After you, please: browser extensions order attacks and countermeasures

Citation for the original published paper (version of record):
Picazo-Sanchez, P., Tapiador, J., Schneider, G. (2020)
After you, please: browser extensions order attacks and countermeasures
International Journal of Information Security, 19(6): 623-638
http://dx.doi.org/10.1007/s10207-019-00481-8

International Journal of Information Security (2020) 19:623–638
https://doi.org/10.1007/s10207-019-00481-8

REGULAR CONTRIBUTION

After you, please: browser extensions order attacks and countermeasures

Pablo Picazo-Sanchez[1,2] · Juan Tapiador[3] · Gerardo Schneider[1,2]

Published online: 21 November 2019

Abstract
Browser extensions are small applications executed in the browser context that provide additional capabilities and enrich the user experience while surfing the web. The acceptance of extensions in current browsers is unquestionable. For instance, Chrome's official extension repository has more than 63,000 extensions, with some of them having more than 10M users. When installed, extensions are pushed into an internal queue within the browser. The order in which each extension executes depends on a number of factors, including their relative installation times. In this paper, we demonstrate how this order can be exploited by an unprivileged malicious extension (i.e., one with no more permissions than those already assigned when accessing web content) to get access to any private information that other extensions have previously introduced. We propose a solution that does not require modifying the core browser engine, since it is implemented as another browser extension. We prove that our approach effectively protects the user against *usual* attackers (i.e., any other installed extension) as well as against *strong* attackers having access to the effects of all installed extensions (i.e., knowing who did what). We also prove soundness and robustness of our approach under reasonable assumptions.

Keywords Web security · Privacy · Browser extensions · Malware · Chrome

1 Introduction

Web browsers have become essential tools that are installed on nearly all computers. The most popular browsers as of this writing (April 2018) are Chrome (77.9%), Firefox (11.8%), Internet Explorer/Edge (4.1%), Safari (3.3%) and Opera (1.5%) [35]. Most browsers allow users to install small applications, generally developed by third parties, that provide additional functionality or enhance the user experience while browsing. Such plug-ins are known as *browser extensions* and they interact with the browser by sharing common resources such as tabs, cookies, HTML content or storage capabilities. As of May 2017, the Chrome Web Store[1] (the official repository where all Chrome extensions are stored and distributed) contains more than 135,000 extensions, whereas for the case of the second most popular browser (Firefox), its extension store contains almost 70,000 items.[2]

When an extension is installed, the browser often pops up a message showing the permissions this new extension requests and, upon user approval, the extension is then installed and integrated within the browser. Extensions run through the JavaScript event listener system. An extension can subscribe to a number of events associated with the browser (e.g., when a new tab is opened or a new bookmark is added) or with the content (e.g., when a user clicks on a HTML element or when the page is loaded). When a

This work was partially supported by the Swedish Research Council (*Vetenskapsrådet*) through the Grant PolUser (2015-04154), the Swedish funding agency SSF under the Grant Data Driven Secure Business Intelligence, the Spanish Government through MINECO Grant SMOG-DEV (TIN2016-79095-C2-2-R) and by the Comunidad de Madrid under the Grant CYNAMON (P2018/TCS-4566), co-financed by European Structural Funds (ESF and FEDER).

✉ Pablo Picazo-Sanchez
pablop@chalmers.se

Juan Tapiador
jestevez@inf.uc3m.es

Gerardo Schneider
gersch@chalmers.se

1 Chalmers University of Technology, Göteborg, Sweden

2 University of Gothenburg, Göteborg, Sweden

3 Universidad Carlos III de Madrid, Madrid, Spain

[1] https://chrome.google.com/webstore/category/extensions.

[2] https://addons.mozilla.org/.

JavaScript event is triggered, the event is captured by the browser engine and all extensions subscribed to this event are executed.

In this paper, we focus on Chromium [23], which is an open source browser and the basis for Chrome, Opera, Brave, Edge Chromium or Yandex browsers. Extensions installed in Chromium can also run in all mentioned browsers. The execution engine is exactly the same in all the browsers and follows the same pipeline model that will be explained in some detail later (Sect. 3.1). For this reason, we will refer to Chrome and Chromium interchangeably. Extensions in Chromium can be of three types: *content scripts*, *background pages* or both. In what follows, our main focus is on content scripts, which are JavaScript files that run in the context of the loaded web page. It is important to emphasize that the main aim of content scripts is to access and interact with the *Document Object Model (DOM)*. This fact alone raises a fundamental privacy question, since it is explicitly assumed that extensions will have full access to any (sensitive or not) content that the user is accessing. Browsers (including Chromium) dodge this issue by assuming that the user should trust the extension before installing it. In this paper, we do not address this problem, which is essentially related to determining if an extension's behavior is benign or malicious, but a related one described in what follows.

1.1 Extension order attacks

When analyzing the security and privacy implications of browser extensions, one question that has been largely overlooked is the potential leakage of information among extensions. In nearly all browsers, each content script uses its own wrapper of the DOM to read and make changes to the page loaded by the browser. They also run in a dedicated sandbox that the browser provides for security reasons. However, there is no isolation in terms of privacy, since all changes an extension performs in its own DOM are automatically synchronized with the main DOM. One straightforward—but nonetheless important—consequence of this is that a malicious extension could eavesdrop on other extensions (i.e., it can get access to the data they put on the DOM and observe their actions) and even manipulate their behavior by acting on their DOM elements (e.g., clicking on elements introduced by another extension). An attacker can exploit this using two different strategies:

1. *Exploiting the order* The way Chromium manages extensions (see Sect. 3.1) introduces a default execution order among extensions with undesirable consequences. One key issue is that the nth extension in the pipeline can learn all contents introduced by the first $n - 1$ extensions in the HTML document. Thus, extensions located at the end of the pipeline enjoy more privacy than ones installed earlier. More importantly, the order could be explicitly exploited, eventually producing privacy leaks and security problems.
2. *Order-independent attacks* Some attacks enabled by the absence of effective isolation among extensions' actions do not require exploiting the execution order (i.e., getting the malicious extension to be placed at the end of the execution pipeline). However, exploiting the order provides the attacker with a privileged position that facilitates such attacks, which will result in a simpler code for the malicious extension that will increase the chances of passing the analysis performed by official stores [17]. Furthermore, not all attacks might involve adding event handlers, since access to information put in the DOM will only be possible once the attacked extension has executed.

We have experimentally verified the previous attacks and demonstrated, for instance, that an extension with no privileges can learn which pictures a user likes in Pinterest or change the picture a user wants to share; that it can tamper with the notes and events provided by the popular Evernote Web Clipper; or that it can profile the user's video browsing preferences in YouTube (see Sect. 3.3 for details.) This lack of effective isolation is not only inherent to Chromium's extension model, but also explicitly acknowledged. Browsers such as Chrome do not even attempt to guarantee some form of "non-interference" among extensions. On the contrary, developers are encouraged to implement appropriate mechanisms to protect any sensitive information that ends up in the DOM, since it is assumed that other extensions could simply read or manipulate it. Even if browsers do not factor this into their threat model, we believe that this is a serious vulnerability that has not been discussed before. More importantly, it can be easily exploited by a malicious extension, regardless of the fact that it is explicitly assumed in the browser's extension model or not.

1.2 Our contributions

In this paper, we make the following contributions:

1. We discuss a vulnerability inherent to the way extensions are handled in Chromium. The problem originates in the fact that extensions can effectively interfere with each other, which can be exploited by an attacker to access sensitive information injected by other extensions, and also to manipulate their implemented event handlers. To the best of our knowledge, this is the first work that discusses this security and privacy threat.
2. We formalize this problem in terms of knowledge gained by the attacker. In particular, we establish what the *default* knowledge any extension has, and then define what an attacker might get to know based on her attacking capa-

bilities. A *usual* attacker is basically any other installed extension just taking advantage of its position in the execution pipeline, while a *strong* attacker has the capability of knowing the effect of the execution of each extension.
3. We propose a solution that provides practical security isolation among extensions and does not require altering the core browser engine. The key idea is to replace the extension pipeline by a (simulated) parallel execution model in which all extensions receive the same input page (see Fig. 1). An additional component identifies the changes introduced by each extension and applies all of them to the original input page. We prove properties (soundness and robustness) of our solution and also discuss limitations of this approach.
4. To facilitate the reproducibility of our results, we make our implementation freely available.[3]

The rest of this paper is organized as follows. A brief background on browser extensions and the architecture is given in Sect. 2. In Sect. 3, we describe Chromium's extension model in some detail, characterize the threat posed by a pipeline-based execution model for extensions and discuss attacker models. Section 4 describes our solution and discusses the main advantages, properties and limitations of our approach. Section 5 reports the experimental results obtained with our implementation. Finally, Sect. 7 discusses related work and Sect. 8 concludes the paper.

2 Chrome browser extensions

A browser extension is basically a collection of packaged files which can perform specific operations in the client browser and that can interact with the HTML file accessed by the user. In Chrome, it is mandatory for browser extensions to have a JSON file named `manifest.json` that contains information about the extension such as its name, permissions and capabilities that it is allowed to use, and meta-data, among others. Browser extensions may consist of one or more JavasScript or HTML files, as well as of additional resource files such as *Cascading Style Sheets (CSS)*, text, fonts or images. Table 1 shows the most used file types in Chrome extensions. This statistic has been obtained by running a static analysis over 173,553 extensions and some of their versions found in Chrome's official repository (Chrome Web Store).

Browser extensions may be formed by *background scripts*, *content scripts*, or both. Roughly speaking, the main difference between background pages and content scripts is that the former is not allowed to directly interact with the DOM, but it can use the chrome API to interact with the browser

Table 1 Files frequency in Chrome extensions

File type	Number	File type	Number
JSON	173,564	MIN	46,810
PNG	167,831	SVG	29,394
JS	137,325	GIF	27,771
HTML	106,383	TTF	24,885
CSS	86,772	WOFF	24,761

events, e.g., get the number of installed extensions, the user's history, retrieve the browser cookies, be able to know when the user opens/closes tabs. In contrast, content scripts are focused on the final representation and basically interact with the content. (Content scripts can also use a small subset of the chrome API.) In content scripts, extensions can modify the DOM and be run when some events are fired such as clicks on elements, when the page is loaded or when a form is sent, among others.

Extensions that follow a background page architecture means that there is an HTML file that implements the extension behavior. It is worth noting that it is mandatory to have at least one HTML file, which may eventually contain JavaScript code (or links to other JavaScript files that can be stored in the same extension or allocated in external servers). Moreover, there are two different types of background page extensions: *persistent background pages* and *event pages*. Persistent background pages are extensions that are always running as soon as the browser is opened, while event pages are extensions than run only when needed, i.e., extensions can subscribe to some JavaScript events (`addListener`), and they remain idle until any of those events is fired.

On the other hand, extensions that follow a content script architecture are aimed for an explicit interaction with the DOM. This means that such extensions can access all the information of the HTML and, thus, they might interact (alter, delete, insert, etc.) with the page contents. This kind of extensions, however, cannot directly modify the DOM of the extension (the background part). So, a content script is some JavaScript file(s) injected in the context of a page that has been loaded into the browser, and it may be seen as part of that loaded page, not as part of the extension it was packaged with (its parent extension).

Content scripts and background extensions work in two different worlds where direct communication between them is banned to avoid possible *Cross-Site Scripting (XSS)* attacks and information leakages. Nevertheless, they are not totally separated from each other. Both kind of architectures can share information and collaborate through some predefined *extension message passing*. Figure 2 illustrates the architecture of browser extensions that can work as a background to interact with the browser, as a content script to interact with

[3] https://github.com/Pica4x6/Ghost_Extensions.

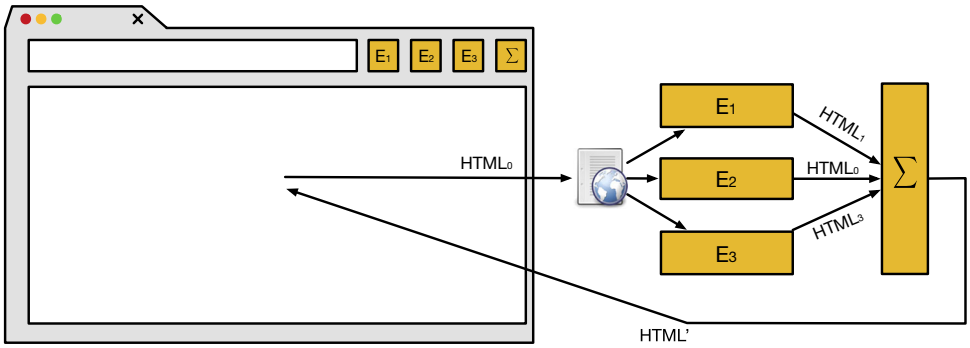

Fig. 1 Modified extension execution pipeline in our solution

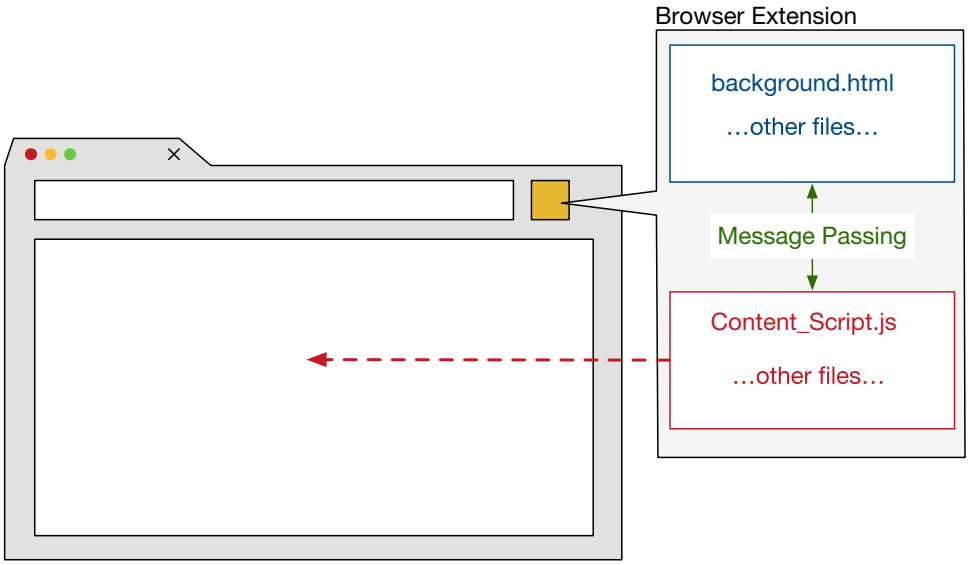

Fig. 2 Browser extension taxonomy

the content, or both at the same time and communicate with each part by using message passing.

2.1 Events order in JavaScript

In JavaScript, the *event propagation* mechanism determines in which order an event is received by HTML elements. For instance, when two nested elements are subscribed to the same event (e.g., `div1` and `div2` subscribes to click in Fig. 3), and this event is fired, there are basically two ways to propagate the event in the DOM: *bubbling* and *capturing*. By using capturing, the event is initially handled by the root element and propagated to its children. In contrast, with bubbling the event is initially captured and handled by the children (leaves nodes in the DOM tree) and then propagated to their parents.

In JavaScript, extensions subscribe to events by using the `addEvenListener()` function. In our example above, we use capturing, so the first *alert()* will correspond to the `<divid="1">`. In case of using bubbling, then the first `alert()` will correspond to the `<divid="2">` element.

Apart from the JavaScript event propagation mechanism, extensions developers can define one additional *order level* through a property named `run_at` in the `manifest.json` file that allows them to control at which moment the extension will be injected. That property has three possible values: `document_start`, `document_end` and `document_idle`. When the value is set to `document_start`, the extension is injected when the document element is created. Setting it to `document_end` would cause the extension to be injected when the DOM is completed but before any other subresources are loaded, such as e.g., images, iframes, etc. Internally, Chromium loads the extension when the `DOMContentLoaded()` is triggered. Finally, the `document_idle` value would cause the extension to be injected after `document_start` and before `document_end`, that is, once the page has been created and after the DOM is loaded. Internally, Chromium loads the extension after the `window.onload()` is fired.

Additionally, Chrome currently works by delegating to the HTML parser the way the content scripts are inserted when they are tagged as either `document_start` or `document_end`. Thus, if the HTML parser schedules `document_start` or `document_end` as tasks, then content scripts are inserted in separate tasks. However, content

```
<!DOCTYPE html>
<html>
   <body>
      <div id="div1" style="border:1px solid red; width:10px;height:10px" >
         <div id="div2" style="border:1px solid black;width:5px;height:5px">
         </div>
      </div>
   </body>
   <script>
      capturing = true;
      document.getElementById('div1').addEventListener('click',function(){alert
          ('1')},capturing)
      document.getElementById('div2').addEventListener('click',function(){alert
          ('2')},capturing)
   </script>
</html>
```

Fig. 3 JavaScipt event flow

scripts tagged as `document_idle` are always injected in separate tasks.

Despite the existence of the aforementioned strategies to control the execution order of extensions, explicitly writing them does not unequivocally determine the order in which they will be executed. Whenever two or more extensions have the same configuration parameters, the Chrome extension engine decides which one will be executed based on the extensions' installation date. This behavior follows a FIFO policy: The oldest installed extension will be the first to be executed whereas the newest will be the last one to be executed.

Apart from the event management mechanism explained above, it is worth mentioning how Chrome manages tasks and microtasks. A task—a click event, for instance—is run in its own thread and is composed of a set of JavaScript sentences, actions to handle the event, which belong to the *event loop*. All tasks are queued and executed sequentially. Moreover, when a `setTimeout` is used in the event loop, the callback function that is executed asynchronously is queued as a new task. This is specially useful for monitoring delayed functions in browser extensions.

Nevertheless, some operations can be also executed in the middle of a task execution, e.g., to make something asynchronous without being scheduled as a new task and queued in the tasks queue. Those operations are called microtasks (composed of promises and mutation observers) and are executed intermediately after the task execution. The main reason for this new types of queues is to enrich the user experience. However, microtasks are not executed when the event loop is not empty, e.g., if two click events are fired using JavaScript code and the function that handles the click event uses promises, those microtasks will not be run until both clicks events are executed. We refer the reader to [8] for more information and practical examples about how tasks, microtasks and how their execution queues work. In this work however, we are not taking microtasks into consideration and they are left as future work as it is mandatory to modify the Chromium's source code to take them under control.

2.2 Extensions in Chromium's source code

The security model of browser extensions in Chromium is based on isolated worlds in (JavaScript) V8. Its main purpose is to isolate the execution of different untrusted content scripts (with a wrapper of the original DOM) while keeping the main DOM structure synchronized.

Essentially, a *world* is a "concept to sandbox DOM wrappers among content scripts" [12]. Each world has its own DOM wrapper, yet there might be different instances from one particular world and, thus, all of them would share the same Blink C++ DOM object. The main reason for this partition is that instances belonging to the same world cannot share DOMs but can share C++ DOM objects, i.e., no JavaScripts can be shared between different worlds but C++ DOM objects can be, thus permitting to run untrusted content scripts on shared DOM.

Roughly speaking, in terms of browser extensions this world concept means that the content scripts of each extension will run its own JavaScripts over different DOMs. However, all these DOMs are synchronized so that all changes made by each individual JavaScript will automatically be sent to other DOMs (other wrappers and the main DOM the user sees).

According to the official documentation, V8 has three different *worlds*: a main world, an isolated world and a worker world. A main world is where the original DOM with all its original scripts are executed. An isolated world is where the content scripts of the extensions are executed—all of them can access the main DOM through Blink C++ shared objects. Finally, a worker world is associated with threads in such a way that each isolated world is associated with one worker,

i.e., the main thread is the main world plus each of the content scripts. Figure 4 represents how Chrome manages and isolates content scripts of browser extensions.

Overall, this isolation mechanism prevents Chrome from being vulnerable to attacks such as the one recently demonstrated in [26] against Firefox, whose security model lacks isolation. Nevertheless, Chrome's security model has not considered privacy between extensions as part of its architecture. This allows, for instance, that if the Pinterest extension inserts a `<span>` element on each picture contained in the DOM, then all these changes will automatically be visible to the rest of the extensions, regardless of whether they run in isolated worlds. This observation is the basis for the attack discussed in this paper.

3 Attacker model

In this section, we describe how the extension engine that Chromium implements introduces both privacy and security risks by default. In particular, we show how a simple malicious browser extension can exploit those issues with no more privileges than having access to the DOM. We then introduce a formal model for the execution pipeline of extensions and describe our threat model.

3.1 Chromium's extension execution model

Chromium manages all installed extensions through the class *ExtensionRegistry* implemented in *extension_registry.cc* and the associated header file *extension_registry.h*. This class implements methods to add, remove or retrieve all extensions that for a particular browser context have been enabled, disabled, blocked, blacklisted, etc. Each set of extensions is internally managed through the *ExtensionSet* class implemented in *extension_set.h* and *extension_set.cc*. This is just a standard C++ map to manage sets with methods to insert, remove and retrieve items. Importantly, it also provides a standard C++ iterator to enumerate all set elements. Overall, this means that installed extensions in Chromium are placed in a pipeline and are called sequentially to inject the content script(s) in the DOM. Apart from that, the position at which the browser injects content scripts is determined by a number of factors:

1. First, by the implicit order declared in the *run_at* property in the *manifest.json* file.
2. If two or more extensions have the same *run_at* value, the browser tries to determine their execution order using the JavaScript event propagation mechanism [18].
3. Finally, if the event propagation order is the same, extensions are executed according to their installation time: A executes before B if B was installed after A.

This pipeline works as illustrated in Fig. 5. The content script of the first extension is inserted and then it is executed in an isolated world by using the method *executeScriptInIsolatedWorld* (see [12] for more details about worlds in Chrome). The output of an extension is automatically synchronized with the shared DOM, so when the next extension is executed its wrapper DOM already contains all the changes that previous extensions have made.

3.2 Manipulating the execution pipeline

We assume that the attacker gets one malicious extension at the end of the execution pipeline. Even if the extension is not the last to be installed, Chromium's extensions model provides various mechanisms that an attacker can exploit to modify the order in which parts of the extension code will run. For example, a malicious extension can be marked to run once the DOM is loaded. This is achieved by setting the *run_at* property to *document_end* in the *manifest.json* file. Additionally, the extension can use the *capturing* JavaScript event propagation property to force that the fired event would execute the extension in the return journey of the event propagation (check [18] for further details about this). Moreover, modifying the default execution order could be done by following two different approaches:

1. Through another extension's *management* permissions, similarly to what the *Extensity* extension does [11]. Essentially, this extension enables and disables extensions automatically in the browser. The attacker will disable all installed extensions and then re-enable them again, but putting the malicious extension at the end. For this to work in practice, the user must explicitly approve the malicious extension requirement to extend its permissions (namely, management). Since many users do not pay attention to requested permissions, this could guarantee success for the attacker.
2. Modifying the *Secure Preferences* file. This is a JSON configuration file that was initially thought to be modified only by the browser [6]. However, Chrome allows developers to distribute extensions as part of other software so this file could also be externally modified by other processes. This is the basis for most of the malware installed in the browser because of its deficient security [15]. The attacker can thus modify the *install_time* property in the *Secure Preferences* file, and put her extension at the end of the pipeline. See [32] for a detailed explanation on how to modify the manifest file.

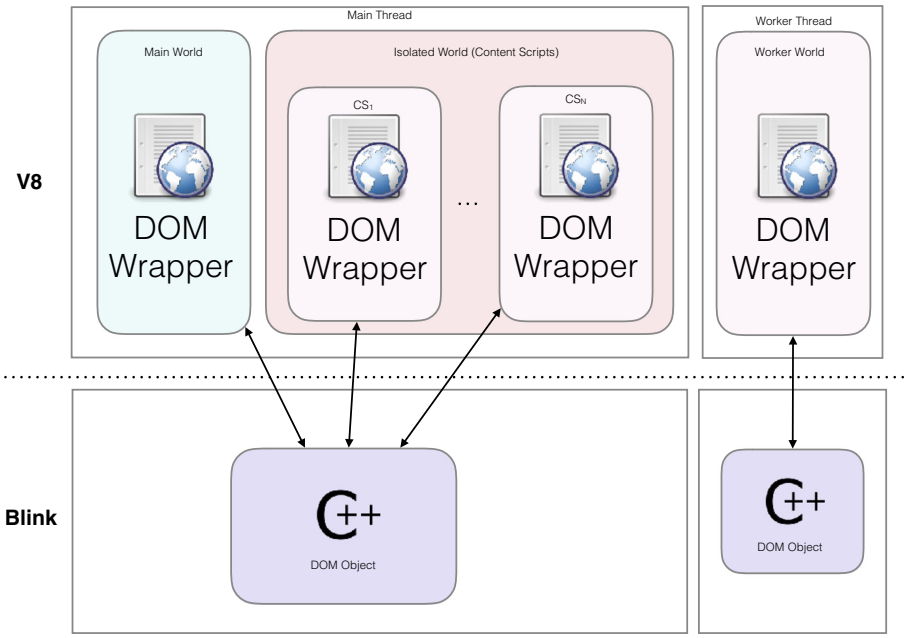

Fig. 4 Browser extensions architecture in chromium

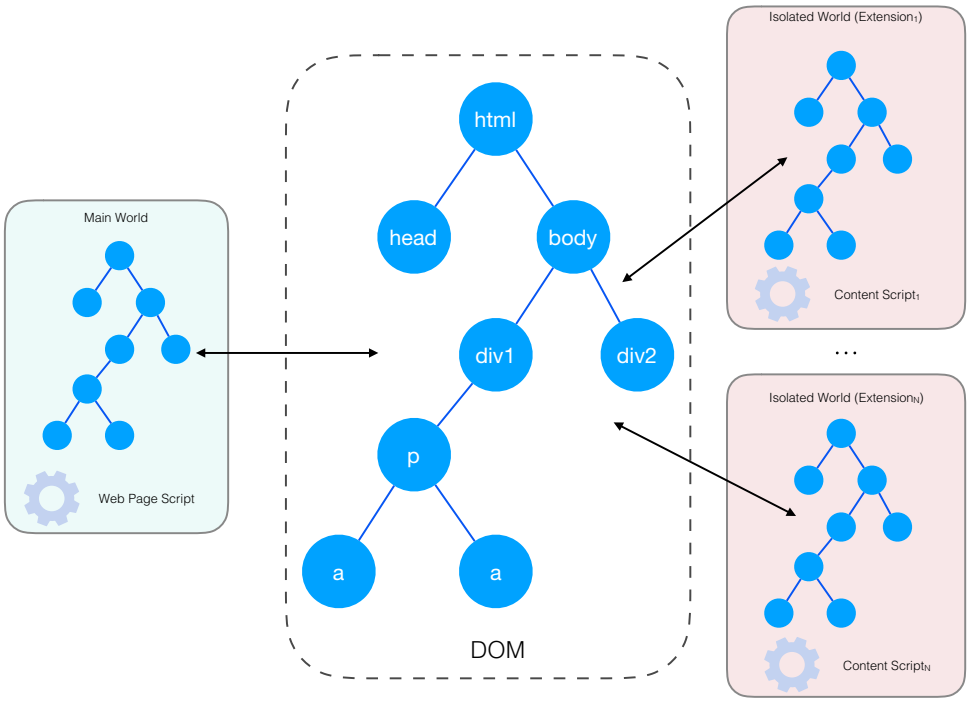

Fig. 5 Chromium's execution model for extensions

3.3 Attack examples

To exemplify the problem, we tested the attack in a Chrome browser with four extensions already installed: Pinterest, Evernote, vidIQ Vision for YouTube and our custom extension. Our malicious extension subscribes to all possible JavaScript events in the browser (i.e., in the web page context). This guarantees that the extension will always be executed whenever any event is fired.

The official extension of Pinterest, which has more than 10M users, parses the entire content and adds hidden ** elements on each picture it finds in the DOM, as well as some CSS elements. When the user triggers the *onmouseover()* event by passing the mouse pointer over a picture, the span becomes visible in the form of a button and, if the user clicks on it, the picture is automatically shared on her Pinterest board. Assume now that the user has some "secret boards" defined in her Pinterest account to avoid sharing pictures

with all the world. Our malicious extension can carry out the following actions:

1. It can add a listener to the same *onclick()* JavaScript event to know which pictures the user adds to her account, and thus the photographs will no longer be private.
2. It can learn what pictures the user likes and it could share that information to an advertisement company [37].
3. It could generate a *click* JavaScript event on each picture, automatically sharing all pictures in the user's account without any confirmation pop-up.
4. It can replace the picture the user wants to share by another one.

Evernote Web Clipper has currently more than 4.5M users. The extension parses the web page and inserts some CSS code and hidden <*span*> elements on each picture contained in the DOM. Additionally, it adds a contextual menu when the user performs a right click either on a single tag or in the whole document. Using this contextual menu, the user can add items such as meetings, personal notes, or any other information to her calendar. Our malicious extension can subscribe to the click events and, in addition to the attacks described for the Pinterest scenario, it could also learn all details about the notes or calendar entries added by the user.

Finally, we tested it against vidIQ Vision for YouTube. This extension has more than 500,000 users. Among other actions, it inserts a <*div*> element in the right banner of the screen when a user visits Youtube in order to provide her with richer information and track her viewed videos. When the user visits Youtube for the first time, this extension asks her for her username and password. (As a matter of fact, all extensions subscribed to either *onkeydown()*, *onkeypress()* or *onkeyup()* events may get both the username and password.) Our malicious extension, apart from getting the username and password, could also get all viewed videos and profile the user's habits.

3.4 Modeling extension effects

Before formally defining our attacker model, we first introduce some notation and definitions. In what follows, $E = \langle E_1, \ldots, E_i, \ldots, E_n \rangle$ $(n > 0)$[4] will denote the "set" of extensions already installed in the browser, where the index indicates their default execution order (i.e., E_1 is the first to be executed).

When extensions are executed, they have an *effect*. For our purposes, we split such effects into two parts: a *functional* effect that is reflected on the changes done to the DOM the extension acts on, and some *side-effects* that are not directly reflected in the DOM (e.g., sending information to other servers, interacting with the browser, executing external scripts, etc.). The functional effect of an extension E_i when applied to a DOM will be denoted by $f_i(\text{DOM}) = \text{DOM}_i$. In this paper, we are only concerned about the functional effect of DOMs, so all the results that follow only apply to what extension can do on the DOMs and, thus, no claim is done concerning extensions' side-effects.

Extensions can perform four different types of high-level operations while being executed: *insertions*, *deletions*, *updates*, and simply doing *nothing*. An extension E_i does nothing when the result of its execution is the same as the input. As expected, the effect of the other operations (insertions, deletions and updates) implies that the new DOM is modified by the corresponding change.

Definition 1 Let $E = \langle E_1, \ldots, E_i, \ldots, E_n \rangle$ with $n > 0$, be the set of extensions that a browser has already installed and DOM_0 the original content provided as input. We define the *execution pipeline* as the result of the execution of the n extensions as composite functions: $f_n \circ \ldots \circ f_1(\text{DOM}_0) = \text{DOM}_n$.

DOMs can be seen as trees [7]. We will use this fact to define the above operations in terms of operations on trees. Thus, if extension E_i only inserts elements in the DOM, then, $\text{DOM} \subseteq f_i(\text{DOM})$. In case E_i only deletes something from DOM, then, $f_i(\text{DOM}) \subseteq \text{DOM}$. Finally, if E_i only updates DOM, then $f_i(\text{DOM}) = \text{DOM}_i$ where DOM is equal to DOM_i except for the field that has been updated.[5]

We assume a tree operation that allows us to compare DOMs and give us the *difference* between them: $\text{DOM} - \text{DOM}'$. Moreover, we say that DOM is *smaller or equal* than DOM' (denoted $\text{DOM} \leq \text{DOM}'$) if and only if DOM is a subtree of DOM'.[6]

Finally, we say that the *default* knowledge of an extension is the amount of information it can get from the DOM at the moment of its execution. Note that the actual knowledge of an extension might not be equal to the default knowledge.

Note that the *real* knowledge of an extension might not be equal (and neither a subset nor a superset) of the default knowledge. The reason is that, as we will see, this knowledge might be affected by attacks or by a solution to those attacks. The concept is in any case useful as it characterizes what the extension knows by default, if no external interference is added to the expected behavior of how the browser works.

If an execution pipeline is such that the overall functional effect of all extensions is only insertions or doing nothing, we say that the execution pipeline is *monotonic with respect to the structure of the DOM* (or simply, that it is *monotonic*).

[4] All the discussion below assumes that there is at least one extension installed.

[5] Note that, to avoid over-formalization, we are not giving formal definitions for these operations in terms of trees as they are rather intuitive.

[6] We define $<$, $>$, and $\geq$ as expected.

Conversely, if any extension E_i in the execution pipeline deletes or updates information, then it is generally impossible to make any statement about whether any other extension knows more or less than E_i. For instance, an extension E_{i-1} could delete something while extension E_i adds it back, in which case any other extension E_j $(j > i)$ will not be able to detect that there has been a deletion in the past.

3.5 Attacker model

We consider two different types of attackers: *strong* and *usual* attackers. Intuitively, a *strong attacker* is a malicious extension that has access to the output of all executions in the pipeline. Note that this provides the attacker not only with the effect of all extensions, but also with knowledge about which extension did what. Alternatively, a *usual attacker* is a browser extension that only has access to the corresponding DOM that the extension receives as input when it is executed (plus the original DOM). More formally:

Definition 2 A *strong attacker* $(\mathcal{A}_s)$ is an extension $E_{\mathcal{A}_s}$ that is interleaved in the execution pipeline such that $f_{\mathcal{A}_s} \circ f_n \circ f_{\mathcal{A}_s} \circ \cdots \circ f_{\mathcal{A}_s} \circ f_1 \circ f_{\mathcal{A}_s}(\text{DOM}) = \text{DOM}_n$. This is the strongest attacker because it can know all the changes that all extensions have performed. A *usual attacker* $(\mathcal{A}_u)$ is an extension $E_{\mathcal{A}_u}$ that is executed in the jth position of the pipeline $(j \leq n)$ such that $f_n \circ \ldots \circ f_{\mathcal{A}_u} \circ \cdots \circ f_1(\text{DOM}_0) = \text{DOM}_n$, having the default knowledge any other extension in position j could have. Note that $j > 1$ as otherwise the attacker would learn nothing.

A strong attacker has definitively more knowledge than any other in the pipeline and can thus take advantage of that. Note that, in particular, a strong attacker gets to know which extension did what changes since it can calculate the effect of each extension. The usual attacker can only infer partial information about the other extensions by diffing DOM_0 and the $\text{DOM}_{\mathcal{A}_u}$ that it receives as input. However, this attacker will know neither the number of extensions nor which operations they have performed over the content. Note that the gain of knowledge is not much over previous extensions except if $\text{DOM}_{\mathcal{A}_u}$ is part of a monotonic subsequence.

An interesting consequence of our threat model is that all extensions which are installed on the browser are potential usual attackers because they might have access to the original DOM and to the input DOM received from the previous extension in the execution pipeline.

Remember that despite our proposed attack might be performed without exploiting the order, i.e., a malicious extension could subscribe to all possible events in the DOM, the amount of needed source code to tackle all possible privacy attacks would be incredibly huge and infeasible due to the amount of possible extensions and attacks.

In this work, we remark the existence of this security threat which is transparent even for the automatic static analysis of the source code that official repositories perform [17]. Notice that by using our attack, the simplest dummy extension installed just after, for instance the official Pinterest extension, would detect the existence of the former one, and thus, it can communicate to an external server to retrieve the customized exploit performing thus an adaptive attack.

Additionally, our scenario can handle situations where, two browser extensions developed by the same person/company but placed for instance at the beginning and at the end of the execution queue will actually access to different information and thus, collaborate to perform attacks like browser hijacking [25,27,37], or fingerprinting [21,31] attacks.

4 Our solution

In this section, we describe our solution to address the non-isolation problem among extensions described previously. We first provide a general overview that sketches the main ideas behind our approach. We then describe in more detail our approach and discuss its main advantages, properties and limitations.

4.1 Approach

Our solution introduces the notion of a *monitor extension*, whose goal is to prevent regular extensions from learning from each other. Intuitively, monitor extensions are used to detect all changes that an extension makes; log those modifications; delete them from the DOM passed on to the next extension in the execution pipeline; and, at the end of the pipeline, merge all changes to produce a final DOM. Figure 6 shows the four main components of our scheme:

- The *Diff* module takes a pair of DOMs (namely, $(\text{DOM}_{i-1}, \text{DOM}_i)$ and performs the difference between them $(\mathit{Diff} = \text{DOM}_i - \text{DOM}_{i-1})$.
- The *Store* module is shared between all monitor extensions and collects all changes in a table. This table can be seen as a *patches* table with the following format: <*Operation*>, <*Position*>, <*Action*>.
- The *Del* module removes all changes from DOM_i, that is, $\text{DOM}_i = \text{DOM}_{i-1} - \mathit{Diff}$.
- The *Apply* module, which is placed at the end of the pipeline, takes all stored differences and patches the DOM by applying them in order.

Note that our solution could be simplified by removing the *Del* method. Thus, once the difference has been computed, the DOM passed on to the next extension would be

just the original DOM (DOM_0). However, by implementing a *Del* module, our approach is more general since it allows to introduce some policies to share limited amounts of information among extensions. This point, however, is not further explored in this work.

4.2 Extension-based implementation

Our extension-based implementation is slightly different from the architecture described above. This is due to some constraints related to the scope of the information that can be filtered out (i.e., diffed) and to issues on how to isolate extensions without modifying the browser's source code. Because of that, we need to differentiate between four types of extensions:

1. A special *initial extension* that must be placed in the first place of the execution pipeline to get the original content (DOM_0). This extension might be forced to be in the first place by using the *capturing* event handler and *"run_at":"document_start"* in the *manifest.json* file. Since we cannot guarantee that this special extension will be placed in the first position, we can then force it to be first by manually modifying the order, i.e., disabling all extensions and start enabling them in the order we want them to be executed.
2. *Official extensions*, i.e., those extensions that the user can install from the Chrome Web Store.
3. *Monitor extensions* which are interleaved between each pair of official extensions. They are in charge of performing the *diff/store* and *del* operations.
4. A special *final extension* that must be placed in the last position of the execution pipeline to merge (patch) all changes that all official extensions have performed previously.

More formally, this solution implements the following transformation over the input DOM : $f_{E_{final}} \circ f_{E_n} \circ f_{E_{monitor_{n-1}}} \circ \ldots \circ f_{E_{monitor_1}} \circ f_{E_1} \circ f_{E_{initial}}(\mathrm{DOM}_0) = \mathrm{DOM}_n$. The information flow is as follows. Assume that Alice accesses a web page. The browser requests the URL and, once the DOM_0 tree is retrieved, the first isolated world corresponding to the initial extension ($E_{initial}$) is executed. This first extension is not part of the general solution (see Fig. 6), but we found out that, when we tried to implement it in a real setting, it is needed because Chrome—and other browsers in general—do some pre-processing to the DOM (e.g., closing forgotten open HTML tags, adding some mandatory HTML tags or changing everything into lower case). This initial extension does not add any changes to the DOM. We note that an extension can request the same content directly by using the *XMLHttpRequest* JavaScript object, but the received DOM could be completely different from the current DOM because of that browser pre-processing.

After that, the output of the initial extension DOM_0 and the rest of the DOMs wrappers are synchronized. At this point, the first official extension is run and may perform some actions over the content. The resulting (DOM_1) is the input to the next *monitor extension*, plus the initial DOM (DOM_0) needed to get the difference between both DOMs: $Diff = \mathrm{DOM}_1 - \mathrm{DOM}_0$. All the possible resulting values of this operation are stored ($Store$) for the final post-processing (*patch* operation), and the difference *Diff* is then removed from the output of the extension DOM_1. It is worth noting that this new DOM will be equal to the original DOM in most cases, i.e., our solution will be valid whenever the execution pipeline follows a monotonic sequence. This process is repeated until the last official extension eventually produces the final DOM_n output. This last DOM is then provided as input to the final extension (E_{final}), which will take all stored changes and will apply them to the DOM_n, thus generating the final document.

We have produced two different implementations of this architecture. In our first simple approach, in order to check which operation each extension performs over the content, we insert a <*textarea*> element in the DOM to keep track of all changes. It also shows all sensitive information that extensions have access to. This <*textarea*> is only available to gather meta-information during the experimentation and should be removed from the deployed version. For the second implementation, we created an extension which communicates with an external server to store all differences between two DOMs. This was needed to emulate what an adaptive attack could achieve by analyzing externally (i.e., out of the browser) the information gathered locally from other extensions. For this, we used the simplest version of a Flask server—a lightweight server written in Python—and a library named difflib [10], which is part of the standard Python library. Alternatively, we could have implemented a full client-based solution by using JavaScript libraries such as jsdiff [19], though our implementation proved to be enough for a proof-of-concept prototype.

The source code of the initial extension can be seen in Fig. 7. Note that the property *run_at* of the *manifest.json* file is set to *document_start* (see Fig. 7a), whereas the original content, i.e., DOM_0, is retrieved in line 5 of the *ContentScript.js* file of the extension (see Fig. 7b). Once the <*textarea*> area is created, each monitor extension simply checks the current content against the original one to extract what official extensions do, delete such changes (if any) and stores them in the <*textarea*> for the *final extension* to include them. Similar to the initial extension, we include a simplified version of the source code we use to execute the *final extension* at the end of the pipeline (see *run_at* property set to *document_end* in Fig. 8a) and our JavaScript (see

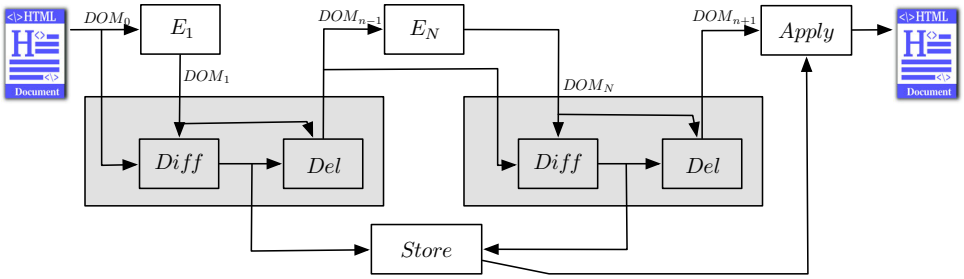

Fig. 6 Architecture of our solution and its four main modules

```
"content_scripts": [{
"matches": ["<all_urls>"],
"js": ["content.js"],
"run_at": "document_start"
}],
```

(a) *Manifest.json*

```
$(function() {
var new_element = document.createElement('textarea');
new_element.setAttribute("id", "Ghost_html_ori");
new_element.style.display = 'none';
new_element.value = document.documentElement.innerHTML
document.body.appendChild(new_element)
});
```

(b) *ContentScript.js*

Fig. 7 Manifest and content script of the *initial extension*

```
"content_scripts": [{
"matches": ["<all_urls>"],
"js": ["content.js"],
"run_at": "document_end"
}],
```

(a) *Manifest.json*

```
$(function(){
window.setTimeout(function(){
var html = document.getElementById('Ghost_html_ori').value;
//Compare the htmls here and add all changes from extensions
document.body.removeChild(html);
},0);
});
```

(b) *ContentScript.js*

Fig. 8 Manifest and content script of the *final extension*

Fig. 8b). Our aim was to extract the effects that the execution of the extensions generate to the DOM and thus the knowledge the extensions have. Additionally, recall that the order in which extension are executed can be easily modified by using existing proposals [33]. Our proof-of-concept implementation is used for experimental purpose only and should not be considered as a final solution for deployment.

5 Experimental results

We next discuss the experimental results obtained after evaluating our solution. We have focused on the first proposed solution where a *<textarea>* is used in the client side instead of using an external server. We have studied the following performance indicators according to [16] and the W3C consortium [34]: (1) memory consumption; (2) time needed to parse the HTML; (3) when the *onLoad* event is fired (many JavaScript files wait for this event); (4) the *processing time* which means that all resources have been loaded (DOM is completed i.e., the loading spinner has stopped spinning); and (5) a final test to show the total time that Chrome needs to generate the *onLoad* event, i.e., the page is ready. All the experiments but the memory consumption were carried out accessing the Alexa's Top 30 web sites and averaging the results over 50 runs. Additionally, in order to measure all the time-based metrics, we have used the DevTools profiling tools provided by the browser.

Our extension-based solution inserts a middle monitor extension between every two original extensions, plus the initial and the final ones. Thus, the number of total extensions is $2n + 1$ (n original extensions plus $n + 1$ added monitor extensions, including the initial and final ones). In order to test what impact these additions have on both Chrome's performance and the user experience, we have installed a set of original extensions in a MacBook Air with 2.2 GHz Intel Core i7 CPU and 8 Gb of RAM. The Chrome version where all test have been run is 60.0.3112.78 (Build official) (64 bits). We used the 10 most downloaded browser extensions from the Chrome Web Store, since according to [5], the average number of installed extensions per user is 5.

All figures related to the monitor extensions depend on the number of original extensions installed in the browser ($2n + 1$). In our experiments, the number of extensions is related to the original extensions installed. This number varies if the experiment is performed by using the original extensions or our proposed solution. For instance, when we say that with 5 extensions it takes 1.3 seconds to load all the scripts of a entire page, it means that in reality there are 11 extensions installed in the browser: 5 original extensions,

Table 2 RAM consumption

#Extensions	Originally (Kb)	Solution (Kb)	#Extensions	Originally (Kb)	Solution (Kb)
2	217.9	255.0	7	420.9	513.1
3	331.6	379.7	8	492.0	595.2
4	348.7	407.9	9	504.3	618.5
5	374.1	444.2	10	527.1	652.3
6	392.1	473.3			

plus 4 middle extensions, plus 1 final extension, plus 1 initial extension. On the contrary, 5 extensions on the *original extension* experiment means that only the 5 original extensions are installed in the browser. Additionally, for all the experiments we have measured times without the browser's cache and by launching one new, fresh instance per experiment, i.e., we have closed and opened Google Chrome each time we added a new browser extension to measure RAM consumption and user experience times.

5.1 RAM consumption

To measure memory consumption, we have used the developer tools provided by Chrome. Table 2 shows the impact on the browser performance in terms of RAM consumed in KB. We have isolated the execution of the original extensions and the monitor extensions in order to show that the impact of our proposed solution is almost negligible in comparison with the performance of the original extensions. Moreover, both the initial and the final extension consume 13 KB of RAM each, whereas our monitor extensions consume 11 KB of RAM on average. These extensions differ considerably from extensions such as AVG Web TuneUp, AdBlock or Ad Block Plus, which consume 27.6 KB, 190.3 KB, and 11.3 KB of RAM on average, respectively. Note that the size of such extensions depends in fact on the content of the web page. For instance, a page containing a substantial amount of advertisements would make Ad Block to consume much more memory. From the results, we can conclude that the impact of our solution is approximately linear in the number of extensions. More concretely, our proposed solution decreases performance by a factor of 1.15 per installed extension in terms of RAM.

5.2 Impact on user experience

Figure 9a shows the time that Chrome needs to parse the HTML. At this point, Chrome has already parsed the entire HTML file and creates the DOM. We can observe that, in the worst case (for 10 original extensions), our solution introduces a delay of 5000 ms. Similarly, Fig. 9b shows the time needed for the browser to fire the event *onLoad*. This event is critical because most of the extensions, jquery, and all libraries based on jquery wait for that event to be executed. From the results, it is remarkable that the inclusion of our solution does not introduce undesired delays in the execution of this event in comparison to the default behavior.

The processing time measures when all resources have been loaded. Currently, the way the user knows when a given page has been totally loaded is when the spinner at the core of most browsers stops spinning. There are a bunch of external parameters that directly affect this time, such as the network overhead or the number of resources previously stored in the cache, among many others. All in all, we can conclude that the number of installed extensions has a potentially large impact on performance and, therefore, in the processing time as it is depicted in Fig. 9c. This, however, is only relatively critical as the average number of installed extensions is very low for most users.

Finally, Fig. 9d shows the time needed to load the whole web page. The total time is calculated from the sum of processing and load times. This plot, together with the ones discussed before, shows that content scripts of browser extensions are not totally decoupled from the rendering process and, therefore, they directly impact performance and user experience.

From the above results, we can confirm that the bottleneck of the browser extensions is, in general, the processing time, i.e., all HTML resources that web pages have. One possible consequence of this is the non-monotonic behavior that can be seen in all subfigures of Fig. 9. However, despite this non-monotonic behavior, we can see how our proposed solution adds some delay (in average) with respect to the default browser.

In general, our solution increases very moderately the amount of time Chrome needs to render the content. This problem might be solved by modifying the browser's source code.

6 Advantages, properties and limitations of our approach

The primary aim of this paper is to demonstrate the feasibility of a lightweight solution to avoid extensions getting sensitive information about the user due to the order execution, while still (substantially) preserving the main functionality of the extensions. We next address a number of natural ques-

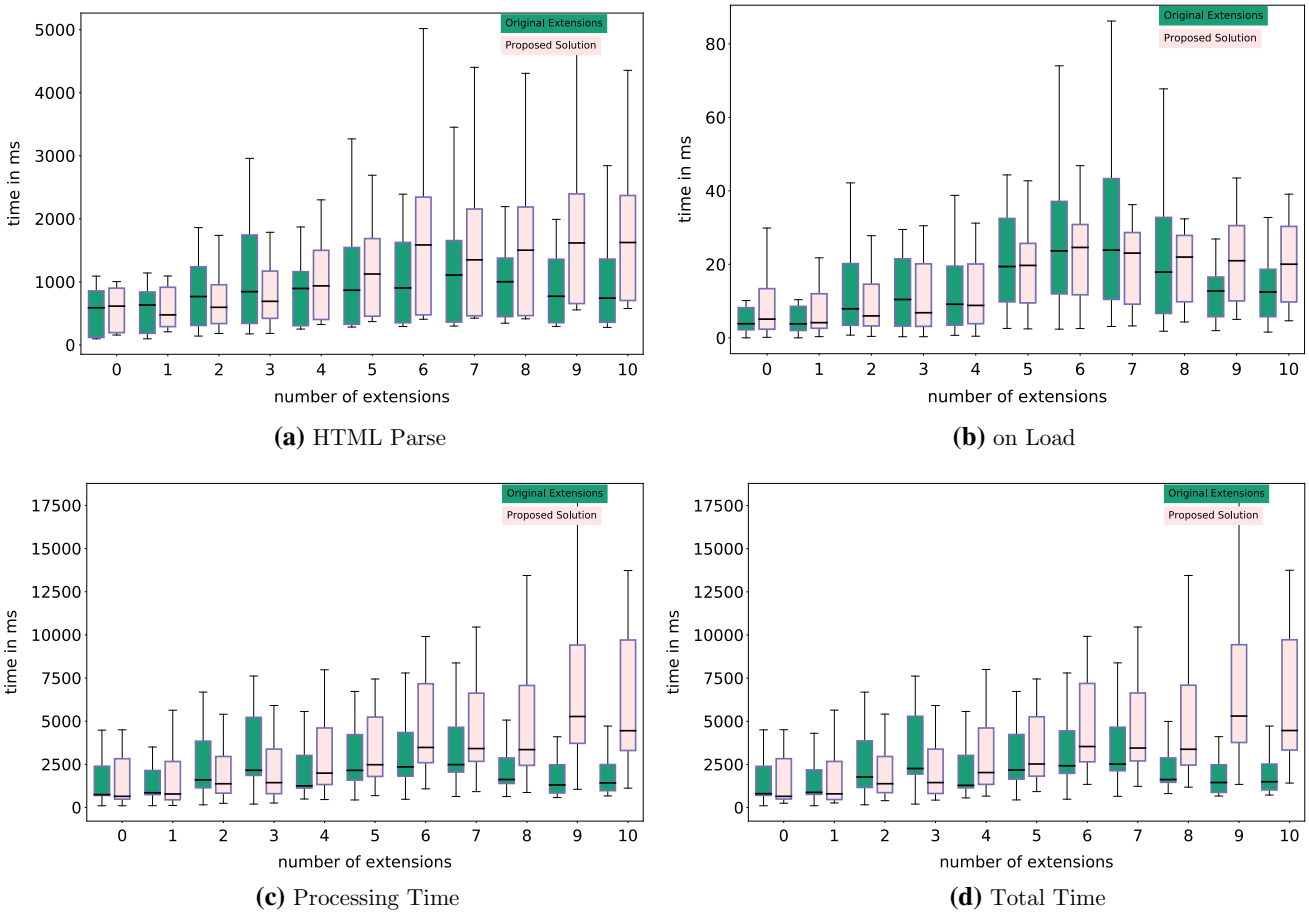

Fig. 9 Evaluation of our proposal according to W3C parameters

tions related to its main properties. In particular, we show that our approach does have some intruding effects, that in fact mitigates both *usual* and *strong* attacks, and that our approach is robust against *strong* attacks under certain reasonable assumptions.

6.1 How intrusive is our solution?

That is, how much of the extensions' (good and expected) behavior do we modify while achieving our goal of preserving privacy? Our solution always preserves the behavior of the original browser execution model (i.e., the final output with or without our solution is exactly the same). In some sense, we do want to make sure that the order of execution is irrelevant with respect to the knowledge the extensions *should* get (i.e., not accessing sensitive information they are not allowed to as an effect of this information being *passed* by other extensions), but we also know that the outcome of the executions of such extensions might be modified by our approach eventually modifying some of the expected output. Let us consider an example showing the possible effects of our solution. Let E_i be an extension that changes the DOM's background color to black. (Let us assume the original color was white and that there is text both in black and blue.) Let us consider that a later extension in the execution pipeline, E_j $(1 \leq, i < j \leq n)$ changes the background color to white. It is clear that in the current order, the final outcome is that the DOM's background color is white and all the text is readable. That being said, it is clear that in case the extensions were executed in different order (first E_j and then E_i) the outcome will be very different: not only the background will be black (instead of white), which by itself does not seem to be a big deal, but more importantly there will be some text not visible to the user. This not only affects the usability of the DOM (the black background will hide all the black text so the user will not be able to see it), but may introduce some security issues. (The hidden text might be clicked accidentally producing undesired effects.)

This is, however, an inherent behavior of the browser and our proposed solution does not modify the default behavior of the browser, i.e., a given HTML content looks the same with a set of extensions enabled and with the same set and the proposed solution. Moreover, JavaScript periodical tasks such as *setInterval(callback, delay)* are not covered in detail with our

solution. This method automatically enqueues the function defined in the callback in the task queue. For instance, if the extension A uses this method to get all password fields from the page the user is visiting each 2 seconds. This is a completely different scenario because the execution of this task cannot be controlled through JavaScript code alone.

6.2 Does our solution indeed mitigate possible attacks?

According to the definitions given in Sect. 3, the knowledge of an extension executed in position j ($1 < j \leq n$) is the same knowledge as the previous extension (E_{j-1}) in the pipeline plus the actions that E_{j-1} performs over the DOM ($\text{DOM}_{j-1} \cup \text{DOM}_j$). On the contrary, when we measure the knowledge of an extension with our solution, it is thus decreased to DOM_0 (given that our solution only passes the original DOM to each extension). Our solution also mitigates a *strong* attacker by limiting what she gets to know in the same way as for the *usual* attacker: Our interleaving guarantees that a *strong* attacker only gets to know the original DOM. The reduction in knowledge is of course more significant than in the *usual* attacker (Sect. 3).

6.3 How robust is the approach?

That is, can we guarantee that a *strong* attacker cannot bypass our solution? One may think that a *strong* attacker could attack our solution by interleaving extensions between our monitor extensions (before and after) thus bypassing our protection in order to get access to the effects of the installed extensions before being modified by our monitor extensions and then restoring it after our modification. To do so, the attacker must create an extension with the *management* privileges. That, however, would only be possible if the user explicitly grants that permission to the attacker. The best we can do is then to show a warning message to the user as soon as we detect the presence of such malicious extension and rely on that the user blocks the attacker. If the user grants the permission, we are thus vulnerable to the attack. In order for our proposed solution to be able to detect the presence of such attacks, our extension would need to have *management* privileges. This could only be granted by the user at installation time.

Proposition 1 *Our extension-based solution is robust against* strong *attackers under the assumption that our (initial, middle and final) monitor extensions are given management privileges and that the user does not explicitly give management privileges to the attacker.*

In case the user (accidentally or consciously) gives the needed privileges for a *strong* attacker to install his extensions, our solution would be able to detect that and communicate it to the user. Indeed, a *strong* attacker would need to install $n + 1$ extensions interleaved between any two extensions, and our monitor extensions would be able to detect that. So, we have a way to detect this issue, notify it, and ask the user to uninstall the extension. Besides, by identifying this we would be able to keep a black list of malicious extensions.

6.4 Extension-based or part of Chromium's source code?

Most of the aforementioned questions would be solved by modifying the Chromium's source code. Note that an attacker might insert as many extensions as desired and could even alter the execution order. By modifying the source code, all extensions receive a fresh copy of the original HTML and, thus, no-one will learn about the actions executed by other extensions. This solution uses a similar approach but requires modifying (and recompiling) Chromium's core to achieve isolated execution. At a logical level, it works exactly the same as the general solution depicted in Fig. 6. The same original DOM is passed on to each browser extension, but we do not allow automatic synchronization between isolated worlds (see Fig. 4). Instead, a final module takes all the changes performed by the official extensions and adds them to the final HTML. However, this does not mean that page rendering is necessarily delayed until the last extension is done. In fact, with our solution directly implemented on the source code, the rendering time of the initial DOM will remain exactly as it is right now, i.e., the webpage is rendered to the user as soon as it is received from the server. However, the DOM the user is reading is automatically updated whenever an extension finishes its execution. Note that extensions do not have the last updated copy of the content but the one provided by the server.

Nevertheless, with this last approach there is an inherent problem: The order still matters. Let us use the same example described before, there are (at least) two browser extensions that modify the same property of the CSS—the background color. In this solution, we decided to keep the same order as if it were executed in the original pipeline so the user will not find any differences with our proposal and the non-modified browser. This is just a coding decision and could be easily modified in our *apply* method.

Note that the main limitation that our proof-of-concept based on extensions has is the ability of attackers to modify the order of the monitor extensions and thus bypass our solution. This issue is solved by directly deploying our solution as part of the source code of Chromium, making thus impossible to bypass.

Finally, we demonstrate that our extension-based solution does not increase significantly the time needed to render a web page (see Sect. 5).

7 Related work

Security and privacy aspects of browsers have received much attention in recent years [24]. Comparatively, the number of research papers published on browser extensions is negligible. This might be attributed to the lack of clarity of the actual security model of extensions in most browsers and how they work in practice. For instance, Bauer et al. in [4] presented a model of Chrome where each content script runs in the same process as the web page into which it is injected. However, this is not sound due to the existence of isolated worlds where each content script is executed in dedicated sandboxes and the modifications of the DOM are automatically synchronized through the C++ shared objects in Blink. To the best of our knowledge, our work is the first paper that discusses attacks induced by the lack of isolation among extensions (in particular, exploiting their relative execution order) and proposes a countermeasure for it.

As it has been recently demonstrated, users are not aware of the privacy leakages and the consequences that extensions can generate [14,28]. An experiment was conducted with 24 people to check whether they were aware of privacy issues while they were using the browser. To do so, they used browser extensions to alert users when some privacy issues were on-going, but they conclude that users do not know the real implications of those privacy leakages. In our proposal, we minimize the effect of the order that the browser includes by default and thus decrease the sensitive information that other extensions might acquire.

In [17], Jagpal et al. explored the problem of detecting malicious extensions. They show how by performing a static analysis over a set of 45 Gb of extensions within 5 days they are able to catch 70% of the malicious ones. However, their analysis does not consider the fact that the execution order may cause privacy leakages. Several other works have focused on static analysis to classify extensions as benign or suspicious [2,13,38], while others have explored dynamic analysis techniques to monitor their execution [9,20,22,30,36], or a combination of both [39].

Contrarily to other proposals, this work does not modify the browser core, while the performance remains at a reasonable level. To cite a recent example—even though in this case the authors address a different problem—Arshad et al. propose in [1] a modification of the Chromium's core to protect users from malicious code while browsing. Their proposed solution generates a 12.2% overhead in browsing time on average, though they claim that the trade-off between security and performance is acceptable for many users. Bauer et al. [3] proposed a taint analysis model that also modifies the browser's source code to track components that access sensitive information. The work shows some promising results, though at a performance overhead of 55%. In addition, they only fired one javascript event (*onload*) out of more than 279 existing events. Additionally, their security assumptions are at least questionable due to two main reasons: (1) They use the *manifest.json* file to assign labels for both the content scripts and the background files and (2) *Content Security Policies (CSPs)* are used to assign the permissions the HTML resources have. It is worth remembering that according to [29] only 1.17% of the Alexa Top 1 Million rank use CSP. For those reasons, they had to manually include CSPs and permissions to test their experiments.

8 Conclusions

In this paper, we have discussed one important security and privacy implication of Chromium's extension model: The effects of one extension are visible to others in the execution pipeline. This can be exploited by a malicious extension that can, for example, get access to sensitive information or manipulate the DOM elements introduced by other extensions. We call this a *usual* attacker, in contrast to a *strong* attacker who has access to the effect of each single extension in the execution pipeline. A strong attacker may, in particular, install itself as the last extension in the pipeline and produce many copies interleaving itself in between all other extensions. In this way, it could be possible to get to know what all other extensions are doing and exploit this fact. We have shown examples on how to perform both a usual and a strong attack.

We have provided a proof-of-concept to address this problem which relies on replacing the pipeline execution model by one in which each extension executes in isolation and then combine all individual effects to create the final DOM. Our implementation does this through a set of *monitor extensions*. As a first approach, we decide to take the effect of the last extension in the pipeline. We could, however, easily provide a solution based on user intervention (asking the user to decide) or to apply a different policy (choose the first one, or non-deterministically). A more refined way to do so is left as future work (e.g., one could gather information on how harmful the effects are, rank them and choose the less harmful using machine learning algorithms). We have open sourced the proof-of-concept and we are close to having a fully operational version based on the modification of Chromium's source code, which will be also open sourced.

Acknowledgements Open access funding provided by University of Gothenburg.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Arshad, S., Kharraz, A., Robertson, W.: Include me out: In-browser detection of malicious third-party content inclusions. In: J. Grossklags, B. Preneel (eds.) FC, pp. 441–459 (2017)
2. Bandhakavi, S., Tiku, N., Pittman, W., King, S.T., Madhusudan, P., Winslett, M.: Vetting browser extensions for security vulnerabilities with vex. Commun. ACM **54**(9), 91–99 (2011)
3. Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., Tian, Y.: Run-time monitoring and formal analysis of information flows in chromium. In: NDSS (2015)
4. Bauer, L., Cai, S., Jia, L., Passaro, T., Tian, Y.: Analyzing the dangers posed by chrome extensions. In: CNS,, pp. 184–192 (2014)
5. ons Blog, M.A.: How many firefox users have add-ons installed? https://blog.mozilla.org/addons/2011/06/21/firefox-4-add-on-users/ (2018)
6. Chrome: External Extensions. https://developer.chrome.com/extensions/external_extensions (2018)
7. Committee, W.D.T.: W3C DOM4. https://www.w3.org/TR/domcore/ (2018)
8. Developer., J.A.G.C.: Tasks, microtasks, queues and schedules. https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/ (2018)
9. Dhawan, M., Ganapathy, V.: Analyzing information flow in javascript-based browser extensions. In: ACSAC, pp. 382–391 (2009)
10. difflib: difflib. https://docs.python.org/2/library/difflib.html (2018)
11. Extensity: Extensity. https://chrome.google.com/webstore/detail/extensity/jjmflmamggggndanpgfnpelongoepncg (2018)
12. Google: Design of V8 bindings. https://chromium.googlesource.com/chromium/src/third_party/+/master/WebKit/Source/bindings/core/v8/V8BindingDesign.md#World (2018)
13. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: S&P, pp. 115–130 (2011).
14. Gulyas, G.G., Some, D.F., Bielova, N., Castelluccia, C.: To extend or not to extend: On the uniqueness of browser extensions and web logins. In: Proceedings of the 2018 Workshop on Privacy in the Electronic Society, WPES'18, pp. 14–27 (2018)
15. HMAC: Chromium Secure Preferences. http://www.adlice.com/google-chrome-secure-preferences/ (2018)
16. Ilya Grigorik: Measuring the Critical Rendering Path. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/measure-crp (2018)
17. Jagpal, N., Dingle, E., Gravel, J.P., Mavrommatis, P., Provos, N., Rajab, M.A., Thomas, K.: Trends and lessons from three years fighting malicious extensions. In: USENIX, pp. 579–593 (2015)
18. JavaScript: Bubbling and capturing. https://javascript.info/bubbling-and-capturing (2018)
19. jsdiff: jsdiff. https://github.com/kpdecker/jsdiff (2018)
20. Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., Paxson, V.: Hulk: Eliciting malicious behavior in browser extensions. In: USENIX, pp. 641–654. USENIX Association, San Diego, CA (2014)
21. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: A survey (2019)
22. Onarlioglu, K., Buyukkayhan, A.S., Robertson, W., Kirda, E.: Sentinel: Securing legacy firefox extensions. Computers & Security **49**, 147—161 (2015)
23. Projects, C.: Chromium. https://www.chromium.org (2018)
24. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The ghost in the browser analysis of web-based malware. In: HotBots, pp. 4–4. USENIX Association, Berkeley, CA, USA (2007)
25. Rogowski, R., Morton, M., Li, F., Monrose, F., Snow, K.Z., Polychronakis, M.: Revisiting browser security in the modern era: New data-only attacks and defenses. In: EuroS&P, pp. 366–381 (2017)
26. Saini, A., Gaur, M.S., Laxmi, V., Conti, M.: Colluding browser extension attack on user privacy and its implication for web browsers. Computers & Security **63**, 14—28 (2016)
27. Saini, A., Gaur, M.S., Laxmi, V., Conti, M.: You click, i steal: analyzing and detecting click hijacking attacks in web pages. International Journal of Information Security (2018)
28. Schaub, F., Marella, A., Kalvani, P., Ur, B., Pan, C., Forney, E., Cranor, L.F.: Watching them watching me: Browser extensions' impact on user privacy awareness and concern. In: NDSS (2016)
29. Scott Helme: Alexa Top 1 Million Analysis. https://scotthelme.co.uk/alexa-top-1-million-analysis-feb-2017/ (2018)
30. Shahriar, H., Weldemariam, K., Zulkernine, M., Lutellier, T.: Effective detection of vulnerable and malicious browser extensions. Computers & Security **47**, 66—84 (2014). Trust in Cyber, Physical and Social Computing
31. Sjösten, A., Van Acker, S., Picazo-Sanchez, P., Sabelfeld, A.: Latex gloves: Protecting browser extensions from probing and revelation attacks. Power p. 57 (2018)
32. Software, T.A.: Chrome Secure Preferences Modification. https://cs.chromium.org/chromium/src/rlz/lib/machine_id.cc?sq=package:chromium (2018)
33. Trickel, E., Starov, O., Kapravelos, A., Nikiforakis, N., Doupé, A.: Everyone is different: Client-side diversification for defending against extension fingerprinting. In: USENIX, pp. 1679–1696 (2019)
34. W3C: Navigation timing. https://www.w3.org/TR/navigation-timing/ (2018)
35. w3schools: Browser Statistics. https://www.w3schools.com/browsers/ (2018)
36. Wang, L., Xiang, J., Jing, J., Zhang, L.: Towards fine-grained access control on browser extensions. In: M.D. Ryan, B. Smyth, G. Wang (eds.) ISPEC, pp. 158–169 (2012)
37. Xing, X., Meng, W., Lee, B., Weinsberg, U., Sheth, A., Perdisci, R., Lee, W.: Understanding malvertising through ad-injecting browser extensions. In: WWW, pp. 1286–1295 (2015)
38. Zhao, B., Liu, P.: Behavior decomposition: Aspect-level browser extension clustering and its security implications. In: S.J. Stolfo, A. Stavrou, C.V. Wright (eds.) RAID, pp. 244–264 (2013)
39. Zhao, R., Yue, C., Yi, Q.: Automatic detection of information leakage vulnerabilities in browser extensions. In: WWW, pp. 1384–1394 (2015)