A Runtime Monitoring System to Secure Browser Extensions

Raúl Pardo¹, Pablo Picazo-Sanchez¹, Gerardo Schneider², and Juan Tapiador²

¹ Dept. of Computer Science and Engineering, Chalmers — University of Gothenburg, Sweden. 
pardo@chalmers.se, pablop@chalmers.se, gersch@chalmers.se
² Dept. of Computer Science, Carlos III University of Madrid
28911 Leganes, Madrid, Spain. jestevez@inf.uc3m.es

1 Problem

Web browsers are applications originally created to surf over the Internet in a friendly way. Nowadays these browsers have turned into a richer software where, apart from surfing the web, users are provided with a vast variety of small applications, called browsers extensions, that are not maintained by the web browsers. Those browser extensions are usually developed by external developers and directly interact either with the web content or with the users.

Browser extensions considerably increase the functionality of the browser. For instance, using a well known translate browser extension, users can have any web page translated to their preferred language; telephone numbers can be remarked in the web page so that users can click on them and automatically open a desktop application to make that call or browser extensions that block the advertisement that some sites insert in the HTML.

However, the inclusion of these third parties applications in the web browsers poses new security and privacy challenges in terms of malware [4, 5, 3], advertisement [8, 9, 2, 1] or disclosing personal information about the user [6], e.g., getting access to browser history or reading another site’s password.

Nevertheless, despite working properly, an extension can have undesirable consequences. Imagine that Alice, who is not a Swedish speaker, visits her bank account website to check her current balance. She gets a HTML content with the latest transactions written in Swedish. Alice decides to use an extension to translate from Swedish to English and confirm that all transactions are correct. To perform this task the extension sends the whole HTML to an external server so that it is automatically translated. Finally, the server sends back the website completely in English. One might argue that this is an undesirable behaviour, since sensitive information—which was not relevant for the task that Alice wanted to perform—was disclosed. In this abstract, we describe preliminary ideas on how to effectively prevent leaks of this kind.

According to the official browser extension developer guidelines³, a browser extension is an application composed of a manifest file, one or more HTML file(s) and zero

³ https://developer.chrome.com/extensions
or more JavaScript. A manifest file is where the information about the extension, such as the capabilities that the extension might use (Content Security Policy (CSP)), is. The HTML files define the User Interface (UI) and it is the link between the extension and the user. Finally, JavaScript files contain the logic and how the extension behaves.

Conforming to the interaction of the extensions with the browser, they can be classified in two different groups: persistent or event. Both types are based on HTML and Javascript files whose content can be accessible indistinctly. However, persistent extensions are always running, whereas event extensions are opened and closes as they are needed.

Content script is an additional functionality that browser extensions can have. A content script is a Javascript file that can interact with the web content and alter it Interactive extensions. It can be used together with both persistent and event extensions. Communication between the content script and the extension can only be done by using specific calls to the API.

Figure 1: Browser Extension Engine

Browser extensions are typically executed in a sequential manner. Figure 1 shows the execution order together with their inputs and outputs. Note that, in the figure, Ext$_1$ takes the original HTML file, performs some actions, and passes the resulting HTML$_1$ to Ext$_2$. The actions that extensions are allowed to perform can be controlled by means of CSPs. Roughly speaking, CSPs act as a filter to allow extensions to load and execute external resources.

2 Our Approach

Consider again Alice’s example in the introduction. In order to avoid having her current balance sent to the translation server she can activate a policy that says: “Extensions cannot send web pages to the internet”. This policy would make the Translate extension unusable, since it relies on external servers to perform translations. Moreover, Alice
does not want to forbid the extension to send any kind of information. She only wants to prevent the extension to send the balance in her bank account. In particular, a better policy would be: “When I visit my bank website do not send any numerical digits to the internet”. Note that the previous policy includes a condition that depends on Alice’s visiting her bank website.

In [7] Pardo et al. introduced policy automata, a formalism to describe policies which depend on the state of the system and events, also known as evolving policies. It was applied in the context of social networks. Nevertheless, in this abstract we propose a similar approach to tackle the problems in browser extensions we have described.

Policy automata consists of a set of states, which indicate the policy \( P \) that must be activated in the system, and transitions between states. Each transition is labelled with an event, a boolean condition and an action, which we denote \( e/c/a \). An event \( e \) can be any event that is detectable in the browser. For instance, a user’s visiting a website or an extension sending information to an external server. The condition \( c \) is any condition regarding the state of the browser. For example, checking whether the tab is in incognito mode or certain extension is enabled. Finally, the action \( a \) can be an arbitrary program, which would be executed as a consequence of triggering the transition.

Example 1. Consider the policy we mentioned earlier: “When I visit my bank website do not send any numerical digits to the Internet”. It can be modelled using policy automata as follows:

When a user opens the browser, the automaton is in the initial state \( s_0 \). The elements inside the states represent policies that must be enforced. In \( s_0 \) the policy \( P_{HTML} \) represents that extensions can access all the HTML content that the browser has rendered. The transition from \( s_0 \) to \( s_1 \) models that, when the translation extension is activated, if mybank.com is opened in a tab then the automaton changes to state \( s_1 \). In state \( s_1 \) the policy \( P_{HTML-NumericalDigits} \) means that extensions can get the whole HTML except for numeric values. Finally, if the automaton is in state \( s_1 \) and the translation starts, but the tab mybank.com is not present then the automaton enables again the policy \( P_{HTML} \). When an event occurs but the condition is not satisfied the automaton remains in the same state.

Note that, in the previous example, it is possible for another malicious extension to modify the numerical digits of the HTML so that the information is sent to the translation server. This is known as collusion attack and protecting against this type attacks would require a more sophisticated policy. Policy automata can also be used to prevent instances of this type of attacks. The sequence of events which occurs during the collusion can be specified in the automaton, and the required policies can be activated.

Policy automata define the behaviour of monitors that will run in parallel together with the web browser. The monitors will be in charge of activating and deactivating the
static policies according to the specification in the automaton. Policy automata can be automatically compiled to Java monitors using LARVA [7]. However, we are still evaluating other approaches to directly implementing the monitors that are more targeted for our setting.

3 Discussion

We have identified browser extensions as a potential source of personal information leakage. We are currently looking into policies that can be enforceable using our technique. In particular, we are focusing in the implementation of an extension or a plug-in for the web browser Chromium. The events and information that can accessed as well as the (static) policies that can be enforced in Chromium will determine the type of policies that we can effectively implement.

Related Work. In [4] authors proposed an application that classifies extensions according to some parameters (developer reputation, code base or behaviour) as malware and they were automatically removed from the Chrome Store. A similar work named Hulk was proposed in [5] were extensions were classified by identifying suspicious behaviours. In our proposal, we go one step further and not only do we detect whether a browser extension is or is not disclosing sensitive information about the user but we also create a mechanism to avoid that sensitive information — without skipping the execution of the browser extension which is leaking our data — will be sent.

Acknowledgements This research has been supported by the Swedish funding agency SSF under the grant Data Driven Secure Business Intelligence and the Swedish Research Council (Vetenskapsrådet) under grant Nr. 2015-04154 (PolUser: Rich User-Controlled Privacy Policies).
Bibliography


