Lightweight Integrity Protection for Web Storage-driven Content Caching

Sebastian Lekies and Martin Johns
SAP Research / WebSand Project
martin.johns@owasp.org

07.11.2012
Agenda

Technical Background
- Context
- What is Web Storage
- Use Cases for Web Storage

Attacks
- Insecure Usage
- Attack Scenarios

Survey
- Research Questions
- Results

Countermeasure
- Implementation
- Evaluation
Technical Background

Context

- Classical Web Applications…
- Not able to keep client-side state
- State is kept on the server side

Ø New use cases require client-side Storage
Ø Web Storage API introduced by HTML5
Technical Background
What is Web Storage?

"Web Storage is a mechanism that allows a piece of JavaScript to store structured data within the user’s browser (on the client-side).

- Web Storage consists of three APIs
  - Local Storage
  - Session Storage
  - Global Storage (deprecated)
- Other new client-side storage technologies exist
  - IndexedDB
  - Web SQL Databases
  - File API
- Scope of this Presentation is limited to Local Storage, however our findings apply for all client-side storage technologies
Technical Background
What is Web Storage?

- Access to Web Storage API is restricted by the Same-Origin Policy
- Each origin receives its own, separated storage area
- Origin is defined by

  http://www.example.org:8080/some/webpage.html

  - protocol
  - host
  - port

<script>
//Set Item
localStorage.setItem("foo","bar");
...
//Get Item
var testVar = localStorage.getItem("foo");
...
//Remove Item
localStorage.removeItem("foo");
</script>
Use Cases for Web Storage

- Client-side state-keeping
  - E.g. for HTML5 offline applications
  - Store state within Local Storage and synchronize state when online

- Using Web Storage for controlled caching
  - Current caching mechanism only allow storage of full HTTP responses
    - Transparent to the application and hence “out of control”

- Web Storage is useful when...
  - only sub-parts of HTML documents needs to be cached e.g. scripts
  - close control is needed by the application

- Especially important in mobile environments
Attacks
Insecure Usage

- Observation: Web sites tend to cache content that will be executed later on HTML-Fragments
- JavaScript code
- CSS style declarations

```html
<script>
    var content = localStorage.getItem("code")

    if(content == undefined){
        content = fetchAndCacheContentFromServer("code");
    }

    eval(content);
</script>
```
Attacks
Insecure Usage

- First thought (WebApp Sec Pavlovian reaction): XSS!!!11!
  - If the attacker can control the content of the Web storage, he can execute JavaScript!!!

Second thought: This behavior is safe
- Web storage can only be accessed by same-origin resources
- So you need JS execution to cause JS execution

Third thought: What if an attacker is able to circumvent this protection?
- Persistence: This is a persistent attack scenario
  - Even if the causing vulnerability has been resolved in the meantime
  - Client-side: Attack payload exists purely in the compromised browser
  - Invisible from the server-based point-of-view
  - “WebApp rootkit”
Attacks
Attack scenarios: Cross-Site Scripting

Scenario: Reflected XSS problem somewhere in the site
  • Vulnerability that does not necessarily require an authenticated context / session
  • Attacker can exploit this vulnerability while the user is interacting with an unrelated web site
    • E.g., a hidden iFrame pointing to the vulnerable application
      • During this attack, the malicious payload is persisted in the user’s browser
  • The payload now “waits” to be executed the next time the victim visits the application
    • This effectively promotes a reflected unauthenticated XSS into a stored authenticated XSS
  • Hence, the consequences are much more severe
    • Furthermore, the payload resides a prolonged time in the victim’s browser
  • Invisible for the server
Attacks

Attack scenarios: Untrustworthy Network

http://unrelated-website.com

Untrustworthy Network

Javascript
mybank.org

User

http://mybank.org

Trustworthy Network

http://unrelated-website.com

Untrustworthy Network

Javascript
mybank.org

User
Attacks
Attack scenarios: Shared Browser

Server
http://mybank.org

Client
Browser
http://mybank.org
Welcome to mybank.org

malicious user
victim
Survey
Methodology

Scope
- Crawl of the Alexa Top 500,000 Web sites

Research Questions
1. Penetration:
2. How many Web sites utilize Web Storage?
3. What kinds of storage APIs are used (Local-, Session- or GlobalStorage)?
4. Does a relation exist between the popularity of a Web site and the usage of Web Storage?
5. Security:
6. How many Web sites utilize Web Storage for storing code fragments?
7. How many Web Sites utilize Web Storage in a secure/insecure fashion?
Survey – Results

Penetration

Usage of Web Storage
Survey – Results
Penetration

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Web sites</th>
<th>% Sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawled Pages</td>
<td>500,000</td>
<td>500,000</td>
<td>100 %</td>
</tr>
<tr>
<td>Total Web Storage Accesses</td>
<td>122,615</td>
<td>20,421</td>
<td>4.08 %</td>
</tr>
<tr>
<td>LocalStorage Accesses</td>
<td>82,884</td>
<td>18,811</td>
<td>3.76 %</td>
</tr>
<tr>
<td>SessionStorage Accesses</td>
<td>39,068</td>
<td>11,288</td>
<td>2.26 %</td>
</tr>
<tr>
<td>GlobalStorage Accesses</td>
<td>663</td>
<td>202</td>
<td>0.04 %</td>
</tr>
<tr>
<td>via getItem()</td>
<td>81,811</td>
<td>19,890</td>
<td>3.98 %</td>
</tr>
<tr>
<td>via setItem()</td>
<td>35,823</td>
<td>16,169</td>
<td>3.23 %</td>
</tr>
<tr>
<td>via removeItem()</td>
<td>4,981</td>
<td>2385</td>
<td>0.48 %</td>
</tr>
</tbody>
</table>

**TABLE I: General overview of crawling results**
Survey – Results
Security

Categorization

- **Problematic:** Code that is *very likely executed* by the Web site (e.g. HTML, Javascript, CSS)
- **Suspicious:** Code that could *potentially be executed*. (e.g. JSON data: Secure parsing via JSON.parse or insecure execution via eval)
- **Unproblematic:** Content that is *unlikely being executed*. (e.g. numbers, alphanumeric strings, empty values)

**Methodology**
- Prefiltering: Values not containing “<“,”>”,”{”,”}” were marked unproblematic
- Manual categorization of the remaining items.
Survey – Results

Security

An additional interesting attack vector
68 entries of the “JSON without code” category contained URLs to Javascript or CSS files

Manual inspection revealed that those URLs were used to fetch additional content

• Manipulation of these URLs leads to code execution capabilities
• Hence these entries were also marked as problematic

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Containing brackets</td>
<td>10,547</td>
</tr>
<tr>
<td>Empty JSON (&quot;{}&quot;)</td>
<td>5,055</td>
</tr>
<tr>
<td>JSON without code or markup</td>
<td>3,408</td>
</tr>
<tr>
<td>Code or Markup</td>
<td>2,084</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problematic</td>
<td>3,042 (2,014 + 1,027)</td>
</tr>
<tr>
<td>Suspicious</td>
<td>8,305 (3,340 + 5,055)</td>
</tr>
<tr>
<td>Unproblematic</td>
<td>112,158</td>
</tr>
</tbody>
</table>
Countermeasure

Problem
• Anti-XSS techniques such as output encoding do not work
• This would make cached code unusable
• This would not help against the URL attacks

Alternative
• Verify that values from Web Storage originate from your application and that integrity is guaranteed
Countermeasure Implementation

http://a.net

Server

+ Checksum
code

Client

OutputEncode()

== Checksum

Non-code value

Web Storage
Countermeasure

Implementation:

- JavaScript Library:

  ```html
  <script type="text/javascript" src="/webStorageWrapper.js">
  ```

- Transparent to the applications by utilizing function wrapping techniques:

  ```javascript
  var wrapper = new StorageWrapper();

  Object.defineProperty(window, "localStorage", {value: wrapper});
  ```

  //Get Item

- `var testVar = localStorage.getItem("foo");`
Countermeasure Evaluation

Performance

- Two performance critical steps:
  - Transfer of our library to the client
  - Calculation of checksums on the client-side

**Transfer of our Library**
- Size of the library 563 bytes (packed) + 1731 for SHA256 = 2,294 bytes in total
- Average size of the 2,084 code fragments: ~76,000 bytes
- Hashing library not necessary in future (with the JS Crypto API available)

Calculation of checksums

- For collision free checksums we chose SHA256 as a hashing algorithm
- To evaluate hashing performance we used the 2,084 code values from our survey

<table>
<thead>
<tr>
<th>Browser</th>
<th>Total time in ms</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox</td>
<td>55,790</td>
<td>0,026 s</td>
</tr>
<tr>
<td>Safari</td>
<td>51,284</td>
<td>0,024 s</td>
</tr>
<tr>
<td>Chrome</td>
<td>55,087</td>
<td>0,026 s</td>
</tr>
<tr>
<td>Opera</td>
<td>180,372</td>
<td>0,086 s</td>
</tr>
</tbody>
</table>
Conclusion

Web Storage is a client-side storage mechanism that is used for
- Client-side state keeping
- Content caching

- **Caching code within Web Storage is a dangerous practice**
- Enables second order attacks
- Used in practice
- Usage is likely to increase

Traditional Anti-XSS mechanisms are not applicable for cached code
- Would make cached code unusable

We proposed a lightweight integrity preserving mechanism for Web Storage
- Enables secure usage of Web Storage
- Preserves benefits
- Only implies very small overhead
Thank you

Contact information:

Martin Johns
martin.johns@sap.com
http://martinjohns.com
@datenkeller