# Development of security extensions based on Chrome APIs

Riccardo Focardi and Mauro Tempesta

Dipartimento di Scienze Ambientali, Informatica e Statistica
Università Ca' Foscari, Venezia, Italy
focardi@unive.it, 827400@stud.unive.it

**Abstract**

Client-side attacks against web sessions are a real concern for many applications. Realizing protection mechanisms on the client side, e.g. as browser extensions, has become a popular approach for securing the Web. In this paper we report on our experience in the implementation of SessInt, an extension for Google Chrome that protects users against a variety of client-side attacks, and we discuss some limitations of the browser APIs that negatively impacted on the design process.

## Introduction

In the last years, the Internet has become an integral part of our life. Everyday activities are more and more digitalized and made available on-line, including critical ones such as financial operations or health data management, and need to be adequately protected to prevent a variety of cyber-criminals from stealing confidential data and impersonating legitimate users.

A relevant subset of web security incidents is represented by client-side attacks against web sessions [1], i.e. attacks that corrupt activities or communications performed by honest user clients. For instance, in a cross-site request forgery attack (CSRF), a malicious site may force the browser into creating arbitrary requests to a trusted website. These requests are considered part of the victim's interaction with that site and may have severe effects on the integrity of the ongoing session.

While web application frameworks allow to realize secure websites, developers often misuse them or are reluctant to adopt recommended security practices: enforcing protection at the client side has thus become a popular way for securing the Web.

A convenient approach, often adopted in the literature [2, 3, 4, 5], is to realize the security mechanisms as web browsers extensions: this simplifies both the implementation and the deployment of the solution, as the developer does not need to inspect and modify the source code of the browser, while the user is just required to install the extension in the browser he habitually uses. A drawback of this choice is that the implementation phase may be hindered by some restrictions of the API exposed to extensions developers.

In the following sections, we give an overview on Google Chrome extensions and discuss some limitations we have met when implementing SessInt [5, 6], a security-oriented extension aimed at protecting the integrity of web sessions against a number of existing client-side attacks, including CSRF, session fixation, reflected cross-site scripting (XSS) and password theft.

## Overview on Chrome extensions

An extension is basically a bundle of files (e.g. JavaScript, HTML, CSS) that adds functionality to the Google Chrome browser [7]. From a functional perspective, the components of an extension can be divided into content scripts and background pages (cf. Figure 1).

Content scripts are pieces of JavaScript code executed in the context of a page that has been loaded into the browser. They have read and write access to the DOM of the page where they

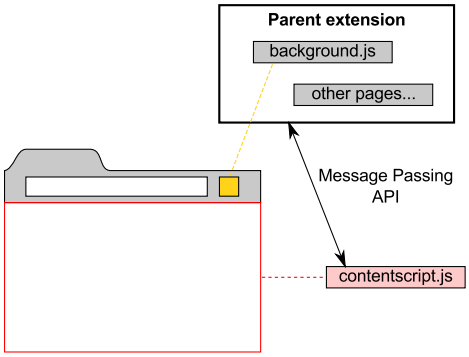


Figure 1: Overview of the architecture of a typical Google Chrome extension.

are injected, but they can neither manipulate any JavaScript variables or functions created by the page nor invoke most of the APIs offered by Chrome.

The background page, instead, is a page running in a separate context without access to the DOM of the pages that are opened in the browser. The background page typically holds the main logic of the extension. Depending on the capabilities granted at install time, background pages can rely on a number of APIs to access the cookie jar of the browser, create or modify the existing tabs and analyze, modify or block network traffic on the fly [8]. Content scripts and the background page can communicate via the message passing API [9].

Most of the methods offered by the Chrome APIs are asynchronous, i.e. they return immediately without waiting for the requested routine to complete. The developer can specify a callback function that is invoked when the outcome of the operation is available. Listing 1 shows how to use the asynchronous method `getAll` in order to retrieve all the cookies satisfying a specified criterion. This method takes, as first parameter, the information to filter the cookies being retrieved and, as second parameter, a callback function which is fed with all the existing, unexpired cookies that match the given filter. In this specific example we filter cookies and log only the ones with `httpOnly` flag set.

Listing 1: prints on the browser console all the `httpOnly` cookies set for `www.google.com`.

```
chrome.cookies.getAll(
  { url: "http://www.google.com/" },
  function (cookies) {
    var http_only = cookies.filter(function (c) { return c.httpOnly; });
    console.log(http_only);
  }
);
```

Some methods and events are instead synchronous. For example, the `onBeforeSendHeaders` event handler allows to modify the HTTP headers of an outgoing request: it is enough to let the method return an object containing an array of headers with the desired values, as shown in Listing 2. The first parameter is a function that processes the requests by stripping the `Referer` header. The second parameter declares that the listener is invoked on all the HTTP(S) requests. The third parameter specifies that the listener is synchronous (`blocking`) and requires the presence of HTTP headers in the request object provided to the listener (`requestHeaders`).

Listing 2: strips the `Referer` header from outgoing HTTP(S) requests.

```
chrome.webRequest.onBeforeSendHeaders.addListener(
  function (req) {
    for (var i = 0; i < req.requestHeaders.length; i++)
      if (req.requestHeaders[i].name === "Referer") {
        req.requestHeaders.splice(i, 1);
        break;
      }
    return {requestHeaders: req.requestHeaders};
  },
  { urls: ["http://*/*", "https://*/*"] },
  ["blocking", "requestHeaders"]
);
```

## Limits of Chrome APIs

The APIs offered by Google Chrome, although powerful enough for general purpose extensions, lack some functionalities or exhibit characteristics that hamper the development of security-oriented extensions. The main limitations we have found during the development of SessInt are either due to the asynchronous behavior of methods and events offered by the APIs or to an insufficient level of detail of the information it exposes.

**Example 1: issues with asynchronous behavior.** In a session fixation, the attacker sets an authentication cookie to a value that he knows so to hijack next sessions. The attack works if the server does not refresh the cookie during a login. In order to prevent session fixation attacks performed by exploiting a script injection vulnerability, SessInt attaches authentication cookies to HTTP(S) requests only if the `HttpOnly` attribute is set. To implement this behavior we need to rely on the `onBeforeSendHeaders` event handler, since we need to modify the `Cookie` header of outgoing requests. However, we cannot use `getAll` to retrieve the cookies, because the execution may reach the `return` statement of the event handler before the callback function supplied to the method `getAll` is executed.

The only way to solve this issue is to keep in the extension a copy of the browser's cookie jar and take care of the operations of insertion or removal of cookies. A reasonable way to proceed seems to be the following: we load all the cookies when the extension is started and then we rely on the `onChanged` method, which is triggered when the cookie jar is modified, to keep the copy up to date. However, we can still have problems due to the fact that events can be raised in an unexpected order, e.g. events related to updates of cookies can be raised after the event of a new network request triggered by the response including those cookies. This is critical when the user successfully authenticates on a website and is redirected to a new page, since the request must attach the authentication cookies that have just been set to maintain the session.

The solution we have been "forced" to adopt in SessInt is to inspect incoming responses for cookies that are set via HTTP headers and to rely on the `onChanged` method for cookies set via JavaScript, which can be safely handled in an asynchronous way. This requires a significant effort to implement our security mechanism for fixation attacks that, at first sight, seems to be quite easy to achieve.

**Example 2: insufficient granularity of APIs.** To prevent CSRF attacks, SessInt automatically strips authentication cookies when cross-origin operations are performed. However we may be interested in enforcing different policies depending on the event that has triggered the request: for instance, we may decide to attach cookies if the request is caused by an interaction of the user with the page, but not if it is caused by JavaScript. Unfortunately, it is not possible to realize this behavior in a safe way: in fact, one may use a content script to register event listeners for clicks on all links inside a page, however the APIs do not provide a way to distinguish between a click of the user and one generated by a malicious piece of JavaScript code that invokes the `click` method on the node in the DOM associated to a link.

Other examples of limitations include the fact that the APIs do not provide any means to access the code of the page before it is rendered, neither a way to execute a content script before inline scripts inside the page are evaluated. Moreover, detecting the interaction with the address bar is a bit quirky, as the user is forced to insert a particular keyword before the extension is allowed to capture his input.

Our opinion is that current Chrome APIs are not very well suited to develop security extensions as they basically give a rather poor control of *what* content can be processed, *when* it is possible to process it and *how* processing can be performed, due to the asynchronous implementation of APIs. We believe that the APIs could be improved with limited effort on these aspects to ease the development of security extension.

# References


[1] OWASP. Top 10 Security Threats. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013.

[2] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and Precise Client-Side Protection against CSRF Attacks. In *European Symposium on Research in Computer Security (ESORICS)*, pages 100–116, 2011.

[3] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Serene: Self-Reliant Client-Side Protection against Session Fixation. In *Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, pages 59–72, 2012.

[4] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Automatic and Robust Client-Side Protection for Cookie-Based Sessions. In *Engineering Secure Software and Systems - 6th International Symposium, ESSoS 2014, Munich, Germany, February 26–28, 2014, Proceedings*, pages 161–178, 2014.

[5] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably Sound Browser-Based Enforcement of Web Session Integrity. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 366–380, 2014.

[6] Mauro Tempesta. Enforcing Session Integrity in the World "Wild" Web. Master's thesis, Università Ca' Foscari, Venezia, 2015.

[7] Google. Overview of Google Chrome Extensions. `https://developer.chrome.com/extensions/overview`.

[8] Google. Google Chrome Extensions APIs. `https://developer.chrome.com/extensions/api_index`.

[9] Google. Message passing. `https://developer.chrome.com/extensions/messaging`.