Lightweight Integrity Protection for Web Storage-driven Content Caching

Sebastian Lekies and Martin Johns
SAP Research Karlsruhe
{firstname.lastname}@sap.com

***Abstract*—The term *Web storage* summarizes a set of browser-based technologies that allow application-level persistent storage of key/values pairs on the client-side. These capabilities are frequently used for caching of markup or script code fragments, e.g., in scenarios with specific bandwidth or responsiveness requirements. Unfortunately, this practice is inherently insecure, as it may allow attackers to inject malicious JavaScript payloads into the browser's Web storage. Such payloads reside in the victim's browser for a potentially prolonged period and lead to resident compromise of the application's client-side code.**

In this paper, we first present three possible attack scenarios that showcase how an attacker is able to inject code into web storage. Then we verify that Web storage is indeed used in the outlined, insecure fashion, via a large-scale study of the top 500.000 Alexa domains. Furthermore, we propose a lightweight integrity protecting mechanism that allows developers to store markup and code fragments in Web storage without risking a potential compromise. Our protection approach can be introduced without requiring browser modifications and introduces only negligible performance overhead.

I. Introduction

Since the rise of Web 2.0 applications a shift from server-side to client-side functionality is perceivable on the Web. Especially new HTML5 features such as Web Messaging, Cross-Origin Resource Sharing or Offline Apps enrich the user-experience of modern Web applications. However, with the power of these new APIs comes the responsibility to utilize these features in a secure fashion. In the past some research work has already been conducted to reveal potential security issues with client-side technologies [1, 3, 4, 6, 12].

In this paper we investigate HTML5's Web Storage API that consists of the SessionStorage and LocalStorage attributes [5]. Web Storage is a mechanism that allows a Web application to store structured data within the user's Web browser via Javascript. While this API can be used for client-side state management, it is also often used for caching[17] (See Section IV for more details). Especially, in mobile environments where bandwidth and latency matters Web-Storage-based caching can be a powerful technique to decrease loading times by saving and reusing frequently required scripts or style declarations on the mobile device[17].

However, caching such content in a storage that is accessible via scripting is a dangerous practice as it creates new attack vectors for adversaries. The cause of the problem is the fact that at one point in time, code written to the storage has to be executed again. Hence, if an attacker is able to exchange the cached code with his payload, the application automatically runs the malicious content (See Section III for examples). Well-known cross-site scripting defense techniques such as input validation or output encoding are not applicable in this scenario, as legitimate code fragments would also be rendered void by these XSS filters.

In this paper we first investigate the usage of Web Storage with regards to code caching by investigating the front pages of the Alexa top 500.000 Web sites. Thereby, we found out that 20,422 Web sites make use of client-side storage and that 386 Web sites store 2084 pieces of HTML, Javascript code or CSS style declarations within Local- or SessionStorage. Furthermore, we present a method that allows a Web application to securely store code fragments on the client-side. We achieve this by utilizing checksums that are calculated for cached code. Whenever the code is fetched and executed from Web Storage the application validates the checksum in order to ensure integrity of the stored content. Therefore, an attacker is not able to inject his payload into client-side storage capabilities and thus attacks are rendered void.

The rest of the paper is structured as follows. After we outlined the basics of Web Storage in Section II, we present three attack scenarios in Section III that could be utilized by an attacker to smuggle his payload into the client-side Web storage of a victim. After that, we investigate the usage of Web Storage by presenting the results of a large-scale study of the Alexa top 500,000 Web sites in Section IV. As pointed out by the study results Web Storage is used in an insecure fashion when it is utilized for client-side code caching. Therefore, we developed a JavaScript library that protects Web applications from being exploited while still preserving the benefits of code caching. The basic idea, an evaluation and possible limitations of our approach are discussed in Section V. Finally, we present related work in Section VI and a conclusion in Section VII.

II. Technical Background

In this Section we briefly outline the technical backgrounds. After covering the basics of Web Storage, we present different use cases for the presented capabilities.

A. What is Web Storage?

Web Storage is a mechanism that allows a piece of Javascript to store structured data within the user's browser [5]. Web Storage is, thereby, an umbrella term for two related functionalities - SessionStorage and LocalStorage. Each of these storage types implements the same API and adheres to the

same security restrictions. The underlying storage mechanism is implemented via a key-value scheme that allows to store, retrieve and delete a String value based on a certain key (See Listing 1 for an example).

Listing 1 Exemplary usage of LocalStorage

```
<script>
  //Set Item
  localStorage.setItem("foo","bar");
  ...
  //Get Item
  var testVar = localStorage.getItem("foo");
  ...
  //Remove Item
  localStorage.removeItem("foo");
  ...
  //Clear all
  localStorage.clear();
</script>
```

In general, access to data stored within Web Storage is limited to same origin resources only. Each site gets one storage area assigned to it, so that data of different origins is strictly separated. Therefore, data stored by a Web site on *a.net* is only accessible to other resources from *a.net*, but not from *b.net*.

1) SessionStorage: SessionStorage was designed for transaction-based scenarios in which a user is able to simultaneously carry out the same transaction in multiple browser windows. Within the same window, data can be stored and retrieved from the storage by any Web page loaded from the same origin. A page loaded within another window posses its own storage and hence is not able to access the data from another window.

2) LocalStorage: LocalStorage differs from SessionStorage in respect to scope and lifetime. As opposed to SessionStorage, data within the LocalStorage can also be accessed across different browser windows by same origin Web pages. Furthermore, LocalStorage is persistent across sessions, while data within SessionStorage is discarded whenever the corresponding session is closed. (Note: The lifetime of a session is unrelated to the lifetime of the corresponding user agent process, as the user agent may support resuming sessions after restart[5].)

3) GlobalStorage: Earlier versions of the HTML5 specification also contained the GlobalStorage directive. However, it was removed from the specification in favor of the LocalStorage API [9]. GlobalStorage holds multiple private storage areas that can be accessed over a longer period of time across multiple pages and sessions. Although, GlobalStorage is deprecated it is still implemented and used by some Web applications.

4) IndexedDB: IndexedDB [14] is a further, experimental client-side storage feature which is currently only supported by Firefox and Chrome. Due to the currently incomplete browser support for IndexedDB, we do not specifically address this technology for the remainder of this paper. However, we do not expect the usage patterns and resulting potential security implications to differ from the corresponding results for the established technologies.

B. Use cases for Web storage

Up to now, two usage patterns for Web storage have received some attention: Keeping state in offline situations and using Web storage for caching purposes. We briefly revisit these concepts in this section.

However, the general concept of persistent, client-side Web storage is still a rather recent addition to the Web application paradigm. Hence, not much experience has been documented, how these APIs end up being used by real-life Web applications. Hence, to collect practical insight into this area, we conducted some applied survey work, which will be the subject of Section IV.

1) State-keeping for offline apps: Modern browsers allow Web applications to provide offline capabilities. For this, the application can explicitly specify which of its Web resources should be kept in the browser's application cache [2]. This is done using a dedicated manifest file that lists the URLs of to be stored resources. In situations, in which the Web browser is disconnected form the network, these files, which were stored earlier, are loaded and rendered from the appcache. However, as no network connection is present to propagate the user's action to the Web server, all actions, which may have a permanent effect, have to be temporarily stored in the browser until the browser reenters the online mode. For this purpose, Web storage is well designed.

2) Using Web storage for controlled caching of Web content: The current caching facilities of Web browsers only allow to cache the content of full HTTP responses, i.e., complete documents, scripts, or images. Furthermore, the actual caching process is transparent to the application and not under its control. Hence, in situation, in which the need occurs to either cache only sub-parts of HTML documents or in which the application needs closer control in respect to the cached content and its usage, Web storage provides the needed capabilities [15]. This seems to be especially appreciated in the context of Web applications that target mobile devices, which, unlike their modern desktop counterparts, may have to deal with limited network bandwidth and high network latency [17].

III. Attacks

A. Insecure usage of Web storage

As motivated in Section II-B2, a potential use case for Web storage is application-level content caching. In this context, it has been documented [4, 7], that some applications use Web storage for caching Web page components, such as HTML fragments, CSS styles, or Javascripts, which are meant to be included verbatim into the Web page after retrieval from storage (see Lst.2).

On first view, this behavior is safe: A site's Web storage is isolated from untrusted parties via the same-origin policy: Only JavaScript, which is executed within the application's origin is allowed to access or modify the stored content. However, the problem is, that in situation in which an attacker

Listing 2 Insecure usage of LocalStorage-data

```
<script>
if (cached){
   var fragment = localStorage
                    .getItem("analytics");
   document.write(fragment);
} else {
   [retrieve code over the network]
}
</script>
```

might be able to temporarily circumvent this protection (more on this below), he is able to elevate a temporary breach (e.g., through a reflected XSS vulnerability) into a persistent compromise of the Web application's client-side code: By inserting JavaScript code into one of the site's code fragments which are kept in Web storage, the attacker can ensure, that his malicious payload is executed every time the victim accesses the Web application with his browser, potentially for an unlimited amount time. This in turn, enables attack payloads, which were not possible otherwise, such as intercepting of password entry or continuous user observation [7].

B. Attack scenarios

In total, we were able to identify three distinct attack scenarios, which could allow an adversary to persist his script payload in the victim's browser.

1) Cross-site Scripting (XSS): If an XSS vulnerability exists in the application, an adversary is able to permanently inject his payload into the site's Web storage. It is insignificant, if this XSS problem is 'only' a reflected XSS that occurs within a section of the site without sensitive information or interfaces. Neither is it of importance, if the targeted user maintains an authenticated state with the application at the point of time when the attack happens.

The attack could be triggered much later, for example, when the user actively enters the URL of the poisoned application into the address bar of his browser. At this time security aware users will be much less suspicious, as they did not enter the Web site via a manipulated link. Therefore, victims will probably more likely accept strange dialogs or weird behavior.

2) Untrustworthy networks: The existence of an XSS problem is not a necessary condition for the attack: Also every time a potential victim uses an untrusted network, e.g., at a local coffee shop, this action can open an opportunity for an attacker to poison the targeted browser's Web storage: Within this untrusted network, the attacker utilizes active network-level attack techniques to insert himself as a man-in-the-middle into the victims HTTP communication. Having achieved this, he is able to transparently modify the victim's HTTP communication. If the user directly accesses a site, which uses Web storage in an insecure fashion, the attacker can simply insert the JavaScript to modify the site's storage into one of the site's legitimate HTTP responses.

Even if the user only visits unrelated sites and avoids accessing security sensitive Web applications while using an untrusted network, this attack is still possible: In such a case, the attacker can force the browser to access the vulnerable site through inserting a hidden iframe pointing to the site into any delivered HTML content. In such cases, the attacker's payload will patiently reside in the browser's Web storage until later, potentially much later when the user actively accesses the poisoned application again. In consequence, the actual exploitation occurs at a point in time, in which the man-in-the-middle condition does not exist anymore, and the user operates in a perceived trusted network context.

Please note, that this specific attack vector only affects HTTP traffic, which is served without HTTPS protection.

3) Shared browsers: Finally, the outlined insecurity can also be exploited whenever Web browsers are shared by multiple persons, e.g, in the case of internet cafes or public computers in hotel lobbies. In such situations, an attacker can insert his payload into the application's storage as follows: First he accesses the Web application using the shared browser. He navigates the Web application until a state has been reached in which the application has filled the browser's Web storage with the cached code fragments. Then, he executes JavaScript in the context of the loaded Web application through entering a `javascript:`-URL into the browser's address-bar. This JavaScript is executed under the origin of the Web application that is currently displayed in the browser window, i.e., the targeted application. In consequence, the script has full access to the site's Web storage, allowing the attacker to insert his payload.

From this point on, every further user that accesses the Web application using this particular browser will involuntarily execute the attacker-controlled script. Unlike other 'internet cafe'-attack scenarios, which usually rely on malware running on the computer which executes the affected browser, in this case the attacker does not need elevated privileges or additionally installed software, making it a low effort and hard to detect attack.

C. Analysis

All three discussed attack scenarios have a common characteristic: JavaScript, executed in the victim's browser, manipulates the code fragments, which are held in the LocalStorage. This action in done purely on the client-side, without any involvement of the server-side component of the Web application. Hence, the capabilities of a Web site's developer or operator are severely limited, when it comes to detecting such malicious activities: Both the manipulation as well as the subsequent usage of the code fragments never leave the browser.

IV. Survey

A. Research questions

In order to investigate the usage of Web Storage in the wild, we conducted a large-scale study. Thereby we were mainly interested in the following research questions:

1) Penetration:

(RQ1) How many Web sites utilize Web Storage?

(RQ2) What kinds of storage types are used(Local-, Session- or GlobalStorage)?

(RQ3) Does a relation exist between the popularity of a Web site and the usage of Web Storage?

2) Security:

(RQ4) How many Web sites utilize Web Storage for storing code fragments?

(RQ5) How many Web Sites utilize Web Storage in a Secure/Insecure fashion?

B. Survey methodology

In order to answer the presented research questions we crawled the front-pages of the Alexa top 500,000 Web sites and examined the obtained data with the open source Web testing framework HTMLUnit[1]. To do so, we augmented HTMLUnit's native Storage objects in such a way that any access to Local-, Session-, or GlobalStorage is logged during the analysis.

One particularly difficult problem was the detection of insecure usage. Hereby, we came to the conclusion, that storing any content within Web Storage that leads to uncontrolled code execution is dangerous. Therefore, we established the following classification for stored values:

1) **Problematic:** Problematic data is *very likely executed in an uncontrolled fashion*. Such data consists of Javascript fragments, CSS style declarations or HTML source code.
2) **Suspicious:** Suspicious content could *potentially be executed*. One example for this category is JSON data. JSON can either be parsed in a secure fashion by using secure parsers like *JSON.parse()* or simply by executing it via *eval()*. While the former is quite secure, the latter could be misused by an adversary to execute Javascript code other than JSON.
3) **Unproblematic:** Content that seems to be *unlikely to trigger an uncontrolled code execution*. Examples for this category are numbers, alphanumeric strings or empty values.

In order to categories the observed values we utilized a half-automatic approach. First we only selected the values from our dataset that contained either angle ("<", ">") or curved brackets ("{","}") as code fragments are very dependent on these characters. Everything else was marked unproblematic. Then we manually categorized the remaining values according to our classification.

C. Results

In this section, we present the results of our survey and discuss these results in the context of the identified research questions.

a) Penetration: In the first part of our evaluation we are interested in the general usage of the Web Storage APIs. On the investigated 500,000 Web sites we recorded more than 122,615 calls (by 20,421 Web sites) to Web Storage directives. The biggest part of 82,884 targeted at LocalStorage, while about 39,068 targeted at SessionStorage and still 663 made use of the deprecated GlobalStorage. For more details on the general crawling results, please refer to Table I.

[1]HTMLUnit, version 2.9, http://htmlunit.sourceforge.net/

Name	Total	Web sites	% Sites
Crawled Pages	500,000	500,000	100 %
Total Web Storage Accesses	122,615	20,421	4.08 %
LocalStorage Accesses	82,884	18,811	3.76 %
SessionStorage Accesses	39,068	11,288	2.26 %
GlobalStorage Accesses	663	202	0,04 %
via getItem()	81,811	19,890	3,98 %
via setItem()	35,823	16,169	3,23 %
via removeItem()	4,981	2385	0,48 %

TABLE I: General overview of crawling results

Another fact that we were interested in was the relation between the usage of Web Storage and the popularity of a Web site. Figure 1 shows the relation between Alexa Rank and the usage of Web Storage aggregated to a level of 10,000 sites. The diagram clearly shows that popular Web sites are more likely utilizing Web Storage than not so popular ones.

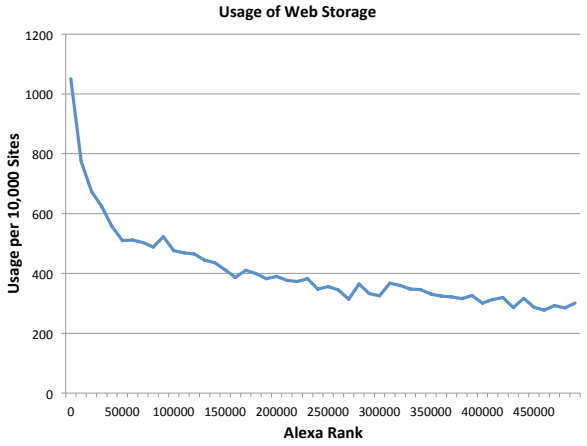

Fig. 1: Usage of Web Storage related to Alexa Rank

b) Security: Besides investigating the pure usage of Web Storage we were also interested in the fact whether Web Storage is used for code caching in practice and hence used in a potentially insecure fashion. Therefore, we applied the methodology described in the previous Section to categorize the values received during our crawl. From the 122,615 values, 10,547 contained angle or curved brackets. 5055 of these potential code fragments were empty JSON Objects ("{}") and hence categorized as suspicious. The remaining 5492 were manually inspected and categorized as follows:

- 2084 values stored by 386 Web sites contained Javascript code, HTML or CSS style declarations.
- 3,340 contained non-empty JSON objects that did not contain any code fragments (except legitimate JSON syntax).

During our investigation we found another interesting attack vector. Within 68 entries we found URLs pointing to CSS or Javascript files. By manually inspecting the corresponding Web sites, we found out that these URLs were used to request additional resources via script tags and CSS style declarations. If an adversary is thus able to manipulate these URLs, he can force a Web site to load an attacker controlled script file instead of the legitimate one. In this way the attacker can inject

code into a Web site via Web Storage despite the fact that the stored content is not evaluated (via eval, etc) directly by the Web site.

Given the numbers above we categorized the values as follows:

1) **Problematic:** 2,084 Storage entries with code + 68 stored URLs pointing to Javascript or CSS files = 2,152
2) **Suspicious;** 3,340 non-empty JSON objects + 5,055 empty JSON objects = 8,305
3) **Unproblematic:** 112,158 apparently safe entries

V. Countermeasure

Based on the results of our survey, it can be observed that indeed Web storage is utilized frequently to locally store code fragments, probably for caching and responsiveness reasons. Furthermore, the number of observed cases suggests that this technique has proven its merit in practice, and hence, will very likely be continued being in use.

In consequence, we designed a lightweight technique that allows Web developers to use this technique in a secure fashion. Our approach centers around verifying the integrity of a value stored within Web Storage before it is injected in the Web pages DOM. This way, a Web application is able to detect whether cached content was manipulated without loosing the benefits of caching.

A. Description of the technique

As opposed to other types of user input, we cannot apply well-known and proven anti-XSS techniques to secure cached Code. Input validation or output encoding would prevent legitimate code from executing and differentiating between legitimate and malicious code is a very hard problem. Therefore, our system relies on integrity checks to validate the legitimacy of stored content. Whenever our system receives a value from Web Storage it calculates a cryptographic checksum and verifies whether the contained code was received from the server before or if it was manipulated/stored by an adversary.

In order to do so, the server calculates and stores a checksum whenever it creates a new piece of code that is supposed to be cached within Web Storage. At each request to a resource that reuses cached code, the server includes the checksum into the response. A client-side script then calculates the checksum of the content received from Web Storage and compares it to the checksum received from the server. Only if the two values match, the code is still legit.

B. Implementation

Our implementation consists of a lightweight JavaScript library that transparently handles caching and integrity validation for the application. As a first step, the library utilizes function wrapping techniques [13] to overwrite the native Storage objects (See listing 3 for an example with LocalStorage). The wrapper implements the same API and hence can be used by the application as if it was the original storage object. Furthermore, the wrapper holds a list of keys that are supposed to be used for code caching and the checksums for the corresponding code fragments.

Listing 3 Wrapping LocalStorage with a custom wrapper

```
<script>
  var wrapper = new StorageWrapper();
  Object.defineProperty(window, "localStorage",
                {value: wrapper});
</script>
```

Whenever the application requests an entry with a key from the list, the wrapper calculates the checksum of the entry and compares it to the checksum received from the server. If the two checksums match the data is passed on to the application. If the checksums do not match our script contacts the server via XMLHttpRequest and requests a new version of the cached code that is then stored within the storage and handed over to the application.

For keys that are not available within the list received from the server, our library automatically applies output encoding before handing the value to the application. As the server did not explicitly express its will to store code within this key-value pair (i.e. the server did not deliver a checksum), this seems an appropriate action. Therewith, caching becomes fully transparent to the application and at the same time we offer a secure Web Storage by default. For cases, were the server explicitly wishes to store code that should not be output encoded, we offer a mechanism to opt-out this feature for certain Storage keys. (See Figure 2 for a more detailed view)

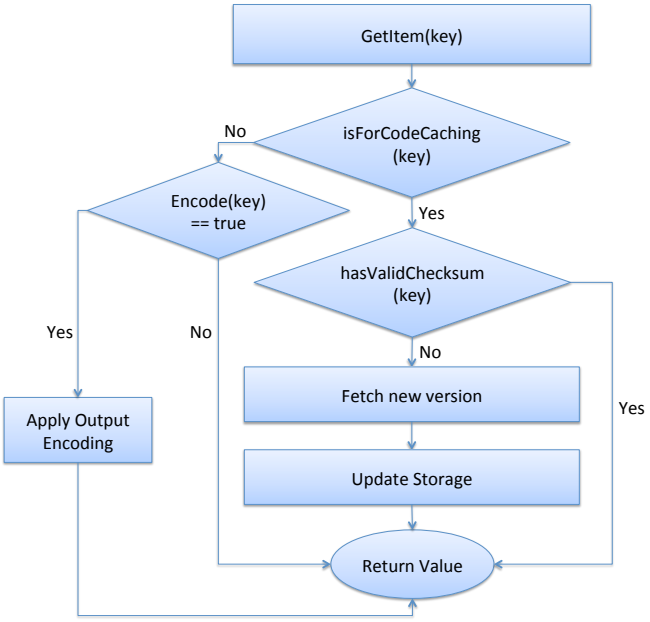

Fig. 2: Flowchart of our countermeasure

C. Security evaluation

In this Section we discuss how our approach defends against the scenarios presented in Section III.

1) **Cross-site Scripting:** In the cross-site scripting scenario, we assume that an attacker is able to misuse a XSS vulnerability, while the user is not authenticated to the Web application. Thereby, he inserts his payload into the LocalStorage. At a later point in time, the user authenticates himself to the application and thus is now able to see private information or to trigger security sensitive actions. Now the code is fetched and executed by the Web application from LocalStorage. With our protection mechanism in place the code manipulation is detected between fetching and execution. Instead of executing the attacker's payload, a new version of the code received from the server is used to proceed.
2) **Shared browsers:** For this scenario we assumed that an attacker has access to the browser, before a victim actually uses it. The attacker injects his payload into the LocalStorage of the Web site to be attacked. Later, when the user enters the same site again, the page loads the cached code, i.e. the attackers payload, and executes it. If the page is equipped with our library, the server sends back the checksum of the cached code when the victim enters the Web site. The LocalStorage wrapper calculates the checksum of the adversary's payload and detects that it is different from the one received by the server. Consequently, a new version of the script is requested from the server and written back to the LocalStorage overwriting the attacker's code.
3) **Untrustworthy networks:** Similar to the Internet cafe scenario, our library is able to detect that the values within LocalStorage were manipulated when the user enters the attacked Web application over a trusted network. Therefore, the library again fetches a new version of the code, effectively rendering the attack void.

As seen above, our Library is able to reliably detect and defend against all the presented scenarios and thus significantly raising the bar for Web Storage-based attacks

D. Functional evaluation

In this Section we seek to evaluate the practical impact of our countermeasure. Code caching is used to reduce response times of Web applications, especially in mobile scenarios where bandwidth and latency matters. As our solution requires checksums to be sent to the client and additional verification steps to be taken during runtime, our solution could possibly affect response times in a negative fashion. Therefore, we conducted some performance measurements to prove the applicability of our solution. First we investigated the additional overhead of sending checksums over the wire and secondly we benchmarked checksum calculation on the client-side.

In order to offer a robust and reliable defense technique checksums have to be calculated using collision free hash algorithms. Therefore we choose *SHA256* for hashing cached code. As the name SHA256 suggests the utilized hashes are 256 bits long. Therefore, code fragments protected by our technique should not be smaller than this size plus the size of our library, otherwise our solution would cause overhead as the hashes and verification methods have to be sent to the client at each request. Our library (including one hash) is 563 bytes long plus additional 1731 bytes for the hashing library. As soon as the Javascript crypto API is generally available within browsers we can get rid of the additional 1731 bytes for the hashing library.

In order to prove the applicability of our approach we examined the 2084 code fragments, which we collected during our survey. In average those code fragments were about 76,000 bytes large. Given the 2,294 bytes for our libraries, the advantage of code caching is still clearly perceivable and as sites often store multiple fragments, the overhead becomes even smaller. For fragments that are smaller than our library we discourage Web Storage-based caching.

Another bottleneck could be the performance of the hashing algorithm. In order to assess the runtimes, we conducted several tests on different browser's for the 2084 values discussed before[2]. The worst performance was observed for Opera, while Firefox, Safari and Chrome offered a much better performance. As seen in table II the performance overhead implied by our solution is almost neglectable.

Browser	Total time in ms	Average
Firefox	55,790	0,026 s
Safari	51,284	0,024 s
Chrome	55,087	0,026 s
Opera	180,372	0,086,s

TABLE II: Browser Performance for 2084 values

During validation we noticed one further advantage of our library. When updating code pieces, servers cannot automatically enforce an update of the cached version within the user's browser. Therefore, the Web application either has to reset code fragments within Web Storage from time to time or it has to wait days or perhaps even weeks until the user triggers a manual deletion (for example by clearing personal browsing data). Our approach offers a very lightweight update functionality for client-side caches. The server simply needs to change the checksums it returns to the client. When the user requests the cached version again, the checksum of the cached code and the checksum received from the server will differ and hence our script will trigger an update.

E. Limitations

Our technique is capable to robustly handle values, which originated on the server-side. For such values, the Web server can calculate the integrity checksums before sending the code to the Web storage. However, dynamic values, which are created on the client-side as part of the user/browser interaction, cannot be integrity protected (e.g., as part of temporary value storage method for offline-mode Web applications): As these values are generated within the browser, the server has no reliable indicators of the values initial integrity. On the server-side it cannot be decided whether these values were generated through legitimate user interaction or were set through attacker controlled JavaScript. Therefore, such values necessarily have to be regarded as potentially untrusted and should be subject

[2]We utilized the Stanford Crypto Library for our tests

to output sanitization before they are included in the site's DOM. In order to offer protection by default, our approach automatically applies output validation to any values for which the server did not provide a checksum.

F. Outlook

As the outcome of Section IV's survey suggest, there is a demand for application controlled caching capabilities. Therefore, in the long run, it should be investigated how to introduce fine-grained, integrity-protecting caching mechanisms to the browsers in the form of native features.

VI. Related work

A. Security and privacy aspect of Web storage

The first public documentation of insecurely using Web storage has been given in [4]. The authors have evaluated eleven selected Web application which use Web storage and found seven of them vulnerable, due using insufficient sanitized data values. In consequence, the paper proposes to solve the problem with mandatory output sanitization of all values which were retrieved from Web storage. While such an approach would successfully mitigate the discussed attacks, it renders the caching of fragments that contain code or markup impossible. In contrast, our technique allows this usage pattern.

Furthermore, [7] explores the potential exploitation scenarios in depth, which result when an attacker was able to successfully persist his payload into the browser's storage facilities.

Finally, it has been shown, that Web storage can be used to create persistent user tracking mechanisms, which undermine the user's privacy. For instance, Web storage is one of the techniques used in *Evercookie* [8], a proof-of-concept implementation of an intentionally hard to purge user-tracking marker that can be set in a user's browser.

B. Security issues with HTML5/JavaScript APIs

In the recent years further new JavaScript capabilities have been added to modern Web browser besides the Web storage APIs. For several of these new capabilities, potential vulnerabilities or insecure usage scenarios have been identified. In the remainder of this section, we list selected cases.

For one, the introduction of Cross-origin Resource Sharing (CORS) [18] added cross-domain capabilities to the XMLHttpRequest (XHR) object. This modified characteristic of the XHR object has invalidated the (previously correct) assumption that data retrieved through this object is implicitly trustworthy, as it was mandatorily retrieved from a Web location that satisfies the same-origin policy. Hence, in situations in which an attacker can control the URL that is used to request content via XHR, he can instruct the application to retrieve the data from his server for which he can allow the cross-domain access. This, in consequence, can lead to XSS issues if the data ends up being included in the site's DOM [11]. In addition, it has been shown, that due to the new capabilities the XHR object can be utilized for CSRF attacks which involve file upload mechanics, an attack that was previously only possible with the help of Flash [10].

Furthermore, [1] and [4] have shown that the postMessage API [16] can be used insecurely, if a JavaScript that accepts postMessage-events does not verifies the origin of the incoming data carefully.

Finally, in [6] unexpected side effects of the initial implementation of the Web Sockets API have been identified, that can occur in situations in which transparent Web proxies are part of the HTTP communication.

VII. Conclusion

In this paper, we proposed a lightweight integrity preserving mechanism that mitigates malicious manipulation of code fragments that were persisted in a browser's Web storage. To substantiate the demand for such a solution, we explored potential attack scenarios and conducted a large-scale survey on the current practice of Web storage usage. Our approach robustly mitigates all identified attack methods, while maintaing all required functional aspects. No browser modifications are needed as our technique can be implemented purely in JavaScript. It works transparently using API wrappers and introduces only negligible performance overhead. Our approach provides excellent protection abilities and, thus, enables developers to avoid the inherently insecure current practice, as long as browsers do not provide native capabilities for integrity preserving Web storage.

VIII. Acknowledgment

This work was in parts supported by the EU Project WebSand (FP7-256964), http://www.websand.eu. The support is gratefully acknowledged.

References

[1] Adam Barth, Collin Jackson, and John C. Mitchel. Securing Frame Communication in Browsers. In *USENIX Security*, page 1730, 2008.

[2] Eric Bidelman. A beginners guide to using the application cache. [online], http://www.html5rocks.com/en/tutorials/appcache/beginner/, last accessed 02/29/2012, June 2010.

[3] Philippe De Ryck, Lieven Desmet, Pieter Philippaerts, and Frank Piessens. A security analysis of next generation web standards. Technical report, European Network and Information Security Agency (ENISA), July 2011.

[4] Steve Hanna, Eui Chul, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The emperor's new apis: On the (in) secure usage of new client-side primitives. In *Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[5] Ian Hickson. Web storage. Available online: http://www.w3.org/TR/webstorage/, December 2011.

[6] Lin-shung Huang, Eric Y Chen, Adam Barth, Eric Rescorla, and Collin Jackson. Talking to yourself for fun and profit. In *Proceedings of W2SP*, 2011.

[7] Artur Janc. 28c3: Rootkits in your Web application. Talk at the 28C3 conference, http://events.ccc.de/congress/2011/Fahrplan/events/4811.en.html, December 2011.

[8] Samy Kamkar. Evercookie. [online], http://samy.pl/evercookie/, last accessed 02/29/2012, October 2010.

[9] Burak Yigit Kaya. Dom storage. Available online: https://developer.mozilla.org/en/DOM/Storage#globalStorage, Oktober 2011.

[10] Krzysztof Kotowicz. Invisible arbitrary CSRF file upload in Flickr.com. [online], http://blog.kotowicz.net/2011/05/invisible-arbitrary-csrf-file-upload-in.html, last accessed 03/01/2012.

[11] Lavakumar Kuppan. Attacking with HTML5. Talk at the Black Hat Abu Dhabi Conference, https://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-slides.pdf, October 2010.

[12] Lavakumar Kuppan. Chrome and safari users open to stealth html5 appcache attack. Available online: http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html, June 2010.

[13] Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self-protecting JavaScript. In Tuomas Aura, editor, *The 15th Nordic Conference in Secure IT Systems*, LNCS. Springer Verlag, October 2010. (Selected papers from AppSec 2010).

[14] Mozilla Developer Network. IndexedDB. [online], https://developer.mozilla.org/en/IndexedDB, last accessed 02/29/2012, Feburary 2012.

[15] Robert Nyman. Saving images and files in localStorage. [online], http://hacks.mozilla.org/2012/02/saving-images-and-files-in-localstorage/, last accessed 02/29/2012, February 2012.

[16] Eric Shepherd. window.postmessage. [online], https://developer.mozilla.org/en/DOM/window.postMessage, last accessed 02/12/12, October 2011.

[17] Steve Souders. App cache & localstorage survey. Available online: http://www.stevesouders.com/blog/2011/09/26/app-cache-localstorage-survey/, September 2011.

[18] Anne van Kesteren (Editor). Cross-Origin Resource Sharing. W3C Working Draft, Version WD-cors-20100727, http://www.w3.org/TR/cors/, July 2010.