A Security Analysis of Emerging Web Standards
HTML5 and Friends, from Specification to Implementation

Philippe De Ryck, Lieven Desmet, Frank Piessens and Wouter Joosen
IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium

Keywords: HTML5, Web Application Security, Standards, Specification.

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. In total, we found 45 issues, of which 12 are violations of the security goals and 31 issues concern under-specified features. 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. All details can be found in an extended version of this paper (De Ryck et al., 2012).

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, such as 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 extension of the client-side platform started with XMLHttpRequest, an explicit client/server communication mechanism and continued with the recently proposed Web Messaging API, which enables communication between browsing contexts. Currently, the client-side platform is being developed at a rapid pace, with features like media elements, an iframe-based sandbox, several client-side APIs (storage, system properties, location, etc.), and many more.

Unfortunately, an extensive set of client-side features poses an attractive target for attackers, making the user into a potential victim. For example, insecurities in an API for capturing media in a web application can lead to stolen captures. 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.

A key aspect of this security analysis is the identification of potential security violations. 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 aims to appropriately *guard 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. Instead, we identify two generally applicable security goals: *preservation of security invariants* and *graceful interaction of co-existing features*.

From the security analysis of ten secure-by-design W3C web 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, 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 viola-

De Ryck P., Desmet L., Piessens F. and Joosen W..
A Security Analysis of Emerging Web Standards - HTML5 and Friends, from Specification to Implementation.
DOI: 10.5220/0004049502570262
In *Proceedings of the International Conference on Security and Cryptography* (SECRYPT-2012), pages 257-262
ISBN: 978-989-8565-24-2

tions of one of the stated security goals under certain application-specific circumstances.

In the remainder of this paper, we present the approach and the process of the security analysis, followed by a selection of three security goal violations (Section 2). We continue to discuss both related work on web and standardization security (Section 4) and future improvements (Section 5).

2 OVERVIEW OF THE SECURITY ANALYSIS

The broad set of new and emerging web standards covers an wide range of functionality and features, from detailed HTML parsing to rich JavaScript APIs and security design guidelines. In the analysis, we focus on the newly-added core features of HTML5 (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. Concretely, we have studied the following list of ten W3C specifications: HTML5, Media Capture API, Web Messaging, XMLHttpRequest Level 1 and Level 2, Cross-Origin Resource Sharing, Uniform Messaging Policy, Web Storage, Geolocation API and System Information API.

The core process of the security analysis is the investigation of each specification in isolation and in coexistence with other specifications, based on the distilled information (fully explained in (De Ryck et al., 2012)). The goal is to detect and identify weaknesses in the specifications leading to an increase of attack surface, due to a violation of one of the security goals: *preservation of security invariants* and *graceful interaction of co-existing features*.

The first security goal states that new mechanisms introduced in the browser 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. Introducing new cross-origin communication mechanisms should not break the security of these existing applications. Akhawe et al. (Akhawe et al., 2010) 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 new kinds of origins (such as the *unique* origin in sandboxes).

Scope and Attacker Model. The ten studied W3C specifications form a representative subset of the emerging web standards, and are per category depicted in an abstract model of modern web standards. 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, requesting information from the device, communicating with external services, storing data on the client-side and caching an entire application or component. Note that for “Client-side storage” we focus on the Web Storage specification, which contains most of the complexity with regard to storage areas and their access rules, compared to other client-side storage specifications.

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

We consider both the *web attacker* and the *gadget attacker* to be valid attacker models (Barth et al., 2008). The former is a malicious principal who owns one or more machines on the network where the user gets and renders content from, whereas the latter has one additional ability: the integrator embeds a gadget of the attacker’s choice. Concretely, we assume that a user visits honest sites, honest sites unknowingly incorporating malicious content (gadget attacker) or malicious sites (web attacker), using an honest, specification-compliant browser.

3 RESULTS OF THE SECURITY ANALYSIS

Performing the security analysis as described before has revealed 45 security issues, of which the majority, 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*. These violations are weaknesses in the specifications that, depending on the specific application, potentially increase the attack surface. Additionally, we identify several cases where browsers fail to comply with specific security considerations stated in the specifications.

In the remainder of this section, we discuss three examples of an increased attack surface. The full report (De Ryck et al., 2012) contains more examples, as well as issues with under-specification and non-compliance with explicit security considerations.

[Example #1] HTML5: Injection of Submission Controls. The HTML5 specification enables the use of form controls outside of a `form` element, to overcome the lack 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 (Zeller and Felten, 2008). 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 (Sterne and Barth, 2011), can still be vulnerable. Additionally, traditional form stealing using HTML injection (Zalewski, 2011) 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 their sites 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.

[Example #2] CORS: Arbitrary Body Format. The Cross-Origin Resource Sharing and XMLHttpRequest Level 2 specifications 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 browser decides whether the script can access the response contents. Actual requests are new types of requests, such as 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 &, or the multipart/form-data format, which contains multiple sections separated by a self-defined boundary. The analysis uncovered that a simple cross-origin POST request sent from JavaScript is still restricted to these content types in its header, but compliance of the body format to the specified content type is never checked. This means that an attacker is able to send cross-origin POST requests with an arbitrary body format, but will most likely be denied access the response, since the appropriate response headers to allow access will be missing.

A concrete attack scenario is an application offering an authenticated API, e.g. using the JSON format. 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.

Traditionally, the body format of cross-origin POST requests was tightly constrained, but simple CORS requests, explicitly aimed at not enabling more features than existing with HTML elements, relax these constraints. CORS opens up existing applications to cross-origin requests with an arbitrary body format, previously only available within the same origin. This is a violation of the first security goal.

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

[Example #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, 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*, meaning that the value *null* will be used in headers or origin checks.

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 for the 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 API provider to open up its API to all origins. The interaction between the sandbox attribute and CORS 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.

4 RELATED WORK

In this section, we briefly discuss relevant subsets of the vast amount of related work on web application security. First, we highlight traditional approaches, focused on patching basic flaws in the classical web paradigm (either in the code itself, or via client- or server-side hardening). Next, we 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 problems in web security 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 (Su and Wassermann, 2006; Zeller and Felten, 2008), but are unfortunately not always consistently deployed. This has in turn led to policy-driven security on the client-side, such as Content Security Policy (Sterne and Barth, 2011) or X-Frame-Options (Law, 2010).

Complementary, several techniques offer controlled execution of third-party JavaScript (as is typically the case in mashup compositions (De Ryck et al., 2011a) and online advertisements (Ter Louw et al., 2010)), which is especially relevant because of the obviously increased set of features available to JavaScript developers. Common approaches towards fine-grained behavior control are the use of security wrappers in JavaScript (Phung et al., 2009; Magazinius et al., 2010), policy-controlled client-side sandboxes (Ter Louw et al., 2010; Miller et al., 2008) and the use enhanced browser support (Meyerovich and Livshits, 2010; Van Acker et al., 2011).

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 (Barth et al., 2008), 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 (Akhawe et al., 2010), specifically focus on web security mechanisms. The formal model is also useful to evaluate the security of newly proposed countermeasures (De Ryck et al., 2011b).

Other related research is by Rydstedt et al. on the effectiveness of clickjacking defenses (Rydstedt et al., 2010) and of Aggarwal et al. about the actual security of private browsing modes (Aggarwal et al., 2010). In this area, we also consider the work of Doty et al. about privacy issues in the Geolocation API (Doty et al., 2010) and the work of Heiderich et al. on cross-site scripting attack vectors in HTML5 (Heiderich, 2011) to be highly relevant.

5 DISCUSSION

The security analysis shows that the studied web specifications already achieve a high level of security, but 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. This work has already started (Akhawe et al., 2010) with an Alloy model of web interactions, which is 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.

The full report of the security analysis (De Ryck et al., 2012) does not only cover potential increases of the attack surface, but also discusses how specifications suffer from under-specification and ambiguity, leading to inconsistent and potentially insecure implementations as a consequence. Additionally, the full report illustrates that mainstream implementations not always comply with explicit security considerations stated in the specifications. As a consequence, securely specified features might in practice still be vulnerable due to this mismatch.

6 CONCLUSIONS

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 co-existing features*. From the security analysis, we can conclude that the overall security of the standards is quite good. Nonetheless did we identify several violations of one of the stated security goals under certain application-specific circumstances.

ACKNOWLEDGEMENTS

The results presented in this paper build on experience from an earlier security analysis performed with the support of ENISA (De Ryck et al., 2011c). This research is partially funded by IBBT, IWT, the Research Fund K.U. Leuven and the EU-funded FP7-projects WebSand and NESSoS.

REFERENCES

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

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

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

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

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

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

De Ryck, P., Desmet, L., Piessens, F., and Joosen, W. (2012). A security analysis of emerging web standards - extended version. Technical Report CW 622, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

Doty, N., Mulligan, D. K., and Wilde, E. (2010). Privacy issues of the w3c geolocation api.

Heiderich, M. (2011). Html5 security cheatsheet. http://code.google.com/p/html5security/.

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

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

Meyerovich, L. and Livshits, B. (2010). 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.

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

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

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

Sterne, B. and Barth, A. (2011). Content security policy. http://www.w3.org/TR/CSP/.

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

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

Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., and Joosen, W. (2011). 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.

Zalewski, M. (2011). Postcards from the post-xss world. http://lcamtuf.coredump.cx/postxss/.

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