaging: Sandboxed Sender

The Web Messaging specification [15] enables communication between browsing contexts, either by sending a single message or by establishing a message channel based on ports. We are interested in the former mechanism, where a sender simply invokes the `postMessage` function on the destination browsing context, for example a framed document. The two relevant arguments of the function are both strings and contain the message and the destination origin. Before delivering the message to the receiver, the browser checks the specified origin against the origin of the document. If the origins match, a message event is triggered in the receiving context, allowing an event handler to process the message. The specification offers two important guidelines for web application developers: first, they should check the origin of the sender, which is available in the message event, and second, they should ensure that the received message is in the format that they expect.

A problem arises when a sandboxed document with a unique origin uses the `postMessage` function to send a message to another context. When the receiver inspects the incoming message and checks the origin, the value *null* is returned [14]. This effectively prevents the receiver from correctly authenticating the message.

A common use case of both the sandbox attribute and the Web Messaging functionality are web mashup components that both need to be isolated from each other, but also need to collaborate. Whenever both features are used in combination, the receiver has no choice but to accept messages from the *null* origin. This potentially compromises security, especially because it is easy for any document to create a sandboxed version of itself.

Similar to the example #3, the combination of the sandbox attribute and the Web Messaging functionality is a violation of the second security goal, since the sandbox security feature does not gracefully interact with the origin checks provided by the Web Messaging specification. This problematic behavior even directly violates security guidelines explicitly stated in the specification.

Concretely, web developers can protect against this attack vector either by not accepting messages from a *null* origin, or by explicitly detecting such requests and carefully handling them under the assumption that it is sent by a random origin.

[Ex. #5] HTML5: Cross-Site Scripting Continued

The HTML5 specification [14] adds numerous new elements and attributes, which often allow the execution of arbitrary JavaScript. Such functionality can lead to cross-site scripting vulnerabilities if a site does not prevent users from adding content containing these elements and attributes. The HTML5 Security Cheat Sheet project [12] already contains an extensive collection of both old and new attack vectors, with new HTML5 features collected in a separate category.

Cross-site scripting attacks using newly introduced elements or attributes are an excellent example of a violation of the first security goal, which states that a legacy application should not become (more) vulnerable through the introduction of new features. In this particular case, legacy applications using a filtering mechanism on user input may not account for the new elements and attributes, and thus become vulnerable to cross-site scripting attacks.

Cross-site scripting is certainly not a new attack, but remains very important and omnipresent, as indicated by the introduction of new security initiatives such as Content Security Policy [26] or HTML5 Sandbox [14]. Since a good reference work already covers new HTML5 cross-site scripting vectors, we do not focus on cross-site scripting attacks in our security analysis. However, we would like to encourage developers to update their XSS protection mechanisms to take into account the newly identified XSS vectors as well.

Summary

In this section, we have illustrated via five examples that the specifications in multiple places violate one of the security goals stated, namely the *preservation of security invariants* and the *graceful interaction of co-existing features*. These violations lead directly to an increase in attack surface for legacy and/or newly developed web applications. Moreover, we provided guidelines for developers to deal with the violations, till the specifications (and major browsers) have been updated.

4 Browser Compliance

One of the design principles of emerging standards is *secure-by-design* [14], taking security into account from the beginning, and not as an afterthought. The specifications aim to securely specify their main functionality, but also include explicit security considerations for implementers of the specification or for web application developers.

However, the security of new features starts with secure specifications, but also depends on implementations actually following the specifications. As a second part of our security analysis, we verify whether existing implementations comply with their respective specifications. In order to make such a compliance test on a set of ten specifications feasible, we focus on the explicit security considerations that we distilled during the distillation step of the security analysis. These explicit security guidelines are very clearly stated within the specifications and leave no room for interpretation. Verifying the level of compliance against these considerations gives an indication of the overall security compliance of an implementation. Of course, the full level of compliance can only be verified with a complete functional test suite, which is a huge effort, and is already coordinated by W3C [35].

From the specifications, we distilled 48 explicit security considerations, of which 30 directly apply to implementers of the specification. Of these 30 considerations, only 11 are applicable to implemented features. We tested the compliance of major browsers against those 11 explicit security considerations, and conclude that with 6 compliance issues out of 11 explicit security considerations, implementers fail to follow several explicit security considerations.

We used working draft versions of the specifications, which are stable compared to the editor's drafts. We carefully checked that the distilled explicit security considerations are still present in the most recent version of the specifications, to prevent conflicts with very recent implementations. Additionally, we made sure that there was a time window of at least 5 months between the release of the working draft and the release of the implementation[2], which should be sufficient to allow the implementers to update their specification to be compliant with the latest working draft.

In the remainder of this section, we zoom in on the results of one representative standard, Web Storage [16]. We selected the Web Storage specification because of its complexity and nearing recommendation status. First, we discuss the relevant functionality of the Web Storage specification, followed by a discussion of the compliance issues we discovered in the mainstream implementations.

Web Storage

The Web Storage specification offers an easy way to store text-based data using key/value pairs. The specification describes two different origin-based storage areas, localStorage and sessionStorage. The functionality for both areas exists of storing/retrieving/removing an item, retrieving the number of stored items, retrieving a list of stored keys and clearing the whole storage area.

localStorage Each origin has its own localStorage area, which is available to all documents within that origin, loaded anywhere within the browser. The data stored in the localStorage area is permanently stored, and will be available after the current session has ended.

[2]We used the following browsers: Mozilla Firefox 8.0.1, Google Chrome 15.0.874.121, Opera 11.51, Safari 5.1.2 and Internet Explorer 9.0.2

sessionStorage Within one top-level browsing context, each origin also has a sessionStorage area. This area is only available to top-level context and its descendants. Two documents of the same origin in different top-level browsing contexts (e.g. two separate tabs) have separate sessionStorage areas. The lifetime of a sessionStorage area is limited to the lifetime of the current browsing session, which can be extensive due to session resuming mechanisms.

[Ex. #6] Non-Compliance with Domain Relaxation

An explicit security consideration in the Web Storage specification (section 4.3.1) states that the localStorage attribute should no longer be available after a script has used the `document.domain` attribute to change its effective script origin. This prevents the abuse of storage areas belonging to a subdomain after setting `document.domain` to a less strict value. Take for example a victim page from `victim.example.com` and an evil page from `evil.example.com`. The victim page uses a variable to refer to its localStorage attribute, and the same origin policy separates both pages from each other, preventing the evil page to access the victim's page storage area. If both pages set their `document.domain` to `example.com`, they are allowed to access each others resources. If the victim script's reference to the localStorage of `victim.example.com` would still be valid, the evil script would be able to access this storage area.

An investigation of the compliance of major browsers to this explicit security consideration reveals that Internet Explorer, Chrome, Firefox, Safari and Opera all fail to comply with this requirement. All browsers allow access to the original localStorage area after the script has relaxed its origin to the parent domain.

[Ex. #7] Non-Compliance with Quota Guidelines

Another security consideration discusses how implementations should behave with regard to storage quota. First of all, the specification states that an implementation should enforce a quota on the space allowed for storage areas. Additionally, the specification states that an implementation should guard against sites using subdomains (e.g. `a1.example.com`, `a2.example.com` ...) to circumvent storage limitations for the main domain (`example.com`).The specification leaves a little room for deviating behavior by using the keyword *SHOULD* [5], meaning that valid reasons may exist to choose another course, but the full implications need to be understood before doing so.

All implementations are compliant with the first consideration by enforcing a quota limit on available storage space. The second consideration is met by Internet Explorer, Opera and Firefox. Chrome and Safari both fail to prevent using subdomains to subvert the storage limit, effectively allowing a denial of service attack. Verification of this denial of service attack shows that localStorage can be used to completely fill all free space on the disk with bogus data.

Summary

In this section, we have illustrated via two examples that even explicit security considerations in the specifications are often overlooked/overruled in the mainstream implementations. As a consequence, securely specified features might in practice still be vulnerable due to this mismatch.

5 Under-specification Consequences

During the security analysis, we also discovered that several specifications contain features that are under-specified. Such under-specification can occur when some functionality is specified rather vaguely or when a feature does not account for external effects, such as features from other specifications (e.g. a sandboxed iframe) or omnipresent browser features (e.g. private browsing mode). We have identified 31 important inconsistencies and under-specifications, and investigated their effect on the corresponding implementations.

In the remainder of this section, we zoom in on the results of the Geolocation API [24]. We selected the Geolocation API specification because of its maturity and available implementations. First, we discuss the relevant functionality of the Geolocation API, followed by a discussion of three important under-specification issues.

Geolocation API

The Geolocation API offers scripted access to geographical information associated with the hosting device. The API offers two ways of accessing location information: one-shot access (using the `getCurrentPosition` function), which gives a provided handler a single position object, representing the current position, and continuous monitoring of location

changes (using the `watchPosition` function), which calls a provided handler whenever the location changes significantly. Access to location information is subject to user consent, who has to explicitly grant permissions to an origin. The UI asking for the user's consent has to display the URI of the document's origin. Additionally, a granted permission can be stored indefinitely by the browser. The browser should also provide easy access to interfaces that enable permission management of granted consents.

[Ex. #8] Under-informed User Consent

The Geolocation API depends on a permission system to obtain a user's consent before actually retrieving any location information. Unfortunately, this permission system is only vaguely specified and contains several ambiguities. A first observation is that even though the Geolocation API explicitly differentiates between one-shot access or continuous monitoring, the permission system does not. Access to both functions is restricted by one type of permission, which is valid once or can be stored until revoked. Verification in current implementations reveals that effectively no browser informs the user about the type of permission the user is consenting to.

[Ex. #9] Suboptimal Permission Management

The Geolocation API states that previously granted permissions should be easily revocable, without any further requirements. Unfortunately, implementations have different opinions about an easy-to-use interface, often in the user's disadvantage. For instance, Internet Explorer does not have a list of permissions and only allows the revocation of all granted permissions. Even worse is Firefox, which not only has no list of granted permissions, but requires a user to revisit each site that has a stored permission, open the page information dialog, before the granted permission can be revoked.

[Ex. #10] Lack of Awareness of Permission Use

A risk introduced by stored permissions is that a site is able to use those permissions at any given time, without the user being aware of this. The Geolocation API acknowledges this risk and advises implementers to enable user awareness. An investigation of current implementations shows that no browser actually uses an awareness system when a site actively uses a permission. Chrome is the only one with a location-indicator, but this indicator only reflects whether permission has been granted to a site, not when a permission is actually used.

[Ex. #11] API Behavior in Unique Origins

The Geolocation API does not specify alternate behavior in contexts with a unique origin. This means that according to the specification, when asking for permissions the browser should show *the URI of the document origin*, which in this case is a globally unique identifier.

An investigation of the Chrome implementation shows that accessing one of the functions of the Geolocation API results in an error with code `PERMISSION_DENIED`. According to the specification, this means that the location acquisition process failed because the application origin does not have permission to use the Geolocation API. Unfortunately, the same error is used for situations where the user denies a site access to location information, making it impossible for an application to distinguish between both.

Summary

In this section, we have illustrated via three examples that the specification studied suffer from under-specification and ambiguity, leading to inconsistent and potentially insecure implementations as a consequence. Moreover, the last example (under-specification for sandboxed executions) is not specific to the Geolocation API, but appeared across numerous specifications.

6 Related Work

There is a vast amount of related work on web application security, and in this section we will only briefly discuss a few relevant subsets. First, we will highlight the majority of related work, which typically focuses on patching basic flaws in the classical web paradigm (either in the code itself, or via client- or server-side hardening). Next, we will describe a recent and gradual evolution in web security towards *patching standards* and *sandboxing environments* imposing additional server-driven constraints on the integrated content. Finally, we discuss previous security assessments on web application standards, and their ability to address some of the core security problem in web

security right at the root cause, namely directly in the specifications themselves.

An inherent problem of web security is the vast amount of deployed browsers and applications, each from multiple vendors in multiple versions. Making fundamental security improvements while still maintaining existing functionality is hard, as documented by the traditional patching of the basic web paradigm. For most classical web application vulnerabilities, such as cross-site scripting, SQL injection and cross-site request forgery, adequate safe-practices and countermeasures have been proposed [1, 27, 38], but are unfortunately not always consistently deployed.

Several recently proposed security policies also address traditional web application vulnerabilities and offer a site owner more fine-grained server-driven control over existing features, even in the case of legacy applications. A policy preventing frame-based attacks, such as clickjacking, is X-Frame-Options [17]. Content Security Policy [26] offers protection against injection attacks, on one hand by strictly limiting the sources where content can be loaded from, on the other hand by preventing in-line script execution, thus effectively disabling script injection attacks. Finally, the Do Not Track [19] policy concerns users' privacy by letting them opt-out of tracking. This policy does not enforce compliance, but major advertisement networks have already voiced their support for the initiative.

At the same time, the controlled execution of third-party JavaScript (as is typically the case in mashup compositions and online advertisements) is becoming an active research domain. This is especially relevant because of the obviously increased set of features available to JavaScript developers. In [7], De Ryck et al. give an overview of mashup requirements, focusing on isolation, interaction and communication. They also stress the need for fine-grained control over third-party component's behavior. Common approaches towards fine-grained behavior control are the use of security wrappers in JavaScript [23, 18], policy-controlled client-side sandboxes [28, 22] and the use enhanced browser support [21, 31].

Web standards have been scrutinized for security before, with positive results. The most relevant related work, discussed below, typically goes into great detail on narrowly scoped functionality. Unfortunately, detailed analysis techniques do not scale well to a set of multiple specifications and their potential interactions. Our work is complementary to such detailed analysis because of its broad set of features and functionality, and the explicit focus on potential consequences of interactions between these features.

Barth et al. investigated the security of frame communication in browsers [4], discovering both weaknesses in fragment identifier messaging, an unintended communication channel, and *postMessage*, the designed communication channel. Their work goes into great detail on a specific aspect of client-side functionality, using very specific security goals. Similarly, Akhawe et al., who use formal modeling to find design vulnerabilities in web specifications [3], specifically focus on web security mechanisms. The formal model is also useful to evaluate the security of newly proposed countermeasures [8]. While their approach is successful on the presented case studies, its feasibility is dependent on the limited scope of features.

Other related research is by Rydstedt et al. on the effectiveness of clickjacking defenses [25] and of Aggarwal et al. about the actual security of private browsing modes [2]. In this area, we also consider the work of Doty et al. about privacy issues in the Geolocation API [11] and the work of Heiderich et al. on cross-site scripting attack vectors in HTML5 [12] to be highly relevant.

7 Discussion

During the analysis, we have observed several topics for further reflection. First, we have identified directions to further enhance the state-of-security of new specifications and implementations (Section 7.1). Next, we would like to question and challenge the current evolution towards user-managed security within web applications (Section 7.2). Finally, we ask standardization bodies as W3C to consider standardizing private browsing contexts (Section 7.3).

7.1 Enhancing Specification Security

Even if the web specifications studied in the analysis already achieve a high level of security, they still violate the two security goals, *preservation of security invariants* and *graceful interaction of co-existing features*, on various occasions. These violations can be ascribed to the fragile balance between functionality and security, a thin line to walk, especially in the web application ecosystem, with the standards simultaneously serving browser vendors, web developers and users. This analysis focuses on imbalances at the expense of security,

but likewise, sacrifices will have been made in favor of security and at the expense of functionality.

The violations of the security goals documented in the analysis should be addressed by the specifications in the future. Whenever possible, the functionality should be updated and adapted to better respect the balance with security. Whenever there is a case of favoring functionality over security, these design decisions and their consequences should be included in the specification. This ensures that all invested parties are informed of potential security risks and can take appropriate measures.

Even though we rigorously and systematically performed our security analysis, guided by the two security goals stated before, it remains an informal and manual analysis approach. Full completeness can only be achieved by formal analysis techniques. Major disadvantages of formal analysis are the tremendous amount of effort involved, as well as scalability to complex models. Akhawe et al. [3] have already started this work with an Alloy model of web interactions, including selected parts of HTML5 and the CORS specification, but are already pushing the limits of model finding tools. Continuing this work, possibly using other formalisms than Alloy, has a great potential to further enhance or validate the security of emerging web technologies and specifications.

7.2 User-based Security

A clear trend in the analyzed security-sensitive JavaScript APIs is the shift towards a user-consent security model. The majority of security decisions is pushed towards the end-user, under the assumption that an average internet user is capable of making correct decisions and understanding the consequences of his decision. However, this paradigm has often failed before (e.g. [10]), and without a clear agenda across the specifications, we expect this to fail here as well. For instance, once the now emerging specifications become mainstream and are used by the majority of applications, users will be bombarded with permission requests. For instance, Firefox currently allows a user to explicitly configure 6 permissions per origin, which does not even include explicitly-revocable permissions, such as the installation of a content or protocol handler.

With the average user in mind, we think it is important to design a coherent and consistent permission model for access to sensitive features. Whenever a standard wants or needs to protect such sensitive features, it should refer to this exact permission model, without having to specify its own. Extracting the permission model from individual standards allows independent evolution, needed for less user-dependent and more preconfigured permission models.

7.3 Private Browsing

In extension to example #11, i.e. the wide-spread under-specification of behavior in sandboxed environments, we also identified similar under-specification for private browsing contexts. For example in Chrome, stored permissions for Geolocation leak from private to normal browsing mode.

All major browsers offer such a privacy-friendly mode, but none of these are actually standardized nor uniform. As a direct side-effect, none of the standards take them into account. The lack of definition of the private browsing context is not new. Aggarwal et al. previously studied the security and privacy of private browsing modes, against both local and remote attackers [2]. They reveal violations that completely defeat the benefits of private browsing mode.

Based on their results and our conclusions, we argue the need to harmonize the security requirements of private browsing modes within the standardization bodies. This would enable graceful interaction of the private browsing context with newly-added features from within the specifications.

8 Conclusion

In this paper, we aimed to thoroughly scrutinize emerging web standards for potential security problems. We performed a systematic and repeatable analysis using two generally applicable security goals: *preservation of security invariants* and *graceful interaction of coexisting features*. From the security analysis, we can conclude that the overall security of the standards is quite good. Two important categories of problems have been observed: (1) 12 violations of the security goals, thus leading to an increase in attack surface, due to application-specific uses or specific interaction scenarios between features, and (2) 31 issues with vague or ambiguously specified features, leading to inconsistent and suboptimal implementations. Additionally, we identified that for 6 out of 11 explicit security considerations in the specifications, major browsers are non-compliant and thus leave secure specifications vulnerable in the end.

Acknowledgements

The results presented in this paper build on experience from an earlier security analysis performed with the support of ENISA [9]. This research is partially funded by IBBT, IWT, the Research Fund K.U. Leuven and the EU-funded FP7-projects WebSand and NESSoS.

References

[1] The Cross-site Scripting FAQ. http://www.cgisecurity.com/xss-faq.html.

[2] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proc. of 19th Usenix Security Symposium*, 2010.

[3] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. *Computer Security Foundations Symposium, IEEE*, 0:290–304, 2010.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *In Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.

[5] S. Bradner. Key words for use in rfcs to indicate requirement levels. 1997.

[6] T. Close and M. Miller. Uniform messaging policy, level one. http://www.w3.org/TR/UMP/, January 2010.

[7] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *15th Nordic Conference in Secure IT Systems (NordSec 2010)*, 2011.

[8] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. *Computer Security–ESORICS 2011*, pages 100–116, 2011.

[9] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A security analysis of next generation web standards. Technical report, European Network and Information Security Agency (ENISA), July 2011.

[10] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[11] N. Doty, D. K. Mulligan, and E. Wilde. Privacy issues of the w3c geolocation api. 2010.

[12] M. Heiderich. Html5 security cheatsheet. http://code.google.com/p/html5security/, December 2011.

[13] I. Hickson. Web sql database. http://www.w3.org/TR/webdatabase/, November 2010.

[14] I. Hickson. Html5. http://www.w3.org/TR/html5/, May 2011.

[15] I. Hickson. Html5 web messaging. http://www.w3.org/TR/webmessaging/, October 2011.

[16] I. Hickson. Web storage. http://www.w3.org/TR/webstorage/, October 2011.

[17] E. Law. Combating clickjacking with x-frame-options. http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx, March 2010.

[18] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *15th Nordic Conference on Secure IT Systems*, 2010.

[19] J. Mayer and A. Narayanan. Do not track - universal web tracking opt out. http://donottrack.us/, 2011.

[20] N. Mehta, J. Sicking, E. Graff, A. Popescu, and J. Orlow. Indexed database api. http://www.w3.org/TR/IndexedDB/, December 2011.

[21] L. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 481–496, 2010.

[22] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf, January 2008.

[23] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, 2009.

[24] A. Popescu. Geolocation api specification. http://www.w3.org/TR/geolocation-API/, September 2010.

[25] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[26] B. Sterne and A. Barth. Content security policy. http://www.w3.org/TR/CSP/, November 2011.

[27] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[28] M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium*, 2010.

[29] D. Tran and M. Froumentin. The system information api. http://www.w3.org/TR/system-info-api/, February 2010.

[30] D. Tran, I. Oksanen, and I. Kliche. The media capture api. http://www.w3.org/TR/media-capture-api/, September 2010.

[31] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Webjail: Least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 307–316. ACM, 2011.

[32] A. Van Kesteren. Cross-origin resource sharing. http://www.w3.org/TR/cors/, July 2010.

[33] A. Van Kesteren. Xmlhttprequest. http://www.w3.org/TR/XMLHttpRequest/, August 2010.

[34] A. Van Kesteren. Xmlhttprequest level 2. http://www.w3.org/TR/XMLHttpRequest2/, August 2011.

[35] W3C. Testing. http://www.w3.org/html/wg/wiki/Testing, November 2011.

[36] M. Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Main, 2011.

[37] M. Zalewski. Postcards from the post-xss world. http://lcamtuf.coredump.cx/postxss/, 2011.

[38] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.