Imperial College London

Department of Computing

MEng Individual Project

Intercepting Suspicious Chrome Extension Actions

Author:
Michael Cypher

Supervisor:
Dr. Peter Pietzuch

Second Marker:
Dr. Peter McBrien

June 19, 2017

Abstract

Browser users have increasingly fallen victim to a variety of attacks carried out by malicious browser extensions. These extensions often have powerful privileges and can execute within the context of sensitive web-pages. Popular attacks include hijacking users' social media accounts, injecting or replacing advertisements and tracking users. Previous work to detect malice in extensions has not succeeded in providing adequate security guarantees about extensions running within browsers.

We present a novel extension security model that categorises certain actions as suspicious and prompts users to allow or prevent suspicious operations when executed by extensions. We propose minimal changes to the Chrome browser that implement this model and that provide guarantees that malicious extension actions cannot evade detection. In order to not inconvenience users, we build features that reduce the quantity of decisions that they are required to make.

We extensively evaluate our modified browser with regards to security guarantees, user interface, user understanding and performance overhead. Results demonstrate our browser's ability to intercept and stop malicious extension operations as they occur. However, our evaluation also suggests that users struggle with this responsibility and we find evidence of significant browser performance overheads.

Acknowledgements

I would like to thank:

- Dr. Peter Pietzuch, my supervisor, for proposing this fascinating project and for providing me with valuable guidance, support and feedback throughout the entirety of this project.
- Dr. Peter McBrien, my second marker, for his useful advice early on.
- PhistucK, proberge, jbroman and the overarching Chromium community for their understanding and assistance without which I would not have successfully achieved key project goals.
- my friends and family for validating ideas, reading over initial drafts of this report and for their continued support during my time at Imperial.

Contents

Introduction

Web browsers are one of the most popular applications. Anecdotally, this is because it is easy for users of limited technological expertise to use browsers and because developing and deploying web applications is far easier than desktop or mobile applications. All major browsers currently support an extension system whereby third parties can add functionality or modify behaviour of the browser or web-pages. Chrome, Firefox, Opera[1], and Safari support JavaScript-based browser extensions, while Internet Explorer (IE) supports binary add-ons.

Extensions provide additional browser functionality which can include blocking advertisements, managing passwords, translating pages and providing dictionaries. In addition, extensions are often granted the privilege to run within sensitive browser tabs that render banking, social media and medical web-pages, as well as email and communication applications. Extension scripts abide by different security models than standard web-page scripts and are frequently granted powerful privileges that let them access elements on a page, use the network to send or request arbitrary information and access the browser's history and tabs.

Chrome is often regarded as the most secure browser and is also, perhaps consequently, the most popular web browser [1]. However, the Chrome extension architecture and security model was designed to defend users from *benign-but-buggy* extensions that are not designed by security-conscious developers [2]. Unfortunately, Chrome's permission and security models do not adequately protect users from malicious extensions that potentially harm and act against the requirements of users. As a result, serious vulnerabilities exist that allow malicious extensions to hijack social media accounts, inject advertisements and track users, or that allow extensions to bypass the permission system by providing useful functionality.

Previous work has suggested improvements to Firefox and Chrome's security model and permission system [2, 3, 4]. However, large scale changes that revoke or modify permission models can have the adverse effect of breaking many existing benign extensions. Other research has automatically and dynamically analysed Chrome extensions and IE add-ons to determine malice [5, 6, 7]. Yet detection by these tools can be easily evaded by sophisticated attackers who can fingerprint analysis environments and therefore do not provide strong guarantees about browser security.

1.1 Objectives

Malicious extensions can harm users. Sophisticated attackers have previously been able to evade detection and successfully carry out attacks on victims. The aim of this project is to solve this problem and guarantee the protection of Chrome users from attacks realized by malicious extensions.

[1] Both Firefox and Opera extensions are called add-ons.

Web-pages can be malicious themselves and much work has been done to extinguish them, but for the purpose of this project, we assume that they are not. When considering approaches and developing different solutions, our goals, in order of priority, are to:

1. Protect users from Facebook hijacking, ad injection, user tracking and other malicious extension related threats.
2. Build a tool or extend the Chrome browser to protect users in a way that cannot be evaded by sophisticated attackers.
3. Break minimal benign Chrome extensions.
4. Break minimal web-pages and not incur a significant browsing performance overhead.

1.2 Contributions

In this project, we:

1. Introduce a novel extension permission model whereby users are prompted to allow or prevent certain suspicious extension actions at run-time. We evaluate and classify a set of Web API [8] operations as suspicious if executed by extensions. We build a tool that allows security-conscious users to configure this set themselves (§3).
2. Suggest and implement a mechanism to detect if execution within Chrome's browser rendering engine was triggered by an extension script or web-page script. If the former, our proposed changes correctly determine the specific extension that is executing within the engine. We incorporate an existing Chromium patch to guarantee extensions cannot evade detection (§4).
3. Use this mechanism to intercept suspicious browser extension actions as they occur and ask browser users if they want to allow or prevent the actions via a pop-up dialog (§5).
4. Reduce the number of pop-up dialogs displayed to users by suppressing benign events, filtering out operations on elements not attached to the web-page, remembering the users decision to allow or prevent certain operations and marking specific elements as sensitive (§6).
5. Evaluate our modified Chrome browser against existing extensions on the Chrome Web Store and against a contrived malicious extension with regards to security guarantees. We assess our browser's user interface and the impact of our changes by surveying users and measuring suspicious actions in popular benign extensions. We finally compare the performance overhead of our modified Chrome browser and determine the feasibility of merging the changes into the Chromium project (§7).

Background

This chapter first gives a high-level overview of the Chrome browser, including its multi-process architecture, rendering engine and compositor thread architecture.

A detailed overview of Chrome extensions is provided which describes the extension architecture and security model, and the Content Security Policy (CSP) that the extension system implements. Chrome's extension permission and capability system, including some limitations, is presented. We discuss current prevalent threats in the Chrome extension ecosystem.

Previous work is investigated that analyses extensions dynamically to determine malice and we study related work that involves tracking sensitive data through PHP applications, Internet Explorer (IE) binary add-ons and across cloud service providers.

2.1 Chrome Browser

We decided to focus on the Chrome browser due to its popularity [1], open-source license[1] [9] and because it is in compliance with various web standards [10]. Furthermore, Chrome also has the most comprehensive extension security system - only allowing installation via the Chrome Web Store (CWS), requiring privileges to be granted by users, enforcing privilege separation and more [2, 5]. Many of the issues and suggested solutions mentioned in this report may also naturally extend to other popular browsers.

As this project concerns itself with protecting Chrome browser users from malicious extensions, we start by researching and providing a high-level overview of the Chrome browser. The information conveyed in this section is necessary to understand the challenges we faced and solutions we proposed through this project.

2.1.1 Multi-Process Architecture

In order to protect against render-engine bugs, crashes and security vulnerabilities, Chrome implements a multi-process architecture [11].

Chrome runs a separate *renderer* process for each web-page, including tabs and extension background pages (§2.2.1). By rendering and executing each page and its scripts within an individual process, a crash in a renderer process is unlikely to affect the browser or other renderer processes. The memory of each process is also separated and protected by the operating systems on which Chrome executes, preventing compromised renderer processes from reading sensitive data across web-pages. Chrome also sand-boxes

[1] The majority of the Chrome browser's source code is released as the open-source Chromium browser. Although we modify the Chromium browser, for the sake of simplicity, we refer to it as the Chrome browser throughout this report.

each renderer process, restricting access to the network, file-system and devices. These restrictions limit an attacker's potential, even if they control a renderer process. Chrome's Site Isolation project aims to continue to strengthen security by further isolating web components [12].

In addition to the security benefits, Chrome provides each renderer process with a priority; hidden windows and tabs are assigned lower priorities. When memory is sparse, Chrome first swaps memory out from low-priority processes to disk in order to keep focused tabs responsive.

Chrome also runs a main *browser* process responsible for managing and communicating with each renderer process and displaying the browser user interface. This includes the navigation bar and pop-dialogs display during run-time.

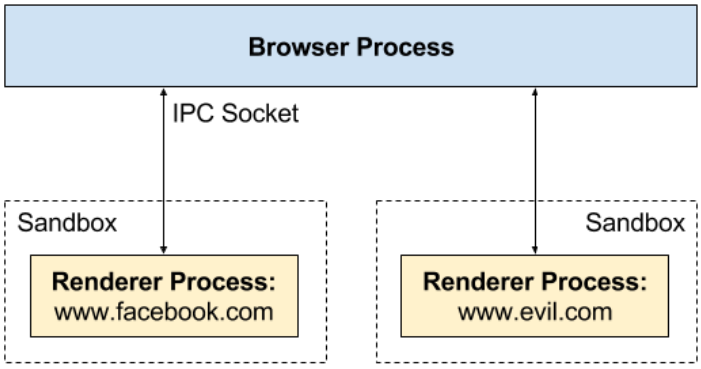

Figure 2.1: Each web-page is rendered within its own sand-boxed renderer process which communicates with its parent browser process.

Inter-Process Communication

Renderer processes and the browser process communicate via a named pipe created using the `socketpair` system call on Linux and OS X operating systems [13]. In order to not block the user interface, messages sent and received in the browser process are processed within a dedicated I/O thread and subsequently proxied to the main thread using a `ChannelProxy`. Each renderer process also has a main thread which manages communication between the browser process and itself, and another thread for rendering (§2.1.3).

To communicate between processes, developers define a message template, including serialisable input and output types, and whether the sender should wait for a response or not. Each message template registers a method in the destination process to handle incoming messages and send back appropriate responses.

2.1.2 Blink Rendering Engine

Each renderer process interprets and lays out web-pages using Chrome's open-source layout engine, Blink. The Blink engine was forked from WebKit's WebCore component, used in the Safari browser, in order to accommodate large scale architectural changes [14].

As well as rendering pages, Blink provides a standardised web interface to scripts via JavaScript objects and functions. This interface, mostly specified using a Web Interface Definition Language (IDL) [15], provides a wide variety of functionality - everything from sending network requests to showing user prompts.

The interface is largely specified by the World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATAG). In order to conform to these specifications, the Chromium team has been known to import and execute a suit of regression tests [16].

2.1.3 Compositor Thread Architecture

The main thread of the renderer process routinely blocks while network requests are sent, styles are recalculated and the garbage collector is triggered. To prevent jarred scrolling or animations, the renderer process follows a compositor thread architecture [17].

The process contains two important threads: the main thread stores a model of what Blink wants to draw and the *compositor* thread[2] which paints version-ed web-page snapshots to the screen. The render tree of each web-page is represented as scroll-able layers which each have a size, content and transformation. A layer tree indirectly[3] represents each web-page and every layer on the main thread (`LayerChromium`) maps directly to a layer on the compositor thread (`CCLayerImpl`). To facilitate smooth scrolling and animations, input events and animations are performed on layers in the compositor thread without needing to consult the potentially blocked main thread.

Whenever changes are made to the page within the main thread, it is marked as "dirty" and the compositor thread is alerted of inconsistencies between the two layer trees. Every so often, a scheduler executing on the compositor thread requests the main thread blocks while it commits changes to the compositor.

2.2 Chrome Extensions

Extensions allow third parties to add additional functionality to web-pages and the Chrome browser without modifying Chrome's native source code. They are essentially bundles of files that include: JavaScript, HTML, CSS, videos, images and other assets. Extensions can be viewed as basic web-pages that have the ability to use all of the Web APIs [8] the Chrome browser provides to regular web-pages.

Most extensions inject JavaScript, in the form of content scripts, into web-pages when they load. Using content scripts or cross-origin network requests, extensions can interact with web-pages and remote servers respectively. They can also interact with other components of the browser such as tabs, bookmarks and browsing history.

All extensions must have a `manifest.json` file that specifies meta-data about an extension (such as version and author), the extension's structure (such as its content scripts, browser or page actions) and perhaps most importantly, its permissions and capabilities [19].

The Chrome extension ecosystem includes an *autoupdating* mechanism in order to quickly dispatch bug and security fixes, functionality and user interface (UI) improvements. Updates to extensions are automatically applied without user involvement. This is performed automatically if third parties publish their extensions through the Chrome Developer Dashboard or can be performed manually by specifying an `update_url` field in the `manifest.json` file. Every few hours, the browser checks whether any extension installed has pending updates by querying the CWS or the server specified by the `update_url` [20].

[2]The compositor thread is commonly referred to as the *impl thread* because it implements the compositor.
[3]The layer tree is a sparse representation of a render tree which in turn represents a page's DOM tree [18].

2.2.1 Architecture and Security Model

The Firefox browser has publicly suffered from vulnerabilities at the hands of popular but buggy extensions [21, 22]. Work was undertaken to analyse *buggy-but-benign* Firefox extensions that have the ability to introduce vulnerabilities in otherwise secure web-pages and to propose a new browser extension system [3]. Barth et al. proposed a system, adopted by Chrome's extension architecture and security model, that contains three core fundamentals: *least privilege*, *privilege separation* and *strong isolation*.

Least Privileges

The Chrome extension security model implements the principle of least privilege[4]. This attempts to ensure that an extension only has the privileges needed for its "legitimate purpose" and functionality. To implement this, the Chrome browser only grants the privileges explicitly requested in an extension's `manifest.json` file.

If a malicious web-page usurps a vulnerable extensions privileges, it may only have a small subset of those it needs to carry out its malicious intent. However, as extensions often request, and are granted by users, more privileges than required, this mechanism alone is not enough to guarantee an extension has the least privileges necessary. Despite this, Kapravelos et al. noted that benign and malicious extensions do not differ greatly in terms of permissions granted and that more permissions are not necessarily required to carry out attacks. Instead, according to Kapravelos et al., it is the use of the Chrome extension API that allows malicious attacks [6].

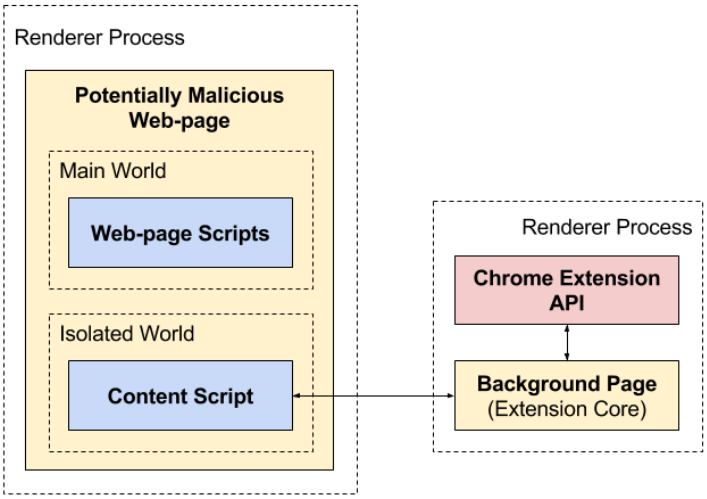

Figure 2.2: Extensions are split into multiple components such as content scripts and extension cores and are protected by process boundaries and the isolated worlds concept.

Privilege Separation

The idea behind privilege separation is to prevent malicious web-pages from gaining all of the capabilities of a *buggy-but-benign* extension [3]. As Chrome extensions can be granted quite powerful capabilities (such as the ability to view what tabs are open), unintentionally granting these capabilities to a malicious web-page can be quite serious.

[4]The principle of least privilege is also known as the principle of least authority or principle of minimal privilege.

To prevent against this, the extension platform requires that developers split their extensions into three main components: *content scripts*, *extension cores* (also called "background pages" [19]) and *native binaries*. The split (excluding native binaries) is shown in Figure 2.2.

Content scripts are injected and executed within a page's context when the page loads and have the ability to change a page's Document Object Model (DOM)[5]. Content scripts can directly interact with potentially malicious web-pages in which they have been inserted. To prevent the malicious page from usurping the content script's powerful extension APIs [24], they do not have any. Instead they must exchange messages and communicate with the background page.

Background pages are either *persistent*, which are always open and executing, or *event*-based, which are opened and closed as needed [19]. They have access to the extension API that has been granted and all standard Web APIs [8], but are insulated from untrusted web content. In order to communicate with untrusted content, the background page can either communicate with a content script or issue a web request.

Native binaries can run arbitrary code and access the browser's API. Although native binaries could contain vulnerabilities, they are not often used and undergo a manual security review before being permitted in the CWS [2]. As a result, we do not discuss native binaries further nor do we consider attacks that could be carried out by them.

One limitation of privilege separation is that the onus is on developers to properly insulate their extension core from their content script. If developers are not security conscious and do not assume that their content script could be compromised, they may unintentionally grant their content script more privileges than necessary. This risk can be exposed through many "mistakes" such as:

- Allowing the extension core to execute arbitrary code passed from a content script by using the `eval` function.
- Allowing the (indirect) use of an sensitive API function from a content script by passing the parameters to the extension core and executing it there.

Strong Isolation

The Chrome extension architecture attempts to isolate extension components from each other and from web-page components. It uses three methods to achieve this.

The first consists of assigning a security origin to each extension that remains the same across different versions. This allows the extension system to reuse the web's same-origin policy which in turn isolates extensions from web-pages and other extensions of different origins.

The second method consists of running each extension component - the content script, extension core and native binary - in a different native process as shown in Figure 2.2. The extension core and native binary both run in a dedicated process whereas content scripts run in the same process as the web-page they are injected into. This mechanism protects against browser errors since no JavaScript objects are leaked between processes and API methods cannot be accessed from content scripts.

[5]The DOM is a programming interface that models a HTML document as a tree structure where each node is an object representing a part of the document [23].

The final method consists of executing content scripts in their own *isolated world*. This mechanism is designed to protect content scripts from malicious websites[6]. Every content script injected into a page can access the DOM through different JavaScript objects[7] but cannot access variables or functions created by the page or other content scripts. Moreover, a web-page's scripts executing within the *main world* cannot access any of the variables or functions created by content scripts. If scripts executing within different page wish to communicate, they must do so via the shared DOM (using `window.postMessage` for example) [25].

Most updates to the JavaScript objects that represent the DOM are reflected across all objects, ensuring all scripts view the same page state. However, when a script modifies a non-standard JavaScript object (such as `document.foo`), the update is not reflected in any of the other (`document`) objects [3].

The *isolated worlds* concept also fortunately prevents conflicts (such as differing versions of the popular JQuery library) between a content scripts and page scripts. This is important as content scripts can be granted privileged functionality from their associated background page that should not be accessible from other scripts [25].

2.2.2 Content Security Policy

The Content Security Policy (CSP), developed and maintained by the W3C, is a mechanism to reduce the risk of content injection vulnerabilities such as cross-site scripting (XSS) [26]. Chrome's extension system implements the general CSP concept to increase security of extensions by preventing (by default) the ability to inject, include and execute remote JavaScript code [27] in an extension's privileged background page.

CSP restricts the privilege of extensions but allows developers to optionally relax the default behaviour [2]. In an attempt to mitigate the risk introduced by extensions relaxing the CSP, Chrome warns users downloading extensions of modifications that may make them vulnerable to attacks [4]. The CSP is an additional layer of security over the permission system (and the *least privilege* principle), but is not intended to be a replacement.

Although it helps protect users and benign-but-buggy extensions from malicious third-parties, it only adds small hurdles to sophisticated and determined malicious extension developers. It is important to note that CSP restricts what an extension's background page can do but not what an extension's content script can do. As a result, malicious extensions can still perform XSS by executing JavaScript from other pages (granted they have the permissions to communicate with the external page) within their content script and querying their background page's privileged browser APIs.

Dynamically Executing JavaScript

By default, `eval`, `window.setTimeout` and other functions that allow developers to execute JavaScript passed as a `String` are disabled in an extension's background page. This is done because the code `eval` executes can access everything in the extension's environment and can call powerful Chrome APIs, with the ability of weakening users' security [28].

[6] A malicious web-page could have been originally benign but modified by an active network attacker.
[7] There is a one-to-many relationship between the DOM and its representative JavaScript objects

For benign-but-buggy content scripts, this defends against malicious websites using common cross-site scripting (XSS) attacks [27]. It effectively makes it impossible for an extension to unknowingly execute malicious code provided by a malicious remote resource or web-page.

However, this policy is less effective against malicious extensions because it is still possible to use `eval` and other related functions by adding `'unsafe-eval'` to an extension's CSP. This is necessary because many popular frameworks still exist that require `eval`.

It is also possible to use `eval` in a safer environment called "sand-boxes". When running `eval` within a sand-box, the executed code can not access any of the extension's data or the extension's high-value APIs. To implement this, developers specify which extension files should be run within their own sand-box and when they are loaded, they are run from a different origin and denied access to the Chrome API.

Inline JavaScript

By default, inline JavaScript is also prohibited. This is because it exposes several attacks similar to those exposed by `eval` (instead of executing code using eval, one could just insert it into the HTML page within script tags). This policy ensures developers create extensions with a separation between HTML content and JavaScript behaviour.

Similarly to `eval`, it is possible to relax the policy by providing the base64-encoded hash of the source code in the extension's CSP [29]. For example, in order for an extension to inject the following element `<script>alert('Hello, world.');</script>` into a page's DOM, see the `manifest.json` file in Listing 1.

```
{
    "content_security_policy": "script-src `sha512-YWIz...OAo='",
}
```

Listing 1: A `manifest.json` file that allows an extension to inject a `script` element into a page's DOM.

Although this "relaxes" the policy regarding inline JavaScript, it still prevents many XSS attacks that are caused by dynamically executing unknown malicious code as the developer would need to know what the code injected is to create a hash.

Remote Scripts and Resources

Remote scripts and object resources cannot be loaded and only local scripts or objects, contained within an extension's package, are permitted. Similarly to the restrictions on `eval` and in-lining JavaScript, this prevents an extension executing unknown malicious code. In particular, it also defends against active network attackers who perform man-in-the-middle attacks and supply malicious resources instead of those requested. It is also possible to relax this policy by white-listing secure origins[8] in an extension's CSP [27] (see Listing 2).

[8]The following schemes: blob, file-system, HTTPS, and chrome-extension are considered "secure". Note, HTTP is not.

```
{
    "content_security_policy": "script-src `self' https://fb.com; object-src `self'"
}
```

Listing 2: A `manifest.json` file that allows remote scripts from `https://fb.com` to be loaded and injected into a page's DOM.

2.3 Threats

Although many attacks are possible when malicious extensions execute within a victim's browser, some are easily monetised and are therefore more common in the CWS. In this section, we discuss these attacks.

2.3.1 Facebook Hijacking

Facebook hijacking has been present since at least 2012 when initial reports emerged. This threat involves malicious extensions "hijacking" a victim's Facebook account and performing actions (such as sharing or "liking" posts, posting spam messages or sending them via private chat and befriending users) without the victim's consent.

According to analysis work by Google, this threat has been present in over 4,809 extensions between 2012 and 2015 [5]. Often these extensions provide actual functionality such as adding a "dislike" button, removing Facebook's "timeline" (to increase the victim's productivity) or changing the websites theme.

Attackers monetize this threat by posting spam messages, inflating "likes" or friend counts as has been observed on Twitter, a different social media platform. Brazilian attackers in control of malicious browser extensions have been discovered selling packages of 1,000, 10,000, 50,000, and 100,000 "likes", for $28, $248, $1,164, and $2,196 respectively [30].

2.3.2 Ad Injection

Ad injection is the second most popular threat present on the CWS, having been discovered in 3,496 different extensions between 2012 and 2015 [5]. This "attack" replaces or inserts advertisements on a web-page with custom ones provided by the extension. Although not explicitly prohibited by the CWS developer agreement, unless injected without user knowledge, it negatively impacts owners of a website by directing revenue away from them and towards extension developers [31].

Other similar attacks involve changing HTTP referrer headers and links on web-pages that point to e-commerce websites like Amazon. These services often have affiliate programs that reward third parties for sending them traffic. Instead of the web-page owner collecting that revenue through the affiliate program, malicious extensions often substitute their own affiliate IDs in and reap the rewards.

2.3.3 User Tracking

This threat involves malicious extensions inserting "tracking pixels" to track users and relay information (such as web-pages visited) to third parties without the consent of the users [5]. This can be used for advertising purposes. A similar threat leaks search queries to third parties which helps improve advertising results.

2.4 Permission Study

Various papers have studied the effect of Chrome's extension permission model to determine if permissions reduce the severity of extension core vulnerabilities (in the case of benign extensions) [2], indicate extension malice [6, 5] and ensure that users know the capabilities of a particular extension [4].

2.4.1 Permission Model Shortcomings

Chipman et al. identified areas where Chrome's permission model fails to inform users of an extension's capabilities [4] and they suggest changes to the model to prevent these discrepancies. They argue that a permission model that does not correctly convey the capabilities of an extension to a user, leaves room for exploitation, potentially by malicious extensions leveraging this discrepancy. Chipman et al. also built an educational extension that teaches users about permission model loopholes and to be more wary of granting unnecessary permissions to untrusted extensions.

Chrome Warnings

Similar to how Chrome notifies users when an extension modifies its CSP, Chrome also alerts users when extensions request *special* permissions that could pose security risks. Whenever a user tries to install an extension that has requested additional *special* permissions, via the extension's `manifest.json` file, a pop-up similar to that in Figure 2.3 appears.

However, not all permissions are considered to pose security risks and therefore display the warning "Requires no special permissions" to the user. Some of these non-*special* permissions include: `browsingData`, `cookies`, `sessions` and `webRequest`. To uninformed everyday extension users, there is no indication that the extension has capabilities to these resources. Chipman et al. explore potential attacks and exploits that could be leveraged using only non-*special* permissions.

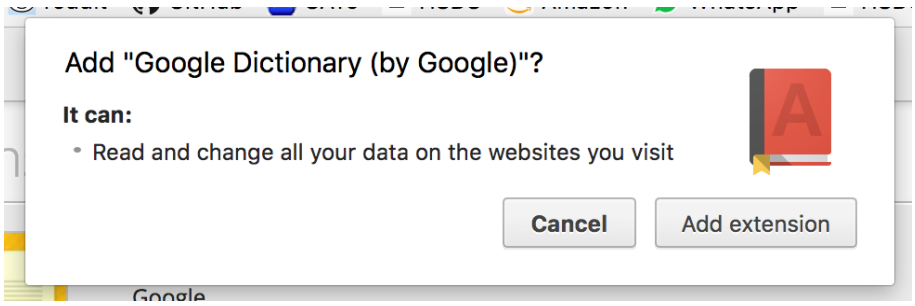

Figure 2.3: An example pop-up that Chrome displays, providing information of an extension's permissions, when installing the "Google Dictionary (by Google)" extension.

Threats

The extension built was able to exploit various attack vectors using only non-*special* permissions that include: preventing a computer from suspending by itself, spamming users with notifications, writing arbitrary text to the system's clipboard and performing a denial-of-service attack on the user's browser by repeatably closing all tabs. Most of these present a mild annoyance to victims and are difficult to

diagnose as the extension "Requires no special privileges". However, it is possible to potentially identify users with reasonable confidence using only non-*special* permissions.

2.5 Analysing Extensions

Previous work has been done that attempts to detect malicious behaviour in browser extensions. In contrast to other work that has suggested improvements to the Chrome extension security model (many of which have been incorporated), this section focuses on tools that use dynamic analysis to monitor execution and corresponding network traffic to determine if a given extension is suspicious or malicious. Some tools also use limited static analysis to check for signs of malice: code obfuscation, high permissions, and similarity to other malicious extensions.

2.5.1 Hulk: Dynamic Extension Analysis

Kapravelos et al. presented an early tool, Hulk [6], in the first broad study of malicious extensions, which attempts to detect malicious behaviour in Chrome extensions. Hulk labels extensions as benign, suspicious and malicious by applying various heuristics. The latter two indicate the presence of potentially or definitely harmful actions respectively.

The tool executes and monitors extensions, using several novel techniques that try and elicit malicious behaviour. Their main contributions are the techniques they present to elicit malicious behaviour (often trying to mask itself) from extensions.

URL Extraction

Most extensions execute within the contents of web-pages but many do not and cannot execute on web-pages of different origins. To trigger extensions' (potentially malicious) functionality, a URL or set of URLs must be carefully chosen and loaded. Hulk uses three sources to extract URLs from:

1. **The manifest file** of an extension specifies the URLs that its content scripts can be injected into and also specifies the extensions' cross-origin permissions. URLs are generated that match those in the `manifest.json` file. If the file specifies `<all_urls>` or many URLs (using wild-cards), it becomes difficult for Hulk to generate URLs where malicious behaviour is elicited.
2. **The source code** of an extension may contain static, hard-coded URLs. Hulk "visits" those URLs as well, in an attempt to elicit malicious behaviour that is only triggered on a small subset of websites.
3. **Popular websites** that have been previously targeted by malicious extensions. Kapravelos et al. continuously improved their list of popular websites as they discovered attacks on particular domains.

However, Hulk's URL extraction has various limitations. It does not consider that pages loaded from the same URL may be different due to a client's location or time. Web-pages that require users to sign in create difficulties. Hulk attempts to solve these by using pre-existing user credentials when visiting popular pages but this may miss malicious extensions targeting specific individuals.

HoneyPages

Some extensions execute malicious behaviour only if there are certain elements on a given page. For example, a malicious extension might replace advertisements with its own - but only in the case when ads already exist on that page. Hulk uses special pages called *HoneyPages* that try and satisfy all the conditions an extensions requires to trigger malicious behaviour (e.g. an advertisement exists).

To do this, HoneyPages have additional JavaScript code that override functions used to query the DOM tree of a particular page. When an extension queries for an element in the DOM, the HoneyPage automatically creates and injects the element, and then returns it to the extension. In doing this, HoneyPages dynamically create a page with the hope of extensions exhibiting additional malicious behaviour [6].

HoneyPages also have many limitations and introduce a mechanism that malicious extensions can leverage to avoid detection.

- Hulk assumes that an extension requires a particular element to be part of the DOM when it queries for it, when in fact the extension might prohibit it. Extensions might elicit less behaviour than without the element.
- "Multi-step querying", where extensions require elements have additional properties, was not supported at the time of publishing Hulk.
- Extensions can determine whether they are currently under analysis by querying for an element that they know does not exist. If the element is successfully returned, the extension knows it is currently executing on a HoneyPage and may decide not to execute maliciously, therefore avoiding detection. Similar evasive behaviour is common in malicious software.

Triggering Events

JavaScript in the browser is event-driven. Web-page and extension scripts can register callbacks that are triggered by browser-level events (e.g. mouse clicks). Additionally, extensions are given access to more browser-level events than normal scripts. For example, extensions are provided with the `chrome.webRequest.onBeforeRequest` function that allows them to intercept and change outgoing HTTP requests. Kapravelos et al. complement HoneyPages with event handler fuzzing which invokes all event callback functions that were registered using the `chrome.webRequest` API by dispatching mock event objects.

However, Hulk only triggers events registered via the `chrome.WebRequest` API and does not consider event handlers registered via the `EventTarget.addEventListener` method. As a result, Hulk is likely to not trigger all possible extension behaviour and is therefore not guaranteed to trigger possible malicious behaviour.

Monitoring

Hulk uses URL extraction, HoneyPages and an event handler fuzzer to attempt to elicit all malicious extension behaviour. In order to analyse the extensions that are eliciting this behaviour, Hulk uses multiple monitoring hooks.

Hulk monitors the extension API and logs[9] any time an extension registers a `chrome.webRequest` callback. It also logs any changes made to HTTP requests using the registered callbacks, such as the removal of security-related HTTP headers, which expose a web server or user to various security risks. For example, by removing a web servers CSP HTTP response headers, a malicious extension can inject code into a page despite the server specifically disallowing it.

Hulk also logs all JavaScript code introduced by the extension being analysed including context scripts and remote scripts. The code is logged to provide a complete picture of the extension's functionality in the case that it is flagged as suspicious. Similarly, a proxy that intercepts HTTP and DNS requests is also used for logging requests for remote scripts and objects and to identify scripts that contact non-existing domains [6].

Analysis

To analyse and label extensions, Hulk uses monitoring, logs and has a set of heuristics. By monitoring the extension API, Hulk labels an extension as malicious if it uninstalls other extensions, prevents uninstallation of the current extension, manipulates HTTP headers or attempts to inspect sensitive DOM elements (such as a password field). If an extension modifies a HTTP request, executes remote code or attempts to contact a non-existing (potentially blacklisted) domain, it is marked as suspicious. Otherwise, the extension is deemed benign.

One limitation of Hulk is that it flags many extensions (4,712 of the 48,332) as suspicious and has a low precision [5] which subsequently requires extensive human analysis to promote to malicious or discard as benign. Additionally, Hulk can completely miss malicious extensions that interact with a page without a user knowing (e.g. clicking a "like" button on Facebook)[10] and may label them as benign.

Moreover, although behaviours such as changing or removing HTTP headers, uninstalling extensions, or preventing the uninstallation of the current extension are exclusive to malware, they are very rare. These behavioural signal were shown to have a precision of 86%, 96% and 100% and recall of 2%, 0.5% and 0.1% respectively [5], emphasising their rarity. This limits the effectiveness of the Hulk tool in detecting malice.

2.5.2 WebEval: Fighting Malicious Extensions on the CWS

Jagpal et al. at Google also presented a similar tool, WebEval [5], which detects malicious Chrome extensions on the CWS and returns a concrete verdict of extension malice. WebEval expands on previous work by analysing an extension's behaviour, code and its developers' reputation. As WebEval was deployed, at the time of publishing, and protecting the CWS, attackers are more likely (in comparison to Hulk, for example) to adapt their malicious extensions to evade detection. WebEval uses a classifier that takes many extension signals and also uses human experts to validate claims that an extension is malicious before removing it from the CWS and to correct off model-drift caused by evasion.

[9] Chrome provides a logging infrastructure used to monitor the activity of extensions.

[10] Kapravelos et al. did find a malicious extension, creating and inserting fake Facebook status updates, by visually inspecting the HoneyPage, which was polluted with spam posts.

Static Analysis

Unlike Hulk, WebEval uses simple static analysis to give its classifier and human experts more information regarding the extension and compliment the tools dynamic part. This analysis is not necessarily complete as extensions with certain permissions can request remote script or resources.

WebEval scans for multiple types of code obfuscation that could indicate malice: minification, encoding and packing. The tool also compares code, file names, and directory structure to that of other known malicious extensions to detect near-duplicates. This aims to prevent other developers uploading the same malicious libraries or uploading a modified malicious extension. WebEval also notes what permissions each extension requires which suggests an extension's capabilities.

Furthermore, due to WebEvals integration with Google's CWS, WebEval also monitors where developers log in from, their email address, the number of extensions developed and the age of the developer account. This attempts to catch fake developer accounts mostly used for publishing malicious extensions.

If an extension passes initial analysis and is published on the CWS, meta-data gathered from installations is also used in the following rounds of analysis. This includes the number of installs, ratings and average rating the extension receives.

Dynamic Analysis

Similar to Hulk, most of the signals that determine if an extension is malicious come from dynamic analysis. WebEval uses black-box testing by running extensions in sand-boxed test suites that simulate common browsing habits and examines changes to the sand-box. The dynamic analysis produces a list of network requests, DOM events and Chrome APIs calls which are supplied to WebEval's classifier and to human experts.

WebEval runs extensions in a sand-boxed environment, a Windows virtual machine, with a *system monitor* and *in-browser activity monitor*. The *system monitor* tracks low-level environment changes such as file system changes, Chrome setting changes and Windows operating system settings. The *in-browser activity monitor*, built into Chrome to prevent extension interference, logs all DOM events and Chrome API calls. Additionally and very similarly to Hulk, WebEval uses a *network logging proxy* that also servers as a *replay cache*, ensuring the test suites are as deterministic as possible across executions, for future human expert verification [5].

Unlike Hulk's event handler fuzzer, WebEval uses a set of behavioural suites that attempt to replay realistic browsing scenarios to trigger malicious behaviour in the extensions under analysis. WebEval takes advantage of human experts by having them record complex interactions (including querying Google, using Facebook and shopping with Amazon or Walmart) with web-pages that are then replayed. These behaviour suites are improved and added to as new threats arise. In order to test a wide variety of browser states and actions, WebEval also uses the generic test suite created by Kapravelos et al. which utilise HoneyPages and Hulk's fuzzer.

Similar to Hulk, and other dynamic analysis tools responsible for detecting malicious software, sophisticated malicious extension developers may be able to evade WebEval's detection using *cloaking*. Malware can delay execution until after evaluation or fetch benign versions of remote JavaScript until after evaluation. This is a serious issue as extensions often successfully employ methods to determine if they are under evaluation (for example, by fingerprinting the evaluation environment or IP address).

Relevance of Individual Signals

WebEval has been extremely useful in surfacing malicious extensions and has had, over an analysis period of three years, a recall (the number of malicious extensions flagged) of 93.3% and precision (those that human experts agreed with) of 73.7%. However, the work carried out by Jagpal et al. in presenting the most relevant signals of malice according to their classifier has proved a great contribution in relation to this project.

DOM Operations	Precision	Recall
`eval`	10%	76%
`Window.navigator`	19%	59%
`XMLHttpRequest.onreadystatechange`	31%	56%
`XMLHttpRequest.open`	21%	53%
`Document.createElement`	20%	47%
`Window.setTimeout`	18%	46%
`Node.appendChild`	20%	45%
`HTMLElement.onload`	25%	30%
`HTMLScriptElement.src`	51%	25%
`Window.location`	23%	12%

Table 2.1: Top 10 DOM operations observed (recall) during WebEval's dynamic evaluation in descending order.

The most relevant signal for detecting malice in extensions are their Chrome API calls and DOM operations (see Table 2.1). Although the majority of extensions inject scripts (through CSP relaxing), generating network requests and adding new DOM elements, malicious extensions have a different set of events triggered and load resources from similar domains. WebEval uses a machine learned model to determine which resources are commonly requested by malicious extensions. As you can see, certain DOM operations indicate malice with quite high precision and if we were to alert users of extension actions, it would be wise to prioritise by precision.

Jagpal et al. also discuss the relevance of different permissions. They note that while malicious extensions tend to request the tab permission (84% requested) or the ability to generate network requests (39% requested), these are also highly prevalent in benign applications. They consider the abundance of coarse permissions a limitation of the Chrome permission model, as they do not indicate whether an extension has malicious intent.

2.6 Sensitive Data Tracking

A lot of work has also been done on tracking sensitive data as it flows through browser-based systems and this has been previously used to detect malicious extensions. Hulk actively tracks the flow of sensitive data, considering any sensitive information leaving the browser via an extension as theft and therefore malicious [6]. Other work has utilised data flow tracking and tainting to prevent cross-site scripting attacks and protect online accidental data disclosure.

2.6.1 PHP Aspis: Taint Tracking

Papagiannis et al. introduced PHP Aspis [32], a tool that applies partial taint tracking at the PHP language level to prevent code injection vulnerabilities, such as cross-site scripting or SQL injection attacks. PHP Aspis performs taint tracking at the source code level by rewriting it to track the origin of characters in strings and then sanitises untrusted values to prevent code injection. Consequently, the tool does not need complex modifications to the language run-time.

The PHP Aspis tool does not directly deal with malicious extensions but instead the general issue of tracking sensitive data through a system. Source code developers are assumed to be trusted and not adversaries.

Run-time Taint Tracking

According to Papagiannis et al., run-time taint tracking is composed of three steps:

1. **Data entry points**. Sensitive and untrusted user data entering the system are "tainted" to specify that they are user provided. The *taint meta-data* can vary as long as it captures the origin.
2. **Taint propagation**. As the system executes, any data that depends on "tainted" data is also tainted. For example, concatenating a tainted string with another would result in the returned value being tainted. This in turn propagates the *taint meta-data* throughout the system.
3. **Guarded sinks**. Every function that can create a code injection vulnerability (e.g. `eval`) is intercepted. If passed tainted data, it then calls a corresponding sanitised function or completely aborts the call. Otherwise, it executes as normal. This can involve replacing insecure low-level functions with secure higher-level ones and can consequently have a negative impact on performance.

Partial Taint Tracking

As not all code is as prone to vulnerabilities, and due to the performance overheard of complete taint tracking, PHP Aspis uses partial taint tracking by focusing on third-party Wordpress plug-ins.

PHP Aspis first modifies code that requests data from users and taints the user provided data. The tool also splits the code base into *tracking* (in this case, third-party plugin code) and *non-tracking* code. In the *tracking* code, PHP Aspis performs taint propagation and guards sinks as in run-time taint propagation. Whereas in *non-tracking* code, PHP Aspis performs no taint tracking, assuming the code to be secure.

As Wordpress third-party plug-ins can be seen as analogous to Chrome extensions, the partial taint tracking principles may be applicable to JavaScript extensions.

Transforming Extension Code

Although the tool is responsible for preventing code injection attacks on benign-but-buggy web applications (such as Wordpress plug-ins), the same principles can apply to tracking sensitive information in systems with malicious components. For example, it is possible to transform a Chrome extension's content script to capture potentially malicious DOM operations. Another possibility is to introduce additional JavaScript, as in Hulk's HoneyPages [6], to override suspicious DOM operations.

In Listing 3, we present some trivial JavaScript code that when added to an extension's code, overrides the DOM method `Node.appendChild` to first prompt a user for their permission. One limitation of is that sophisticated attackers can circumvent this transformation by calling the `Node.oldAppendChild` method instead.

```
var oldAppendChild = Element.prototype.appendChild;
var extensionName = "(Malicious) Ad Blocker";
Element.prototype.appendChild = function() {
    var message =
        "Allow '" + extensionName +
        "' to insert an element into this page?";
    if (confirm(message) == true) {
        oldAppendChild.apply(this, arguments);
    }
};
document.body.appendChild(document.createElement("div"));
```

Listing 3: JavaScript code that transforms the `Node.appendChild` method to request the user's permission before executing.

Dynamic Features

An alternative approach could be to transform the calls directly, replacing calls to `Node.appendChild` with calls to a different method. However, a sophisticated attacker can still potentially overcome this by dynamically executing remotely fetched code (e.g. by using the `eval` function). PHP Aspis solve this dilemma by rewriting the dynamically provided code at run-time [32]. Nonetheless, if an extension requests `eval` privileges, a similar approach is possible with JavaScript but would likely have negative performance impact. PHP Aspis attempts to address this performance impact by using a caching mechanism when available but still incurs a significant performance penalty if taint tracking is not restricted to subsections of PHP applications [32].

Limitations

Unfortunately, PHP Aspis suffers from false negatives as the tool does not correctly propagate taints through built-in library methods, through calls that exit tracking contexts and when data flows to external data stores such as a file system or database [32]. As a result of this limitation, PHP Aspis cannot guarantee that all XSS vulnerabilities are prevented. Papagiannis et al. substantiate this claim by noting that two Wordpress plugin injection vulnerabilities reported via Common Vulnerabilities and Exposures (CVE) [33] reports.

2.6.2 Dynamic Spyware Analysis

Egele et al. present a dynamic analysis approach that tracks the flow of sensitive data throughout a Browser Helper Objects (BHOs)[11] in the IE browser [7]. Their approach is designed to prevent the loss of sensitive data to spyware, code that monitors and steals the behaviour of users and their data. This is closely tied with the user tracking threat that is present in some malicious extensions. Similar to

[11] BHOs are separate and different from the IE plug-ins previously mentioned. They rarely implement complex functionality or user interfaces.

WebEval, their tool allows human experts to follow the flows of sensitive data in a system in order to determine malice.

Their approach presented uses dynamic taint analysis to follow sensitive data (such as URL and web-page information) as it propagates through IE and any loaded BHOs. Whenever a BHO attempts to transmit sensitive data on the network, the helper object is classified as spyware. Their technique does not involve analysing one "extension", but instead classifies all loaded BHOs. Egele et al.'s approach takes into account both *data dependencies* (as in PHP Aspis) and *control flow dependencies.*

A *data dependency* ensures that operation or assignment that uses a tainted value (such as string concatenation) returns a tainted value. On the other hand, *control flow dependencies* ensures that variables are tainted if there is a dependency between its value and a tainted value. It is possible to write code that avoids taint analysis by abusing execution control flow. The contrived code example in Listing 4 shows how a sensitive variable, `data`, can be leaked if *control flow dependencies* are not employed. More complex code can similarly copy strings and other objects without tainting.

```
var data; // Tainted Boolean value
var secret; // This variable remains untainted and can be leaked
if (data) {
    secret = true;
} else {
    secret = false;
}
```

Listing 4: Leaking sensitive information via control flow manipulation.

Direct Control Dependencies

To mitigate the risk of a sophisticated attacker using the control flow to launder tainted data, the researchers use *direct control dependencies*. This process involves tainting the result of an operation if the execution depends on the value of a tainted variable (as shown in Listing 4). A taint engine examines all conditional branches visited during execution and for each one that depends on a tainted variable, it taints all operations in that branch's scope using static analysis.

Operating System Awareness

Determining if a BHO is attempting to steal sensitive data (for example, by writing to a hidden log) is made more difficult by the fact that both BHOs and IE run in the same process. Egele et al.'s approach runs IE and BHOs in a Qemu, an open source system emulator, to facilitate monitoring. This introduces an interesting problem: what actions performed in these IE processes are executed in the context of a BHO. This issue is solved by pushing a flag onto the stack that indicates execution is in a BHOs context. For every following instruction, if the stack pointer is below the flag, then it is executing within a BHOs context.

Automated Browser Testing

Similar to previous tools that attempt to detect malicious extensions, Egele et al. built an automated browser testing suite that mimics the browsing habits of users. Interestingly, they recorded browsing

behaviour using a Firefox extension and then replayed it in IE using a Windows application. The application captures URLs visited and forms fields that are filled in and automatically replays the action of visiting these pages and submitted the forms.

2.6.3 BrowserFlow: Data Flow Tracking

Papagiannis et al. introduced *imprecise data flow tracking*, and built a Chrome browser plug-in[12], BrowserFlow [34], that notifies users if they accidentally expose potentially sensitive data to external cloud services (e.g. Google Docs). BrowserFlow defends against *trusted-but-careless* users that may unintentionally leak, mainly by copy-and-pasting, sensitive data to third-party web-services, thereby breaking data disclosure agreements.

Text Disclosure Model

Papagiannis et al. also introduced a Text Disclosure Model (TDM) to represent the action of copy-and-pasting data between browser tabs. In this model, each text segment on a page is associated with *security label*s, which are sets of *tags* representing the segments' sensitivity based on the browser tab in which they were created. Tags can be used to represent entire categories of sensitive data (e.g. `interview-data`) or specific data (e.g. `product-announcement-x`) and are indirectly assigned by data administrators [34].

Imprecise Data Flow Tracking

Unlike the PHP Aspis tool [32] or Egele and colleagues' work with sensitive data tracking in BHOs [7], BrowserFlow does not require modifications to the browser or system binaries needed to add taint labels to data flowing through a system. Instead, BrowserFlow implicitly and imprecisely tracks data propagation by assigning tags to new text segments based on the similarity between it and other previously examined (and tagged) segments.

Tag propagation is imprecise as modified data is tracked as long as it maintains a certain level of similarity with source data. BrowserFlow does this by comparing the *fingerprints* - a set of carefully selected hashes - of text segments [34].

User Alerts

Instead of completely blocking sensitive data leakage, the BrowserFlow plug-in alerts users when accidental data disclosure is occurring by changing the background colour of text elements containing sensitive data to red. Furthermore, if the data is not permitted to flow to a particular untrustworthy cloud service, it is transparently encrypted before being sent to the server.

Users also have the ability to declassify sensitive data by suppressing tags associated with a particular text segment. Every time a user wishes to disclose the same text segment, they must perform tag suppression. As this may result in users disclosing sensitive information, BrowserFlow stores information about the tag suppression for future audit [34].

[12] Although not elaborated upon, it is likely BrowserFlow is a Chrome extension or application.

2.7 Summary Of Previous Work

In Table 2.2, we compare the 5 existing solutions we researched, which attempt to detect malicious extensions or sensitive data leakage, across multiple dimensions. These approaches directly influenced many of our project's design decisions.

	Hulk	WebEval	PHP Aspis	Dynamic Spyware Analysis	BrowserFlow
Static Analysis		✓	✓		
Automated, Sand-boxed Browser Analysis	✓	✓		✓	
Run-time Analysis with Browser Instrumentation				✓	✓
Code Transformation			✓		
Human Verification	✓	✓			✓

Table 2.2: A comparison of existing solutions researched across various dimensions.

The most common solution we observed in previous work has been to automatically analyse extensions within a sand-boxed environment. The main limitation of this approach, as noted by Kapravelos et al., is that sand-boxes can be easily fingerprinted by malicious extensions and they can in turn suppress their malicious intent while under analysis [6].

The PHP Aspis tool statically transforms source code [32] and WebEval incorporates meta-data in their model [5]. The former can only do this as Papagiannis et al. do not consider source code developers as adversaries and suffers from false negatives. Moreover, the latter only uses static analysis to supplement its main analysis.

Only the Dynamic Spyware Analysis project presented by Egele et al. actually modifies and instruments the browser [7]. The BrowserFlow project proposes using a browser extension to analyse data disclosure at run-time; we consider this browser instrumentation for the purpose of this comparison [34]. The latter is possible as Papagiannis et al. worked to prevent accidental data disclosure by *trusted-but-careless* actors who are not considered malicious.

Both the Hulk and WebEval tools utilised human experts to verify claims of malice in extensions while the BrowserFlow tool allowed users to declassify sensitive data and suppress associated tags at run-time. The former tools produced many false positives while the latter gives allows users to suppress warnings so to not inconvenience them.

Suspicious Extension Actions

Browser extensions have access to a set of powerful APIs from within their background page [24]. These provide a variety of functionality ranging from the ability to create and manage bookmarks, to the ability to observe and edit network requests in-flight. Extension content scripts injected into web-pages also have access to the page's DOM tree and the standard web interface which can be used to click buttons, or read and relay sensitive data to remote servers. A challenge of this project is to determine what web operations suggest malice when performed by a browser extension.

Unlike previous projects that used many method calls as signals to determine extension malice, our approach involves intercepting suspicious extension actions at run-time and providing the user with the capabilities to allow or prevent actions they consider malicious. A user overwhelmed with seemingly benign action notifications will soon ignore them and consequently, a challenge of this project is to only intercept actions that can harm users. Actions that indicate malice but aren't malicious themselves should not be intercepted. We heavily drew upon an evaluation of DOM operations presented by Jagpal et al. [5] for inspiration (see Table 2.1).

In this chapter, we define a threat model, explain our decision to analyse extensions at run-time and give a high-level architectural overview of our solution. By studying previous research, we compile a set of suspicious operations which include dispatching events, sending and receiving network requests, mutating the DOM tree and elements. We also present a Chrome feature that allows users to manually configure the actions that our browser considers suspicious.

3.1 Threat Model

Based on research into threats and popular existing attacks at the hands of malicious extensions, we adopt a simple and practical threat model where we assume all extensions are malicious and are attempting to execute malicious operations on web-pages. We assume the main goal of attackers is to realise a Facebook hijacking, ad injection or user tracking threat as these are the popular attacks (§2.3).

Although malicious extensions can perform attacks from within their background pages (e.g. stealing a user's browsing history), we narrow our focus and consider content scripts as the sole attack vector. Our rationale is that this attack vector is likely due to the high presence of content scripts that can execute on all web-pages and since content scripts can realize attacks using only standard web interfaces. Unlike the Chrome extension API, which requests users grant specific permissions to extensions, the majority of the standard Web API is accessible to content scripts without a fine-grain permission model. We do not attempt to solve the issue of Chrome's extension permission model's shortcomings by purposefully excluding operations that are only available to Chrome extensions through Chrome's extension APIs [24].

Lastly, we assume that web-pages and the scripts that they load are benign and that extensions are not

vulnerable to attacks from malicious web-pages. We aim to protect both browser users and web-pages from malicious extensions.

3.1.1 Malicious Extensions

For the purpose of this project, we define malicious extensions as any extension exhibiting malicious behaviour. We consider behaviour to be malicious if it acts against the requirements of a browser user or acts without their consent or knowledge.

3.2 Run-time Analysis

Previous work has statically or dynamically analysed extensions executed within sand-boxes [5, 6]. One limitation of these approaches is that sand-boxes can be easily fingerprinted by attackers. Malicious extensions can take advantage of this limitation by "acting benign" while they are under dynamic analysis or by dynamically executing code fetched from remote servers. Unfortunately, these analysis methods ultimately turn extension security into a cat-and-mouse game[1] whereby attackers are always attempting to outsmart and evade analysis tools. These analysis tools do not provide the security guarantees that we aim to achieve in this project.

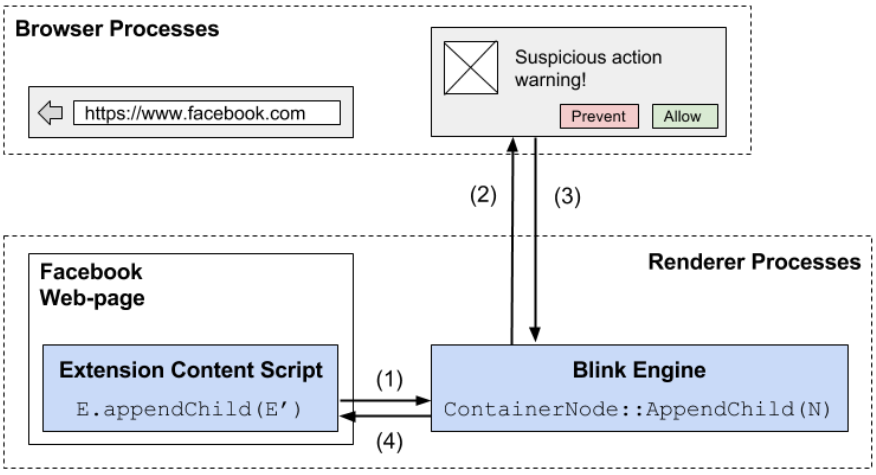

Figure 3.1: A high-level architectural overview of our run-time solution. Web API methods executed by content scripts on web-pages are handled by the Blink Engine (1). If the Blink engine considers them suspicious, we display a warning dialog in the browser (2) and allow or prevent the action based on user response (3, 4). Due to Chrome's multi-process architecture (§2.1.1), some instructions are sent via Inter-Process Communication (IPC) messages (2, 3).

Our project takes a novel approach when analysing Chrome browser extensions. In order to provide guarantees about whether malicious behaviour is executing, we analyse the Blink methods executed by extensions at run-time. If we deem these actions suspicious, we present users with a dialog with which they can allow or prevent malicious actions (see Figure 3.1).

One limitation of this approach is that malicious extensions may have infected many browsers by the time they are detected. Fortunately, Google can and has the right to[2] remotely uninstall malicious extensions

[1] A colloquial term to describe the act of constant pursuit, near captures, and repeated escape between two actors [35].
[2] According to the Google Chrome Web Store Developer Agreement [36]

from users' browsers. As a result, this limitation can be easily sidestepped with cooperation from Google and the Chrome Web Store (CWS).

3.3 Events

Malicious Chrome extensions can perform Facebook hijacking attacks by firing events. For example, a malicious extension can "like" a particular Facebook post by searching the DOM for a "like" button and firing a `click` event on it. Malicious extensions can also utilise events to spread and infect other users by typing and sending seemingly genuine Facebook posts or private messages that recommend the extension.

As a result, we consider most events fired by an extension as suspicious. We implicitly allow certain browser events that do not pose a threat to users and that are triggered frequently when content scripts load (§6.1). There are many ways to trigger events in JavaScript (e.g. calling the `click` method on a element object) but fortunately, each eventually calls the `EventDispatcher::dispatch` method. We therefore intercept this method despite it not being part of a Web API interface to prevent code duplication.

3.4 Navigator

The `window.navigator` object provides information about the identity and state of the user agent (i.e. the browser application) and allows a web-page to register itself as a handler for particular URL schemes (including custom ones).

According to the Web Hypertext Application Technology Working Group's (WHATAG) standard specifications, information from the `window.navigator` API "that varies[3] from user to user can be used to profile the user" [37]. Jagpal et al. also frequently observed the use of the `window.navigator` object in malicious extensions (i.e. this signal had high recall).

Despite this, we do not intercept these operations when performed by browser extensions for several reasons.

- According to the evaluation presented by Jagpal et al., the precision was not very high and therefore, using `window.navigation` is not a good indicator of malice.
- Users are already prompted (see Figure 3.2) when a web-page attempts to register itself as a handler for a particular URL scheme and therefore, displaying another prompt would be redundant.
- We believe that everyday users are unlikely to consider a browser extension attempting to learn more about the particular browser as having malicious intent.

[3]Many methods in the `window.navigator` API are deprecated and return constant strings. See `window.navigator.appCodeName`, `window.navigator.appName` and other fields.

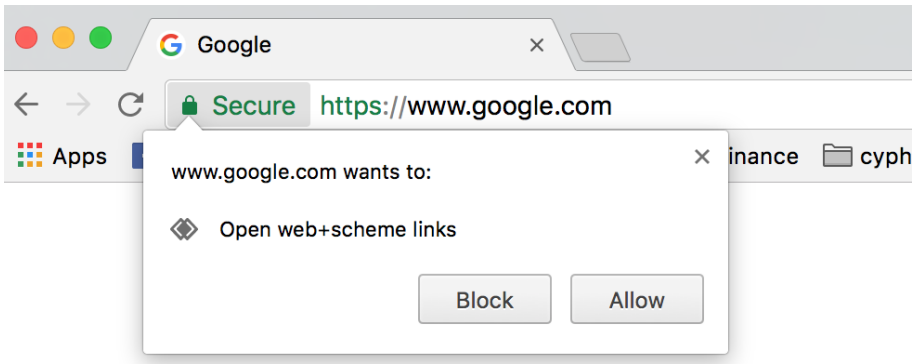

Figure 3.2: The pop-up prompt that is displayed when a web-page attempts to register itself as a URL scheme handler. In this case, the scheme registered is `"web+scheme"`.

3.5 Network

A browser extension's content script requires special permissions in order to communicate with external pages. For example, in order to send network requests to the origin `https://www.google.com` from within a non-Google domain, an extension must include this URL in its `manifest.json` file.

However, as mentioned in Subsection 2.4.1, Chrome's permission model has noticeable shortcomings. For example, when installing an extension with a content script that is injected on `https://www.facebook.com/` pages and that has the permissions to send network requests to the origin `https://www.google.com`, users are warned that the extension "can read and change [their] data on `https://www.facebook.com` and `https://www.google.com`" (see Figure 3.3). To an uninformed user, it is not clear that an extension's content script injected on `https://www.facebook.com` can send network requests to `https://www.google.com` (even though it can).

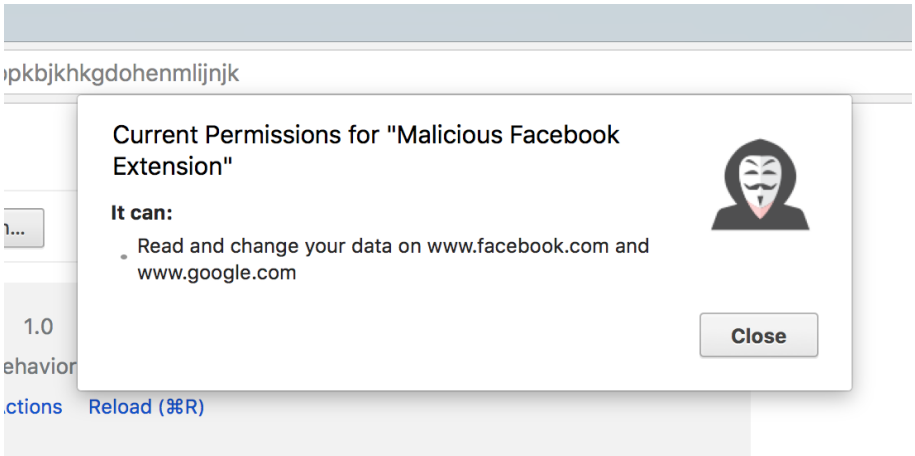

Figure 3.3: The permissions requested from users that allow an extension to be loaded on `www.facebook.com` and that allow an extension to send network requests to `www.google.com`.

Furthermore, a web-page's Content Security Policy (CSP), that prevents scripts and resources being loaded from potentially malicious third-parties, does not apply to an extension's content script. Instead, content scripts, which run within isolated worlds with unique security origins, must abide by their own CSP.

By intercepting network requests and responses, and providing the actual network data to users, we

believe we sufficiently protect against the shortcomings of the permission model.

3.5.1 Network Requests

Since the ability to send network requests from within a content script has very serious consequences (e.g. a malicious content script could send sensitive user data to a remote server), we decided to intercept all network requests sent from within a content script by default[4].

Even though Jagpal et al. mention the `XMLHttpRequest.open` method as being a strong signal of malice, the actual method neither sends a network request nor asserts the provided URL or parameters are correct. We chose to intercept the `XMLHttpRequest.send` method as it does in fact send a network request[5] and takes POST data as an argument. We considered intercepting both methods as redundant and chose not to do so in order to not inconvenience users.

3.5.2 Network Responses

Extensions can also receive remote instructions or scripts to be executed using `eval` and similar methods. Jagpal et al. considered registering a `XMLHttpRequest.onreadystatechange` event handler, which is called on network responses[6], as a key signal and indicator of malice. However, as network responses might not be read in the `XMLHttpRequest.onreadystatechange` event handler, we consider this signal too coarse.

Our approach only intercepts operations that directly read response data. For a given response, we intercept the first read operation (e.g. `XMLHttpRequest.statusText`, `XMLHttpRequest.responseText`, etc.) and allow or disallow all following read operations based on the user's decision. This is implemented using a private `CanReadResponse enum` and corresponding field in the `XMLHttpRequest` class (see Listing 23 in Appendix A).

As response data may arrive at different times (e.g. headers arrive before the response body), we reset the `XMLHttpRequest.can_extension_read_response_` field to `XMLHttpRequest::ReadResponse-Permission::kPromptUser` whenever the state of the response changes. This prevents malicious extensions from attempting to read the response object before it has arrived, receiving permission to read it from the user (as it does not yet contain malicious data) and then reading the response again once it has arrived with malicious data without needing permission from the user.

We intercept all read operations as malicious servers can disguise attack payloads in seemingly benign response fields (e.g. HTTP response status). However, as it is trivial for malicious extensions to disguise responses as benign, it is hard for a user to statically determine if a network response is malicious. We therefore do not intercept response reads by default.

[4] Users are still able to configure Chromium to intercept them if they wish (§3.10).

[5] The `XMLHttpRequest.open` method must be called before the `XMLHttpRequest.send` method which could explain why WebEval used it as a signal.

[6] The `XMLHttpRequest.onreadystatechange` event handler is also called when headers are received, the response is loading and other state changes.

3.6 Script Injection

Extensions can execute code formatted as strings by using the `eval` function, with window timers (e.g. `Window.setTimeout`), by setting the `src` field of `script` elements and via other constructs. Extensions can use these methods to fetch malicious code when they are certain they are not under evaluation (e.g. sand-boxed by the WebEval tool) in order to evade detection. It is therefore not surprising that both the `eval` and `Window.setTimeout` methods are common operations in malicious extensions, according to Table 2.1.

However, unlike events or network requests, we do not consider the act of dynamically executing code as suspicious - even if fetched from a third-party. In Section 4.5.2, we guarantee that suspicious extension actions performed within injected script are still considered "extension actions" and we believe intercepting both operations that execute dynamic code and that execute suspicious actions to be unnecessary.

3.7 DOM Tree Mutation

The ad injection and user tracking threats (§2.3) are only possible if an extension can mutate the DOM tree. Specifically, a malicious extension must call either the `Node replaceChild`, `insertBefore`, `appendChild` or `removeChild` methods to inject advertisements or tracking pixels onto a web-page. Furthermore, Jagpal et al. considered these methods[7] as potential indicators of malice. As a result, we intercept all four methods.

By intercepting the four methods above, we also transitively intercept an extension that mutates the `Element innerHTML` and `outerHTML` fields or executes the `Element insertAdjacentText`, `insertAdjacentHTML` and `insertAdjacentElement` methods.

Although the `Document.createElement` method is also a relevant signal according to Jagpal et al., we decided not to intercept it by default. The element created may not be inserted into the page's DOM tree and even if it is, we believe it is redundant to intercept the `Document.createElement` method considering we also intercept the methods that insert elements on the page.

Scripts can also replace the entire HTML document by using the `Document writeln` and `write` methods. However, we currently do not intercept these methods as this operation would completely change the appearance of the page and since users would likely notice this, there is little point displaying a warning dialog.

3.8 Element Attribute Mutation

Allowing an extension to mutate elements on a web-page can potentially be as dangerous as allowing them to fire events or mutate the DOM tree. For example, a malicious extension could modify the `src` attribute of all hyperlink elements to direct users to a malicious web-page whenever they click the link. In this case, we would not intercept the click event as it was fired by the user and not the extension.

[7] In fact, only the `Node.appendChild` method was listed we consider the others to be synonyms due to they interchangeability.

Fortunately, many operations that mutate elements are intercepted transitively by intercepting operations that mutate the DOM (§3.7). They are intercepted transitively as these methods often internally call `Node.replaceChild` (or similar methods). Users can choose to intercept the following field mutations and method calls:

- `Element.innerHTML`
- `Element.outerHTML`
- `Element.innerText`
- `Element.outerText`
- `Element.insertAdjacentHTML`
- `Element.insertAdjacentText`
- `Element.insertAdjacentElement`

As a result, we only explicitly intercept element attribute modifications via the `Element.setAttribute` and `Element.removeAttribute` methods. By default, we decided not to intercept most attribute mutation in order not to overwhelm the user. However, we consider it suspicious when an extension attempts to change the "sensitivity" or "description" of an element. This data is stored as the Boolean attribute `data-sensitive` and the String attribute `data-description` and as such, we intercept the methods `Element.setAttribute` and `Element.removeAttribute` if the attribute is either `data-sensitive` or `data-description`.

3.8.1 Sensitivity

The `data-sensitive` attribute can be set by web-pages to mark certain elements (and their children) as more sensitive than others. For example, a web-page may mark a login-form or button as sensitive. If users are overwhelmed by the number of extension action permission prompts, they may choose to filter out actions performed on non-sensitive elements (§6.4).

Marking an element as not "sensitive" may result in the user not being prompted when an action occurs on that element. On the other hand, marking many elements as "sensitive" may dilute their importance. We therefore explicitly forbid extensions from doing this. When we intercept an extension calling `Element.removeAttribute("data-sensitive")` or `Element.setAttribute("data-sensitive", false)`, we throw an exception without displaying a user prompt.

3.8.2 Description

The `data-description` attribute can be set by web-pages or extensions to describe certain elements on the page. This provides more information to users when prompted about extensions actions (§5.7).

Some extensions may have a legitimate reason to change the description of an element. For example, if they mutate the element, a new description may be required. For this reason, we show users a prompt, asking them if they would allow the extension to call `Element.setAttribute("data-description", ...)`. However, similarly to the `data-sensitive` attribute, we explicitly forbid extensions from calling `Element.removeAttribute("data-description")`.

3.9 Reading Element Data

As reading data from a web-page can be malicious if it contains sensitive data (e.g. secret tokens[8] or passwords), we also give users the option to intercept extension operations that read element data. We decided not to intercept operations to read element data by default as this would likely flood users with mostly benign extension action pop-ups. We intercept the following field mutations and method calls:

- `Element.getAttribute`
- `Element.hasAttribute`
- `Element.innerHTML`
- `Element.outerHTML`
- `Element.innerText`
- `Element.outerText`

3.10 Configuring Suspicious Actions Manually

Despite evaluating and choosing which extension operations to intercept and deem suspicious, we decided to provide a mechanism for users to configure this themselves. On the "Chromium Extensions" settings web-page, `chrome://extensions/`, we added a link so that users can configure suspicious actions separately for each extension (see Figure 3.4). Once clicked, we display a pop-up dialog (see Figure 3.5) where a user can select or deselect various classes of operations (provided in a human-readable manner).

This gives browser users the ability to tailor their security preferences for each extension they have installed. For example, a security-conscious user may consider more actions executed by a newly-installed, untrustworthy extension as suspicious and may therefore want to be asked permission more often. On the other hand, a user may wish to allow all operations from a trusted extension (for example, their password manager).

3.10.1 Implementation

Configuration Persistence

The main challenge we faced when implementing this mechanism was persistently storing a users settings. If a user restarts their Chrome browser, their settings should remain intact. Configuring the suspicious actions settings should also ideally have an immediate effect on an extension's content script running in another tab (without needing to refresh that page). Finally, according to Chrome multi-process architecture security best practices, we must ensure a compromised renderer process could not change the suspicious actions configuration. This would allow a malicious actor in control of the renderer process to turn off interception for all suspicious actions.

Once created, `Extension` objects in Chrome are immutable in order to make them safe to use concurrently across multiple threads. Given an extension identifier, they can be retrieved via an `ExtensionRegistry`,

[8] Websites often attach secret tokens to forms to prevent Cross-Site Request Forgery (CSRF) attacks [38].

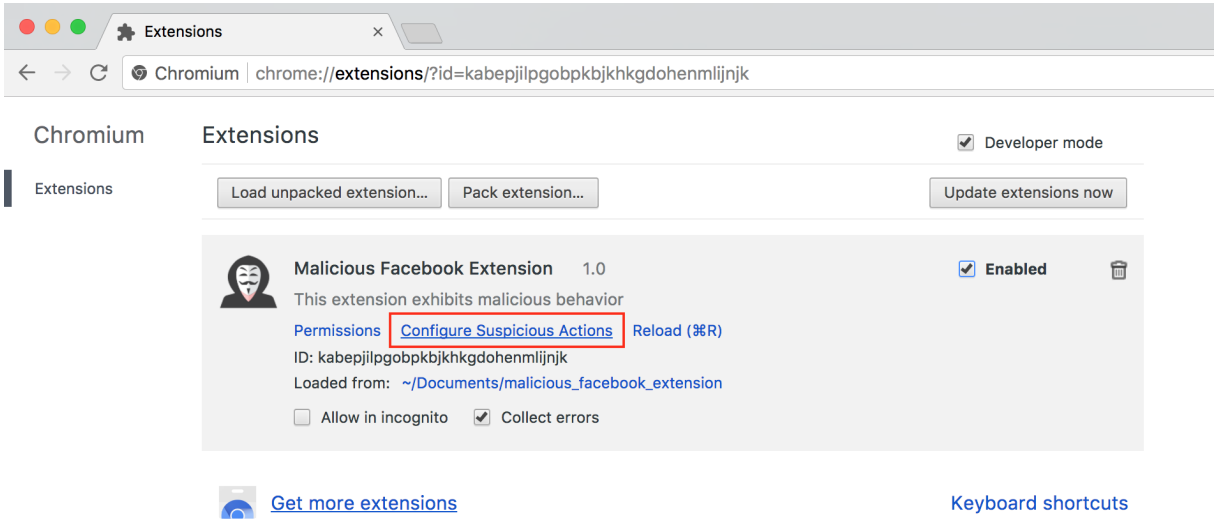

Figure 3.4: An example "Chromium Extensions" settings page that has been updated to add a "Configure Suspicious Actions" link (See red border) to allow users to configure the actions that are intercepted for the "Malicious Facebook Extension" extension.

which also keeps track of enabled, disabled, terminated, blacklisted and blocked `Extensions`. Each extension also has its own `RuntimeData` object that contains mutable data that is accessible on the UI thread in the browser process. However, `RuntimeData` does not persist across browser restarts and therefore was not used as a data store.

Fortunately, a key-value data store exists in the form of `StateStore` object, which is used to store per-extension state that needs to persist across restarts. The `StateStore` object is only accessible from the browser process which protects against compromised renderer process. The data store maps `std::string`s to `Value`s objects which represent data that can be stored in JSON (i.e. of type `NONE`, `BOOLEAN`, `INTEGER`, `DOUBLE`, `STRING`, `BINARY`, `DICTIONARY`, or `LIST`).

We created a `SuspiciousActionConfig` class (see Listing 21 in Appendix A) that abstracts away the underlying `StateStore` and provides default values for keys not already existing in the data store. We map `ExtensionAction` representing each class of actions to a `BOOLEAN` flag.

In order to read a value from the `StateStore` object, you must provide a callback function to be executed once the value is read from the data store. When opening the "Configure Suspicious Actions" pop-up, we perform 9 separate reads to determine the configuration. Although we haven't noticed any serious performance overheads from doing this, we could speed this process in the future up by storing the entire configuration as a JSON dictionary and only reading it once per pop-up.

Dialog

Unlike the user dialog displayed when extensions attempt to trigger suspicious extensions (see Subsection 5.5), the configuration page and associated pop-ups both "live" within the browser process. As a result, we did not need to send IPC messages from a renderer process to the browser process in order to display the configuration pop-up.

Instead, we simply modified the chrome extensions configuration page by adding the additional link and

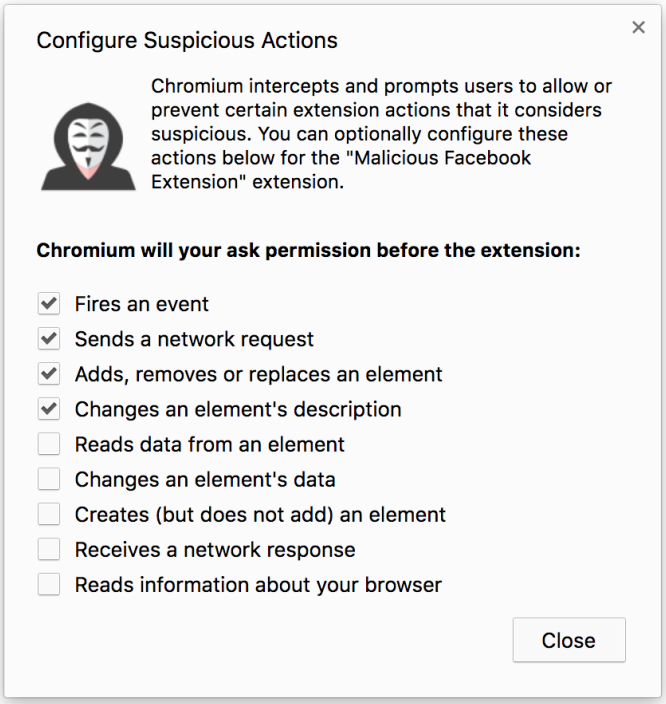

Figure 3.5: A pop-up dialog that is displayed to users when they click on the "Configure Suspicious Actions" link in Figure 3.4.

some JavaScript (see Listing 22 in Appendix A) that calls into private[9] method in a developer tools class which in turns displays a dialog.

The custom dialog is fairly simple and follows a similar design and implementation to the warning dialogs displayed when extensions attempt to execute suspicious actions. The dialog consists of an extension icon, text field and a series of checkboxes that read from the `SuspciousActionConfig` when initialised and write to it when modified (see Listing 24 in Appendix A). We currently do not handle the case where the configuration dialog is opened within two separate tabs or windows and checkbox state goes out of sync with the data store due to concurrent user input. A possible resolution would be to have the configuration view listen to updates to the data store but this has not yet been implemented as we believe this case to be extremely rare.

3.11 Summary

In this chapter, we define a simple threat model which we use to guide this project. The model assumes all extension are malicious, treats content scripts as the main attack vector and considers the aim of attackers to be the realisation of popular threats.

We decide to analyse malicious extensions at run-time, instead of statically or automatically like previous research, to guarantee attackers cannot evade detection by fingerprinting evaluation environments. We also propose a solution that involves requiring user permission when extensions execute suspicious actions.

By considering multiple Web API methods, we decide to intercept extension operations that trigger

[9] In this case private means not exposed to external scripts via a public Web API.

events, send network requests and modify a page. As users trust certain extensions more or less than others, we allow them to configure what operations they consider suspicious. Specifically, we give users the ability to mark operations that read network responses and change or read element attributes as suspicious.

Detecting Extension Actions

In this chapter, we first describe the challenge of detecting whether certain operations were executed by extensions or by other scripts on a web-page. We suggest potential solutions to this challenge that we considered; these include analysing the frequency and type of events fired, and transforming extension JavaScript code before execution. Finally, we present a novel approach and implement it within the Chrome browser such that we guarantee that extension actions cannot evade detection.

4.1 Motivation

In order to intercept suspicious actions so that we can request users to allow or prevent them, we must first be able to determine, with certainty, whether operations were triggered by an extension. Specifically, we must be able to ascertain whether Web API methods within the Blink engine are executing as a result of an extension's content script.

Moreover, users will inevitably take into account the extensions reputation and trustworthiness when classifying suspicious extension actions as malicious or benign. As a result, we must establish what extension in particular is executing a Blink method so that we can present this information to the user.

4.2 Challenges

Detecting the origin of Blink actions is challenging for several reasons. Firstly, the Chrome browser does not currently provide any way, either through public JavaScript APIs or via a private Blink interface, to directly determine whether execution is occurring within the context of an extension script or web-page script.

Secondly, extension content scripts have the ability to insert additional scripts elements into a page's DOM, or execute functions via timers and other asynchronous JavaScript constructs. The Chrome browser treats these inserted scripts identically to script elements originally defined by the web-page. Specifically, these scripts abide by the page's Content Script Policy (CSP) and execute within the page's main world (§4.5).

A key goal of this project is to guarantee that the interception of suspicious actions cannot be evaded by a sophisticated attacker. Consequently, a challenge is to ensure that any method executed by a script element that has been inserted into the DOM tree by an extension is correctly classified as being triggered by that extension. Otherwise, a malicious extension content script can evade detection by executing suspicious actions within an inserted script element.

Finally, a challenge is to solve this problem without modifications to the Chrome browser. By doing so, our solution would hopefully reduce the chance of breaking existing web-pages and extensions, or incurring

a browser performance overhead. A solution that does not involve modifying the Chrome browser might also extend more easily to other browsers with extension systems but different architectures (e.g. the Firefox browser).

4.3 Event Order Analysis

As noted in Section 3.3, malicious extensions can trigger events to perform popular Facebook hijacking attacks. Instead of attempting to solve the problem of determining if an extension is responsible for executing arbitrary suspicious actions, we initially focused on solving this for JavaScript events.

The first idea we explored was inspired by Control Flow Analysis (CFA) techniques and involved tracking the order in which events on a web-page are fired and comparing it to a normal user ordering. Another similar idea was to measure the frequency of events dispatched and compare this to the normal browsing habits of users on particular web-pages. Without any changes to the Chrome web-browser[1], we could potentially determine if an event was triggered by a user (e.g. with their mouse, keyboard or other input device) or triggered virtually by a page or content script.

In general, this solution would be possible since certain events proceed others when triggered by users. For example, if a user were to trigger a `click` event on a particular element, an `onmouseover` event would be expected to fire before. Any events that do conform to normal ordering would have likely been triggered by a script.

4.3.1 Limitations

We decided against this approach as several key limitations were evident. By analysing the order or frequency of events triggered we could only determine if an event was fired by a script (e.g. using the `EventTarget.dispatchEvent` method) or by actual user interaction. If we determined the event was triggered by a script, we wouldn't know if it was a web-page script or potentially malicious content script that dispatched it. Even worse, extensions could disguise events they dispatch by simplifying firing other redundant events, resulting in a cat-and-mouse game. Finally, even if we could detect events triggered by extension content scripts, it is not clear how we would intercept them and prevent other registered event handlers from executing.

4.4 Transforming JavaScript Code

The second idea we considered to determine whether certain actions were triggered by extensions involved transforming content scripts such that suspicious action functions calls would be tainted. This approach is extremely similar to that of PHP Aspis (§2.6.1), which transforms PHP code to eliminate cross-site scripting vulnerabilities.

[1] Similarly to the BrowserFlow plug-in [34], we could track the order of events fired by implementing another browser extension that registered event listeners for every element.

4.4.1 Tainting Extension Events

We first experimented with tainting only extension events for the same reason as in Section 4.3. Our solution involved adding a Boolean attribute to the `Element` interface and transforming every method that triggers an event (e.g. `Element.click`) into the normalised form `Element.dispatchEvent(Event)`. By doing so, we only needed to change the interface for the `Event` constructor and not the declaration of each method that potentially dispatches events (see Listings 5). Specifically, this approach modifies the `EventInit` object, that is passed as an optional parameter to the `Event` constructor[2], to include an optional `extensionOriginating` Boolean field.

Although we chose to pass this information via a Boolean attribute, this only conveyed the fact that the event was triggered by an extension and did not provide information about the extension. Had we pursued this approach further, we probably would have passed an object with meta-data about the extension (e.g. the extension's name, origin and other attributes).

```
// Before transformation
var targetElement = document.getElementById("target-id");
targetElement.click();

// After transformation
var targetElement = document.getElementById("target-id");
targetElement.dispatchEvent(new Event("click", {"extensionOriginating": true}));
```

Listing 5: An example content script before and after transformation.

With the `Event` interface modified in this manner, we can treat `Event`s created by extensions or by the web-page separately within the Blink engine. We can determine if an `Event` object was created by an extension by checking the return value of the `Event::extensionOriginating` method. Using this, we can amend the `EventDispatcher::dispatch` method to prevent extensions from firing events as shown in Listing 6.

```
DispatchEventResult EventDispatcher::dispatch() {
  if (m_event->extensionOriginating()) {
    // Cancel all events that were triggered by an extension before dispatching them.
    return DispatchEventResult::CanceledBeforeDispatch;
  }
  // Rest of the original EventDispatcher::dispatch() function ...
}
```

Listing 6: Modification of a method in Chrome's Blink source code to prevent tainted events from being fired.

4.4.2 Limitations

Dynamic Language

Unfortunately, this method of transforming content scripts has several limitations. Most importantly, it does not provide strong guarantees about action origins (i.e. it does not guarantee an action was not triggered by a browser extension) and attackers could abuse this to evade interception. Due to

[2]This method instantiating `Event`s is a WHATAG standard but not a W3C standard. It is not supported by certain versions of Internet Explorer.

JavaScript's dynamic language nature, an attacker could override the definition of the `Event` constructor (and other functions) to prevent our transformation tool from tainting it (see Listing 7). Moreover, an attacker could trivially obfuscate a content script to prevent a transformation tool from modifying suspicious function calls (see Listing 8). The PHP Aspis tool does not suffer from these limitations as the source code being transformed is assumed to be written by trusted developers [32].

```
// Overrides the Event constructor so that it still works as expected,
// but indicates that an extension did not create it.
oldEvent = Event;
Event = function(type, eventInit) {
    eventInit["extensionOriginating"] = false;
    return new oldEvent(type, eventInit);
}

new Event("click", {"extensionOriginating": true});
// Event["extensionOriginating"] == false
```

Listing 7: A malicious content script can override the `Event` constructor to prevent transformation tools from tainting it.

```
var targetElement = document.getElementById("target-id");
// Call targetElement.click() in an obfuscated manner
targetElement["cli" + "ck"]();
```

Listing 8: A sophisticated attacker can obfuscate calls to suspicious operations to prevent them being transformed by our tool.

Modifying Web Standards

Another issue with this approach is that it would involve proposing and modifying web standards set by the World Wide Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATAG) in order to add caller meta-data to every single suspicious JavaScript method. Without these proposals being accepted, it is highly unlikely that the Chrome browser would accept the changes to the Blink engine.

Furthermore, it is unlikely W3C or WHATAG would approve these proposals as there does not currently exist a cross-browser extension standard [39] and developers must create extensions separately for each browser. Since some browsers software providers might not need to transform extension scripts in order to determine if the caller was an extension and others might disagree on what meta-data is required in order to identify an extension, creating a new standard to satisfy these differences would be non-trivial.

4.5 Isolated Worlds

After rejecting the idea to transform JavaScript code, we began investigating Chrome internals to determine if we can infer execution environment (content script or web-page script) from within the Blink engine.

From earlier research, we recalled that each content script is associated with its own isolated world and each web-page script executes within the renderer process' main world. Within the Blink engine, these

worlds are referred to as `DOMWrapperWorld`s as they "wrap" the underlying internal DOM with different JavaScript objects and prevent access to variables and functions defined in other worlds (§2.2.1).

As extension content scripts and page scripts execute within different worlds, we can determine if execution is occurring within the context of a content script by inspecting the current world. We use this hypothesis to propose a novel solution to this challenge that involves using the isolated worlds concept. To detect if a suspicious Blink method is triggered by a content script, we propose checking if the current world is an isolated one. If so, we assume execution is occurring within an extension's content script. This assumption may be invalid, but we found no evidence of non-extension scripts executing within isolated worlds. We define this functionality in the `Document::isCallerExtension` method (see Listing 25 in Appendix B).

4.5.1 Determining The Extension

Assuming that code executing within the context of an isolated world is defined by an extension's content script, the challenge is to then determine which extension in particular is executing said code.

Fortunately, Chrome's extension system implements a general CSP concept [26]. Specifically, code executing within an isolated world must abide by the world's associated CSP. For example, resources (e.g. remote scripts or images) embedded into the main world's DOM or network requests send from within an content script executing within an isolated world are restricted based on the isolated world's DOM, not the main worlds [27].

In order to achieve this, each isolated world is associated with a security origin, containing a URL, and a CSP. For extension content scripts, the isolated worlds' security origin URLs contain the extension's identifier[3]. We use these security origin URL to retrieve the current content script's extension object, which provides meta-data about the extension including its name and icons, by looking it up in a registry of enabled extensions (see Listing 9).

If our assumption that all code executing within isolated worlds has been defined by a extension's content script proves incorrect, we could easily amend this by checking if there is an associated extension for the isolated world's security origin.

```
const Extension* extension = ExtensionRegistry::Get(source->GetBrowserContext())
                             ->enabled_extensions()
                             .GetExtensionOrAppByURL(security_origin_url);
```

Listing 9: A code snippet that retrieves an `Extension` object, containing extension meta-data, from an isolated world's security origin URL.

4.5.2 Tracking Origin World

Unfortunately, there are many ways for a browser extension to execute a script within page's main world and not within the extension's isolated world[4] [27] (§4.5.3). By just inspecting the world in which Blink code is being executed, we cannot guarantee that a browser extension did not create the code.

[3] An example security origin for an extension's isolated world might be `chrome-extension://hfaagokkkhdbgiakmmlclaapfelnkoah`.

[4] Some operations are not affected by this. For example, network requests executed within the page's main world must abide by the CSP of the page's origin.

Fortunately, we can solve this problem by keeping track of a script's original world. Chromium maintainers provided the foundations for solving this issue by tracking the origin world via a static stack of worlds (i.e. a `Vector<RefPtr<DOMWrapperWorld>>&`)[5] [40]. Their patch makes changes to Chrome's V8 JavaScript engine and works in the following way:

1. Before executing a script in another world (i.e. the main world or another extension's isolated world), we first push the current origin world onto an origin world stack.

2. In order to determine whether code execution is a result of an extension, we check if the current world is an isolated world. If not, we look at the top world on the origin world stack to determine if the current world originated from an isolated world. Instead of calling the `isIsolatedWorld` method in Listing 25, we instead call the `isOfIsolatedWorldOrigin` method defined in Listing 26 in Appendix B.

   If there are multiple extensions running, it is possible for the origin world stack to contain multiple different isolated worlds. For example, say extension A inserts an `iframe` element into the page. The `iframe` will have extension A's isolated world, call it A', as its origin world (the last world on the origin world stack). If another extension, say extension B, inserts a `script` element into this `iframe`, the `script` will have an origin world stack of $\{A', B'\}$ where B' is the last element pushed onto the stack. As B wrote the `script` executing in the `iframe`, we would like to blame them for potentially malicious operations, and therefore we consider B' (the last world on the origin world stack) to be responsible.

3. After executing a script, we pop the origin world off the stack. In order to do this without the potential error of forgetting to pop an origin world off the stack (step 3), we use a `DOMWrapperWorld::-OriginWorldScope` object to push and pop origin worlds from the stack on object construction and destruction respectively (see Listing 27 in Appendix B).

4.5.3 Script Injection Examples

In this subsection, to substantiate work involved in implementing the origin world stack concept, we give examples of ways malicious extensions could execute scripts within the main worlds.

`script` Elements

The script sourced from each `script` element is executed within the main world (by calling the `ScriptLoader::-executeScriptInMainWorld` function). As a result, the most basic way an extension can execute a script within the main world is to insert a new `script` element[6] with the malicious code (see Listing 10). The solution is to push the origin world onto the origin world stack in `ScriptLoader::doExecuteScript`.

This solution also propagates the origin world when a content script creates a `new Function` object and applies the function, or when a content script calls the `eval` function.

[5] Although this was suggested, this patch was never accepted into the Chromium project as it did not cover all script injection cases.

[6] Even though the malicious actions would not be intercepted and trigger a user prompt, we still intercept the operation of inserting a new `script` element (§3.7).

```
// This content script code is executed in an extension's isolated world
var s = document.createElement('script');
s.textContent =
    // This code will be executed in the main world!
    "var targetElement = document.getElementById('target-id');" +
    "targetElement.click();"
document.body.appendChild(s);
```

Listing 10: Executing a script in the main world by inserting a `script` element.

`Event` Handlers

Even if we keep track of the origin world when inserting `script` elements into the DOM tree, a malicious extension can execute code within the main world by registering the code as an event listener as this is run in a separate world (see Listing 11). Similarly to before, the solution is to push the origin world onto the origin world stack before handling an event in `V8AbstractEventListener::handleEvent`.

```
// This content script code is executed in an extension's isolated world
var s = document.createElement("script");
s.textContent =
    // This code will be executed in the main world!
    "document.addEventListener(\"DOMContentLoaded\", function() {" +
        // This code will be executed in the main world!
        "var targetElement = document.getElementById('target-id');" +
        "targetElement.click();"
    "});";
document.body.appendChild(s);
// Fire a white-listed event to trigger the Event Handler
document.dispatchEvent(new Event("DOMContentLoaded"));
```

Listing 11: Executing a script in the main world using event handlers.

Timers

The `window.setInterval` and the `window.setTimeout` functions create scripts in the main world (see Listing 12) and execute them periodically or after a given time period respectively. We solve this by pushing the origin world onto the origin world stack before executing the script in the `DOMTimer::fired` method.

```
// This content script code is executed in an extension's isolated world
var s = document.createElement('script');
s.textContent =
    // This code will be executed in the main world!
    "window.setTimeout(function() {" +
        // This code will be executed in the main world!"
        "var targetElement = document.getElementById('target-id');" +
        "targetElement.click();"
    "}, 1000);";
document.body.appendChild(s);
```

Listing 12: Executing a script in the main world using timer callbacks.

`MutationObserver` Handlers

Scripts can register functions to execute when certain DOM elements are mutated by creating `new MutationObserver` objects. These scripts are executed in the main world (see Listing 13). Unlike `script` elements, event handlers or timers, `MutationObserver`s were not covered by the Chromium patch that attempted to introduce origin world tracking. However, we solved this by pushing the origin world onto the stack within the `V8MutationCallback::call` method using the same pattern as before.

```
// This content script code is executed in an extension's isolated world
var s = document.createElement('script');
s.textContent =
    // This code will be executed in the main world!
    "var target = document.getElementsByTagName(\"p\")[0];" +
    "var observer;" +
    "observer = new MutationObserver(function(mutations) {" +
        // This code will be executed in the main world!"
        "var targetElement = document.getElementById('target-id');" +
        "targetElement.click();"
    "});" +
    "var config = { attributes: true, childList: true, characterData: true };" +
    "observer.observe(target, config);" +
    // Trigger the MutationObserver
    "target.setAttribute(\"data-random-attribute\", true);";
document.body.appendChild(s);
```

Listing 13: Executing a script in the main world using a `MutationObserver`.

`Promise` Handlers

JavaScript `Promise`s are used for asynchronous computation. Similarly to `MutationObserver`s, scripts can be run in the main world asynchronously using `Promise`s (see Listing 14).

```
// This content script code is executed in an extension's isolated world
var s = document.createElement('script');
s.textContent =
    // This code will be executed in the main world!
    "Promise.resolve().then(function() {" +
        // This code will be executed in the main world!
        "var targetElement = document.getElementById('target-id');" +
        "targetElement.click();"
    "});";
document.body.appendChild(s);
```

Listing 14: Executing a script in the main world using a `Promise`.

4.5.4 General Solution

The main issue with this solution is that we must push the origin world onto the origin world stack before executing any other injected script (e.g. script tags, timers, event handlers, etc.). If the Chromium browser ever implements another method to run scripts (either synchronously or asynchronous), developers must remember to push the origin world onto the world stack before executing the script. If this is not done, extensions can exploit this new method to run malicious code within the main world. For this reason, the Chromium patch that suggest these changes was ultimately rejected.

To try and solve this problem, we added the code to push the origin world onto the world stack in `V8ScriptRunner::callFunction` which is called whenever a timer, event and mutation observer fires or whenever a script is injected or promise returns. By doing this, we can remove the origin world related code from many different Blink and V8 engine classes, greatly reducing the code complexity. Additionally, the origin world of code executed within the handlers of a JavaScript `Promise` is correctly set using this general solution.

Limitations

Unfortunately, this introduces a new problem. Timers fire (and execute the provided callback function) within the main world even if they are created in a world with an isolated world as its origin world. This was solved in the Chromium patch by storing the origin world as a private field when first creating the timer, and pushing it onto the stack when the timer fires instead of pushing the current origin world onto the stack when the timer fires. As a temporary workaround, we currently push the origin world onto the stack both in the V8 engine's `V8ScriptRunner::callFunction` method and individually for each JavaScript construct that executes scripts in the main world.

As seen with the Timers dilemma, we cannot truly claim that our project guarantees that extensions cannot evade detection. Instead, our project provides a framework for how to guarantee this and it requires the origin world to be pushed onto the origin world stack wherever a script can execute another script in the main world. We believe that we have handled the majority of cases but we leave this up to the implementers of Chromium to guarantee.

4.6 Summary

In this chapter, we describe a project requirement that involves determining whether a page script or extension content script is executing a given method in the Blink engine. Without the ability to do this, we would be unable to differentiate between extension operations and page script operations and subsequently we would not be able to intercept suspicious extension actions without also intercepting page actions[7].

We present two solutions that we considered. The first approach consists of comparing the ordering and frequency of dispatched JavaScript events to determine if a user or script triggered them. Unfortunately, this would only distinguish between user and script events in general. The second approach is inspired by the PHP Aspis tool [32] and involves transforming extension content scripts to taint suspicious actions. This is not possible as we consider extension developers as adversaries. Neither provided the security guarantees that we aim to achieve and were vulnerable to evasion by sophisticated attackers.

We propose a novel approach to the specified problem that determines if script execution is occurring within a content script by inspecting the current context's world. We use this technique to retrieve the extension in question by using the isolated world's security origin. Finally, we integrate and extend an unapproved Chromium patch to guarantee sophisticated attackers cannot evade detection by inserting new script elements.

[7] Web-page's and the scripts they initially load are assumed to be benign (§3.1).

User Prompts

Existing work has analysed extensions executing within sand-boxes or emulators for suspicious or malicious activity [5, 6, 7]. Most projects researched classify extensions or plug-ins as malicious and either directly remove them from or notify the Chrome Web Store (CWS). Often these tools would classify extensions as suspicious and utilise human experts to verify malice.

We present a novel approach, with respect to browser extension security, that involves user interaction. Our project alerts browser users at suspicious extension activity as it occurs and provides them with the necessary tools to allow or prevent extension actions.

In this chapter, we first discuss the rationale behind preventing malicious extensions and requiring users to classify suspicious extension actions as malicious or benign. We explain our choice behind using browser pop-ups to convey information and present our implementation and design choices. Finally, we describe work undertaken to better describe and convey extension actions to everyday users so that they can determine malice themselves.

5.1 Preventing Malicious Extensions

After deciding to analyse extensions at run-time (§3.2), we were presented with the problem of handling them. In order to guarantee the security of browser extension users, we decided to prevent any suspicious actions classified as malicious and allow ones classified as benign.

An alternative model we considered involved sending suspicious extension actions to a remote command center to be analysed by human experts while gracefully allowing the action to execute. However, this approach is likely not feasible as the quantity of suspicious extension actions that are triggered would result in an impractical human workload.

Exceptions

Whenever a suspicious extension is prevented, we throw a run-time exception that extensions could handle so to not completely break functionality. In order to minimise changes to exciting Web APIs, we only throw exceptions in Blink methods that are currently defined to throw exceptions.

We throw security exceptions, similar to those thrown when network requests are rejected due to Content Security Policies (CSP), with messages which explain that the user prevented the action. For example, an exception message could be "Refused to connect to `https://evil.com` because a user has prevented it.", if a user prevented an extension from sending a network request to the mentioned URL.

5.2 Classifying Suspicious Extension Actions

On deciding to prevent malicious extensions, we were presented with the challenge of classifying suspicious extensions as malicious or benign. We briefly considered three possible methods.

5.2.1 Delegating Responsibility to Web-pages

The first involved alerting the web-page on which an extension content script is executing, that a suspicious operation is occurring. This would delegate the responsibility of determining whether suspicious operations are malicious or benign to the web-page. The rationale being that web developers would have the best context on what behaviour is malicious on their web-page. Unfortunately, this solution presented us with various non-trivial problems.

In order to prevent malicious operations from occurring, we would need to block content script execution until we decide if the suspicious action is malicious or benign (§5.3). If we delegated the responsibility of determining malice to web-pages, we would need quick and precise answers so to not break extension functionality. This may be difficult, as potentially malicious extension background pages have the ability to intercept[1] and read network requests. Additionally, web services may not have the resources to precisely detect malice in extension operations and subsequently some may be more vulnerable to attack than others.

Additionally, web-pages may have an incentive to prevent benign extensions (e.g. ad blockers) from executing on their web-pages. As web-page intentions could be adverse to our project goals, delegating this responsibility to them may result in broken benign extension.

5.2.2 Statistical Model

We also considered building a statistical model similar to the WebEval tool [5] that we would analyse suspicious extension operations at run-time. The main limitation is that the analysis must have a very high precision and recall when analysing operations, which may be disguised by attackers, in order to not break benign extensions or let malicious ones slip by. If not done correctly, this analysis would fail to provide adequate security guarantees due to many false negatives and would likely break many existing benign extensions due to many false positives.

5.2.3 Delegating Responsibility to Users

An alternative solution, which we settled on, involved delegating the responsibility of deciding if an extension action is malicious or benign to human users at run-time. Users would ideally act in their best interest, allowing benign extension actions and preventing malicious ones. By requiring user involvement, we would also be distributing the work load of classifying suspicious operations across millions of browser extension users. This was inspired by WebEval and Hulk's use of human experts to verify claims of extension malice [5, 6].

If it is not possible to guarantee malice or the lack of, users are the most apt to allow or prevent suspicious operations based on their comfort with risk. Users that are prompted to classify suspicious extensions

[1] Assuming they have permission to use the `chrome.webRequest` API [41].

could also account for the sensitivity of the information or functionality of the web-page the extension content-script is currently executing within. For example, a user may chose to be more risk-adverse and prevent an extension attempting to perform a network request on a web-page with sensitive user data (e.g. medical records) even if they are not certain it is malicious. On the other hand, it would be challenging for a model or third-party to analyse the sensitivity of information without exposing personally identifiable or sensitive data.

Limitations

By delegating the responsibility of classifying extensions to users, we in fact relax the security guarantees that we provide. Instead of protecting users from malicious extensions in a way that cannot be evaded by sophisticated attackers, we guarantee that we will notify a user whenever an extension attempts to perform a suspicious operation.

A limitation of this approach is that unlike previous work that used human experts to validate extensions classified as malicious, we delegate this responsibility to everyday users who are most likely not experienced with web or browser security concepts. We imagine everyday users would be more risk adverse and prevent a majority of suspicious actions, thereby breaking many existing benign extensions.

5.3 Return Values

Currently, the majority of Blink methods we consider suspicious return values to users. For example, the `Element.getAttribute` method returns the element's associated value for the attribute name provided.

User dialogs that are displayed when we intercept suspicious extension actions take a long time to be submitted when compared with the original execution time of the method. As a result, we were faced with the challenge of effectively returning values to suspicious method callers when the method triggers a warning dialog. We briefly considered two alternative solutions.

5.3.1 Callback Methods

The first approach involved re-defining all suspicious methods to not return values but instead execute callback methods provided by callers. If a suspicious method triggers a user dialog, we would wait until the dialog is submitted and then subsequently execute the callback method. Otherwise, we execute it immediately. Extension developers could leverage this approach to perform additional computation while dialogs are displayed. Finally, web-page scripts would continue to execute unaffected while user dialogs are displayed.

Unfortunately, this approach would require extensive re-factoring of suspicious methods and would ultimately break many existing Chrome extensions. Furthermore, existing web standards would need to be modified and this would reduce the feasibility of this project. We therefore decided against this solution.

5.3.2 Blocking JavaScript Execution

The second approach which we settled on involves blocking JavaScript execution while warning dialogs are displayed. Although this incurs a very large performance overhead when suspicious extension actions

are intercepted, fewer existing extensions would likely break and no modifications would be needed Blink methods or web standards.

5.4 JavaScript Dialogs

In order to add only minimal changes to the Chrome browser, we explored prompting users to allow or prevent extension actions using existing dialogs. Currently, scripts can create dialogs using the `alert`, `prompt` and `confirm` methods. The latter two display dialog boxes and either prompt users for an input string or prompt them to click an "OK" or "Cancel" button. These existing dialog methods block execution until users submit or close them, which fulfills our specification.

As we want users to have the choice to either allow or prevent suspicious extension actions, we considered using the more apt `confirm` method. By using an existing dialog template, we could avoid defining and sending additional Inter-Process Communication (IPC) messages from the renderer process to the browser process.

Furthermore, by using standardised methods to create dialogs, our modifications to the Chrome browser would more easily be applied to other browsers. After all, other browsers would need to only change their web-page rendering engines (e.g. Blink and WebKit) and not other components that render browser-specific views.

5.4.1 Limitations

Without extensive modifications to existing dialog methods, we were presented with two main disadvantages. This influenced our choice to abandon this approach.

Firstly, to prevent users from being spammed by malicious web-pages, Chrome allows users to suppress alerts and other pop-ups by displaying a "Prevent this page from creating additional dialogs." checkbox within dialogs. Although we too do not want to spam or inconvenience users, by using existing methods, we would not be in control of when to permit users to suppress dialogs. Moreover, we explicitly do not want suppression of dialogs created by potentially malicious scripts to be coupled with the suppression of our high-priority warning dialogs. That is, if a user suppresses many dialogs created by other scripts, we should not suppress suspicious extension action dialogs.

Secondly, with existing dialogs, we have no control over their user interface. Ideally, we would want to display informative buttons and extension icons so that extensions would be easily identifiable. Additionally, in order to reduce user confusion, our high-priority warning dialogs should be distinguishable from dialogs created by malicious web-pages, scripts and extensions.

5.5 Custom Dialog

Instead of using existing JavaScript dialogs, we decided to build our own custom dialog to convey specific information. Similarly to other Chrome extension pop-ups (see Figure 2.3), we wanted to display the extension's icon and name for easy identification, as well as the suspicious action it is attempting to execute and the page on which it is trying to execute it on. We also borrowed inspiration from Little

Snitch's, a host-based firewall application for OS X [42], permission dialogs which prompt users to allow or deny application network requests (see Figure 5.1) and modified the default dialog buttons from "OK" and "Cancel" to "Allow" and "Prevent" respectively[2].

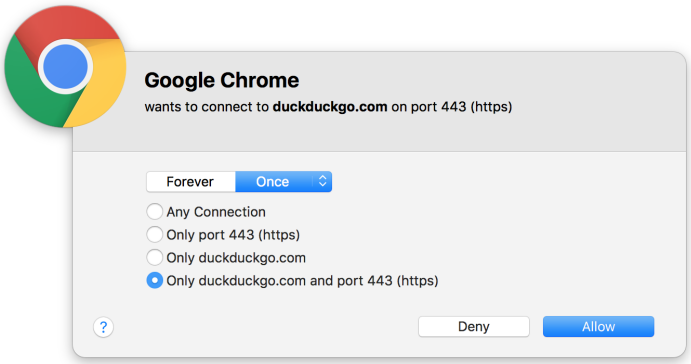

Figure 5.1: A pop-up dialog Little Snitch displays when Google Chrome attempts to connect to the `duckduckgo.com` server.

Although our design met the criteria above (see Figure 5.2), we believe experiments to improve the interface and wording for everyday, non-technical users are necessary and we leave these for future work.

Furthermore, in order to break the fewest number of extensions as possible, we set the default behaviour (and button) of our dialogs to allow extension actions. As a result, whenever dialogs are dismissed via the close button we implicitly allow the extension action. This is different to Little Snitch's implementation whereby they prevent network requests by default.

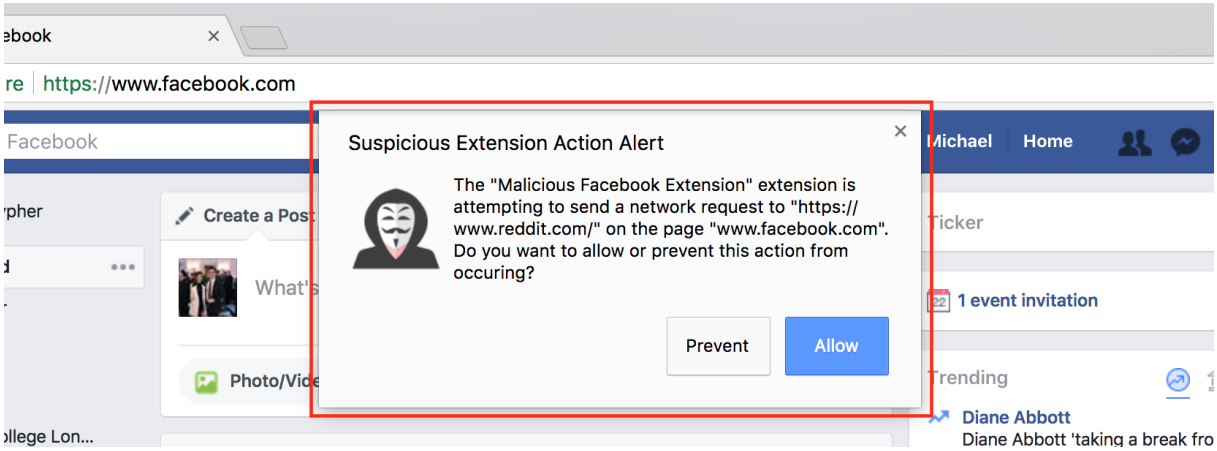

Figure 5.2: The default dialog displayed to users when an extension attempts to execute a suspicious action.

5.5.1 Implementation

The main challenge when implementing custom dialogs was determining how to structure IPC messages between the renderer process, that would intercept the suspicious extension action, and the browser processes, that would render the dialog. A requirement was the renderer process should block until it receives a response from the browser process.

[2]We choose to use "Prevent" instead of "Deny" as the cancel button text but being synonyms, these are interchangeable and we should experiment with both in the future.

As we had access to the security origin URL from within the renderer process, we have the choice to either retrieve the extension from within the renderer or browser process (§4.5.1[3]). In order to minimise the size of IPC messages (see Listing 15), and avoid the performance overhead and serialising `Extension` objects or sending large messages, we chose to look-up the extension within the browser process.

We also decided to build the part of the dialog description string that describes the suspicious extension action (e.g. "send a network request to "`https://www.reddit.com/`"" in Figure 5.2) within the intercepted action in the render process. The aim was to reduce coupling between Blink methods and our dialog as much as possible and to let future developers add information to warning messages without needing to increase the IPC message argument list.

Each message also contains `enum`s representing an extension action group (§3) and the sensitivity of the element on which an operation is performed on (§6.4). We pickle (serialise) these `enum`s as 16-bit unsigned integers[4] to reduce the size of IPC messages. Each message has an output Boolean field that is set to `true` if users allow the action (either implicitly, if the browser remembers their decision, or explicitly if they click the "Allow" or "Close" buttons) and `false` otherwise.

```
// A request to display a prompt dialog, asking for permission to perform
// extension action
IPC_SYNC_MESSAGE_ROUTED5_1(FrameHostMsg_DisplayExtensionActionPrompt,
                           GURL                       /* in - extension url */,
                           base::string16             /* in - prompt message */,
                           SuspiciousExtensionAction /* in - extension action */,
                           SensitiveElement           /* in - sensitive_element */,
                           GURL                       /* in - originating page URL */,
                           bool                       /* out - success */)
```

Listing 15: The IPC message sent from the renderer to browser process requesting that suspicious extension action dialog is displayed.

We follow a similar architecture and implementation to JavaScript dialogs and block script execution in the renderer process by delaying the reply from the browser process until users submit the associated dialog. This is implemented by using the `IPC_MESSAGE_HANDLER_DELAY_REPLY` macro. We also prevent other tabs in the same renderer process from processing user input events by executing `GetProcess()->SetIgnoreInputEvents` before and after we display the dialog.

Whenever we intercept a suspicious operation and determine that the operation is being executed by an extension, we call `Document::userAllowsSuspiciousExtensionAction` which sends the IPC message to the browser process (see Listing 28 in Appendix C). Whenever a user closes the dialog, we set the IPC message `success` variable to `true` or `false` and send a response.

5.6 Highlighting Elements

A fundamental part of this project relies on an everyday user's ability to correctly classify suspicious extension actions as benign or malicious. If they incorrectly classify malicious extension operations as benign, they may be vulnerable to attacks. On the other hand, if they incorrectly classify benign actions as malicious, they may break the functionality of many existing extensions. As a result, an indirect goal

[3]The `ExtensionRegistry` used to retrieve extensions can only be accessed within the browser process but has a counterpart, aptly named `RendererExtensionRegistry`, that can be accessed within the renderer process.

[4]This is the smallest possible type we could serialise. A possible extension would be to serialise both parameters together within this integer.

of this project was to convey as much useful information about extension actions as possible so that users can make informed decisions.

The first feature we considered adding in order to provide more information about extension actions involved "highlighting" the element on which an extension operation executes (if an element exists at all). Chrome's DevTools [43] currently provides similar functionality whenever developers hover over elements within the developer tools tab (see Figure 5.3). This idea is also similar to how BrowserFlow [34] highlights (changes the background colour of) text segments to alert users of accidental data disclosure.

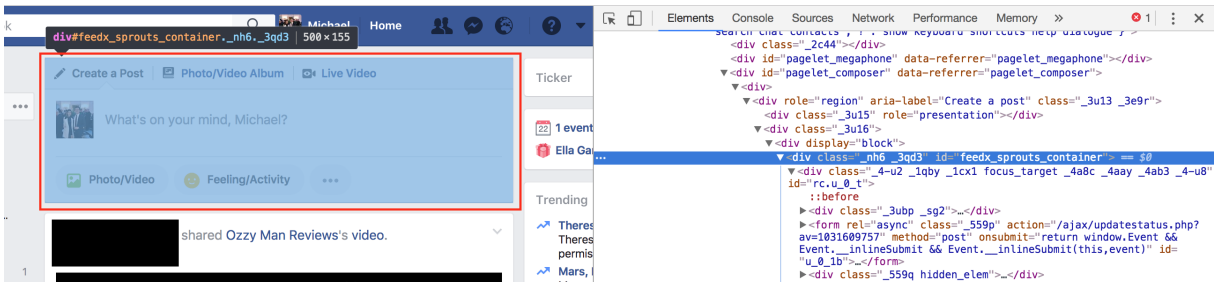

Figure 5.3: An example of a page element highlighted due to Chrome's DevTools [43].

We proposed a simple design in which an element is highlighted just as an suspicious action dialog is display and un-highlighted when the dialog is closed.

5.6.1 Implementation

To highlight an element, we simply inserted another element the same size, with a transparent yellow background color, and positioned it on top of the original element.

However, the main challenge when implementing this feature was updating the page within the Blink method executed by an extension. As mentioned in Section 2.1.3, the main thread must block while page changes are committed to the compositor thread and drawn to the screen. To prevent blocking the main thread often, updates are not committed to the compositor thread during Blink method calls (e.g. when calling the `ContainerNode.appendChild` method). Instead, Blink methods that mutate the page mark it as "dirty" by calling the `SetNeedsCommit` method. This notifies the compositor scheduler (CCScheduler) that the layer trees between the main thread and compositor thread are no longer consistent and that it should request a new commit[5] [17].

If we simply added a highlight element, blocked while the warning dialog was displayed and then removed the highlight element, the user would never see it as the page changes would only be flushed to the compositor thread after the method returned. A solution would be to synchronously pump all of the changes to the compositor thread after adding a highlight element and after removing it[6] but unfortunately Chromium provides no easy mechanism to do this.

We discovered the `Document::RequestAnimationFrame` method which appears to commit changes to the compositor thread and then executes a callback method provided as an argument. A unique solution would be to split every suspicious Blink method in two separate methods. The first method would insert

[5] In fact, the CCScheduler considers the overall system state and may delay a commit due to a variety of reasons including the last screen draw and the time of the next frame.

[6] This second pump is technically not necessary as we could just wait for the CCScheduler to request a commit after the Blink method returns

a highlight element into the DOM, if the original method is executed by an extension, and subsequently call `Document::RequestAnimationFrame` with the second method provided as the callback argument.

We experimented with modifying the `EventDispatcher::dispatch` method as described above but faced many non-trivial run-time errors. We ignored the `EventDispatcher::dispatch` method's return value as we had yet to figure out how to return the correct value via the `Document::RequestAnimationFrame` callback method. This may explain the errors encountered but further investigation is needed.

We managed to modify the `Element.setAttribute` method without run-time errors but unfortunately the page changes were not being pumped to the compositor thread before the callback passed to `Document::RequestAnimationFrame` executed. This resulted in the element under question being highlighted only after the warning dialog was dismissed. We also experimented with executing `Element::SetNeedsCompositingUpdate` and similar methods after inserting the highlight element without success. This might suggests error in our understanding or use of this method and at the time of writing, we have not yet correctly implemented this feature.

5.6.2 Limitations

Although unsuccessful, our proposed approach of waiting for the compositor thread to update the screen with a highlight element before displaying the warning dialog could greatly increase the performance overhead for suspicious operations. Furthermore, as we "remember user decisions" within the browser process (§6.3), we may display a highlight element and incur the overhead unnecessarily if extension action is implicitly allowed or prevented and the warning dialog not displayed.

Moreover, elements can be positioned off the page and can be extremely small so that they are not visible to users. For example, tracking pixels are usually one pixel in width and height, and are designed to be invisible to users [44]. In these cases, highlighting the elements would provide little benefit but still incur performance hits. Potential solutions might include scrolling to the elements off the page and alerting users that the element is "hidden".

5.7 Describing Elements

As well as highlighting elements visually so that users can find them on the web-page, we wanted to give web-site providers the ability to describe elements themselves. We implemented this by using a custom `data-description` attribute which web-pages and extensions can change, but which only web-pages can remove (see Subsection 3.8.2).

When describing an extension action in a warning dialog, we describe the element on which the action was performed as "an element" if there is no description. If the web-site provider has specified an alternative description via the `data-description` attribute, we use this instead.

The main limitation of this approach is that we must rely on web-pages to provide informative and correct descriptions for all elements that may be involve in a suspicious operation. Furthermore, these descriptions may need to be updated when elements are mutated by the page and would need to be localised for the user's browsers.

5.8 Additional Information

During testing, we noticed that our warning dialogs do not give enough information about suspicious extension actions for users - even those experienced with browser security - to classify them as malicious or benign. We experimented with adding additional information (e.g. the old and new values of the element attribute being changed) to the warning dialog but this resulted in our the dialogs becoming too complex for everyday users.

We begun implementing a feature, similar to Little Snitch's (see Figure 5.4) and Chrome's "install extension" dialog view, to display additional information for each suspicious action in a separate expandable view. Due to time constraints, this feature has not been completed but we believe it would adequately solve the complexity issue described above.

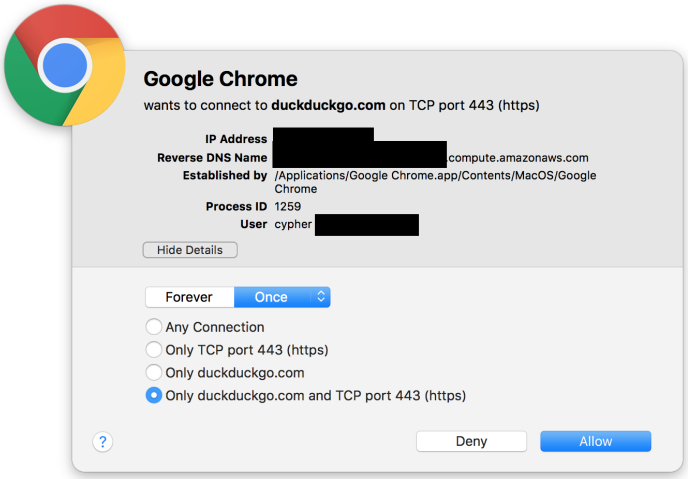

Figure 5.4: A pop-up dialog Little Snitch displays when Google Chrome attempts to connect to the `duckduckgo.com` server with additional information displayed. Personally identifiable information has been hidden.

5.9 Summary

In this chapter, we choose to prevent suspicious extension actions classified as malicious and allow ones classified as benign. We consider using statistical models to classify suspicious actions and delegating the responsibility of classification to web-pages, but decide against both due to weak security guarantees and potentially adverse intentions respectively.

We compare two methods of dealing with return values that involve re-defining Blink methods to take callback methods or simply blocking JavaScript execution while classifying suspicious operations. The latter is choosen due to simplicity and minimal impact on Web API standards.

The advantages and disadvantages of using existing JavaScript dialogs are evaluated and we decide to instead build our own custom warning dialog. This allows us to display the extension's icon and make buttons more informative but requires new IPC messages between the renderer and browser processes to be defined and handled.

Finally, we propose multiple features, including highlight elements, description attributes and additional information views. These aim to provide users with useful information which they can consider when classifying extension actions.

Reducing Dialogs

Our modified browser alerts users of suspicious operations that an extension attempts to perform and shifts the responsibility of classifying these actions as malicious or benign to the user. This responsibility, similar to security in general, is inherently inconvenient for everyday users. We believe that even security-conscious users will resort to allowing or preventing all suspicious extension actions if they are frequently interrupted with warning dialogs.

As a result, an indirect goal of this project was to filter out obviously benign suspicious operations and provide mechanism by which users can reduce the quantity of dialogs they are presented with. In this chapter, we present techniques employed to achieve this goal by ignoring benign events, filtering out operations on elements not attached to the DOM tree, remembering user's decision to allow or prevent actions and by introducing the concept of sensitive DOM elements.

6.1 Suppressing Common Benign Events

While manually testing our modified browser, we noticed extensions were triggering many seemingly benign events. More specifically, every extension we analysed implicitly triggered[1] `readystatechange`, `DOMActivate`, `DOMContentLoaded` and `load` events when they were injected and loaded within a web-page's context. Although these events were routinely dispatched, they did not appear to pose a threat to users and provided little information about their purpose, which would likely confuse everyday users. Therefore, we decided to white-list these events and allow them to be dispatched by a content script without user permission.

Our current implementation of this white-list is naive and utilises a set of white-listed event-type strings which we allocate on the stack for each dispatched event. A better solution would involve comparing the event in question against a static constant bitfield to reduce the performance and memory overhead.

6.2 Allowing Operations On Unattached Elements

While analysing existing extensions, we also discovered that most extensions create an element and append to it other child elements before inserting it into the page's DOM tree. This prevents the element being rendered on the page before it is finished being constructed (i.e. before its children are inserted).

Our modified browser regularly notified us that an extension was attempting to append a child to an element despite that element not being connected to the page's DOM tree. As we have no evidence demonstrating attacks being carried out by mutating or reading elements that are not attached to the

[1] The Chrome browser triggers these events without content script input.

DOM, we chose to always allow them. This was implemented using a simple `Element.isConnected` conditional check before intercepting operations.

The only exception to this condition is the `Document.createElement` method, which we don't intercept by default (§3.7), and which never produces an element that is connected to the DOM tree. In this case, if a user configures our modified browser to intercept operations that create elements (§3.10), we intercept them despite the created elements not being attached to the page.

6.3 Remembering User Choice

In order to not prompt users with redundant warning dialogs, we followed the practice[2] of allowing users to request that the browser remember their decision via checkboxes. If a user opts into always allowing or preventing a certain action, we no longer display warning pop-ups for that action and instead implicitly allow or prevent it respectively. By reducing the number of duplicate warning messages that users are presented with, extension functionality become more consistent and users can focus on unique warning messages.

Our dialogs also give users the option to always allow actions performed on non-sensitive elements (§6.4).

6.3.1 Similar Suspicious Actions

This feature's main design decision was how we determine if suspicious actions have been pre-approved or pre-prevented by users. Specifically, the challenge was deciding how to compare suspicious extension actions.

The first solution we explored was to compare the suspicious action extension group. These would include DOM operations, network requests and more (see Listing 21). For example, if a user decides to always allow a network request between a content script and `www.reddit.com`, we would also allow the extension to send network requests to `www.evil.com` in the future without user permission. This choice is obviously too broad as the destination of each network request (in the previous example) is different, yet the operations are considered the same. As users are very likely to factor in the network request's destination address when deciding to allow or prevent it, this solution was deemed unacceptable.

Another solution would be to only regard suspicious operations as equal, with regards to this feature, if the state of the DOM tree and operations are identical. For example, if a user decides to always allow a network request between a content script and `www.evil.com`, but then adds sensitive information to the page, we would display a dialog the next time the extension attempts to send the same network request. This choice is obviously too fine-grained as everyday users lack the ability to inspect the details of the page or operation for differences. Since pages are frequently mutated, users would almost always be prompted with dialogs despite choosing to "Always allow" or "Always prevent" actions and this would defeat the purpose of this feature.

Another issue with the previous two approaches is that users are not aware of how extension actions are compared. To solve this, we chose to regard two extension actions as equal, with regards to remembering extension actions, if their associated dialogs would display the same warning text.

[2] In particular, the Little Snitch application also provides this functionality (see Figure 5.1).

6.3.2 Implementation

We implement this feature using an in-memory map from user dialog text to a user's preference to either always allow, always prevent or always display dialogs for certain actions (i.e. `std::map<base::string16, RememberPreference>`). This was done, as opposed to using an extension state store (§3.10), so that a user's remembered actions are reset every time they re-launch the Chrome browser. We chose to do this as users currently have no way to modify previously made decisions.

Since content scripts can be loaded within the same web-page within multiple tabs or windows which can each execute within a different renderer process, our map resides in the browser process. The main limitation with this approach is that the renderer process must still send Inter-Process Communication (IPC) messages to, and block until it receives a response from, the browser process. This occurs even if the action will be implicitly allowed or prevented. An unfortunate consequence of this decision is that every suspicious extension action must communicate across process boundaries and this adds considerable performance overhead.

Checkbox Finite State Machine

In order to make our user interface easier to use for everyday users, we spent time tweaking it and imposed certain restrictions on it. The finite state machine in Figure 6.2 represents these restrictions.

If the "Always allow" or "Always allow for non-sensitive elements" checkboxes are selected, the "Prevent" button is disabled (see Figure 6.1). Likewise, if the "Always prevent" checkbox is selected, the "Allow" button is disabled.

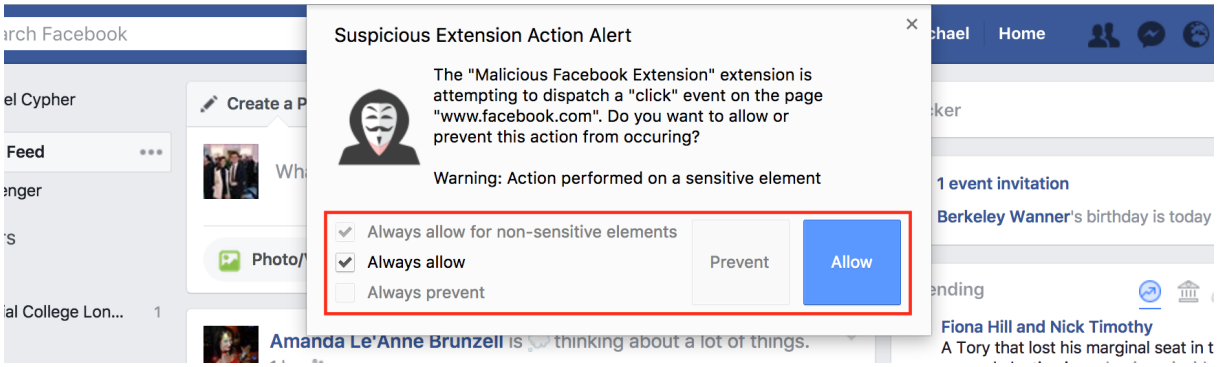

Figure 6.1: Whenever the "Always allow" or "Always allow for non-sensitive elements" checkboxes are selected, the "Always prevent" checkbox and "Prevent" button are disabled.

Furthermore, if a user selects the "Always allow" checkbox, the "Always allow for non-sensitive elements" checkbox is checked and disabled (see Figure 6.1). We chose to do this because always allowing a suspicious operation would allow it whether it is performed on a sensitive or non-sensitive element. Deselecting the "Always allow" checkbox re-enables the "Always allow for non-sensitive elements" checkbox and returns it to its previous state. We also disable and uncheck the "Always prevent" checkbox when either of the other two are checked and likewise, we disable and uncheck the other two checkboxes when the "Always prevent" checkbox is checked.

Finally, if an extension action has been allowed for non-sensitive elements but the current dialog is displaying the action for a sensitive element, the "Always allow for non-sensitive elements" checkbox is checked by default.

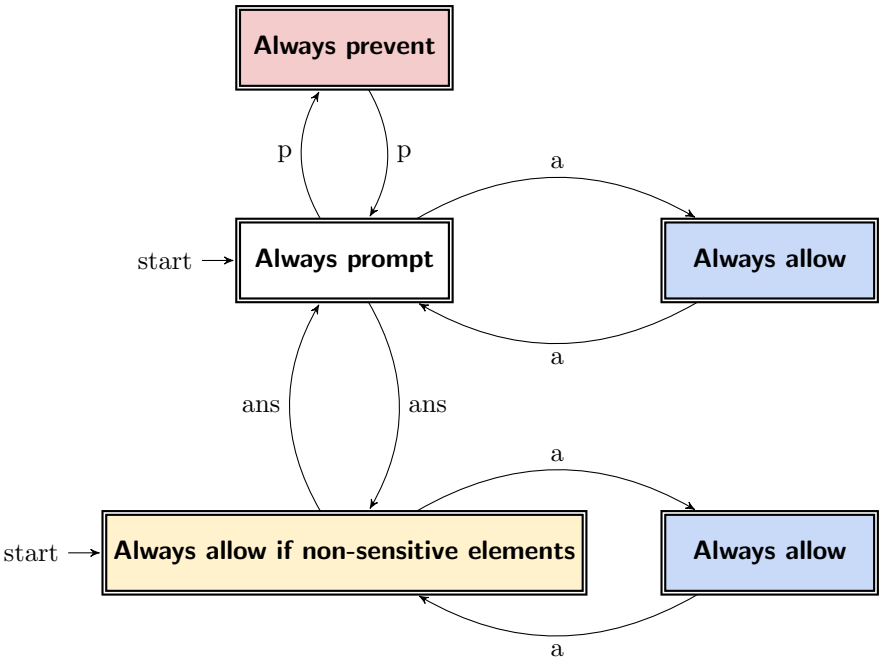

Figure 6.2: Checkboxes in our dialog are disabled and enabled by clicking on other checkboxes. This finite state machine represents that valid transitions. The *p*, *a* and *ans* transitions represent toggling the "Always prevent", "Always allow" and "Always allow for non-sensitive elements" checkboxes respectively. If a transition does not exist from a state, it means the associated checkbox which you would toggle is disabled.

6.3.3 Limitations

The main disadvantage with this feature is that a user's preference to always allow or prevent certain suspicious extension actions only lasts until the browser is closed. This occurs because our modifications persist this information in-memory and do not write it to disk. Ultimately, this inconveniences and potentially confuses users as they required to give permission to extensions actions every time they restart their browser.

6.4 Sensitive DOM Elements

To discourage users from dismissing important notifications, we must be careful not to flood them with alerts of benign actions. For operations that take DOM elements into account (e.g. events and DOM mutation), one option we considered was to only alert users if the element was deemed sensitive.

Unlike the Hulk tool [6], we decided against building a separate tool that classifies the sensitivity of elements for the following reasons:

- Elements might be mutated or created during web-page and extension content script execution. In order to keep the sensitive items on a page up to date, we would need to re-determine the sensitivity of each potentially modified element every time a mutation occurs. This would likely incur significant run-time overhead.
- In order to guarantee that an extension is not operating on sensitive elements, we would have to

correctly classify all sensitive elements as sensitive. If this results in incorrectly classifying many elements as sensitive (i.e. has a low precision and high false positive rate), users will soon ignore any mention of sensitivity as many seemingly unimportant elements would be considered sensitive. Building a tool with high recall and precision would be a difficult challenge in its own right and we leave this for future work.

Instead, we chose to give service providers, who have the most context and knowledge about data and functionality on the web-pages they provide, the ability to mark elements on their web-pages as sensitive. Furthermore, we consider any tainted element or any of its descendants as sensitive.

6.4.1 Changing Sensitivity

Unlike BrowserFlow's Text Disclosure Model [34] that allows users to change privilege labels [34], we also decided against allowing extensions or users to change the sensitivity of elements. If a malicious extension were to mark an element as not sensitive, this may result in fewer dialog pop-ups being shown to users. An attacker could use this to evade interception. On the other hand, if a malicious extension were to mark many elements as sensitive, more dialogs would be displayed with a sensitive element or data warning, which would dilute their importance.

6.4.2 Usage

As we cannot realistically ask service providers to begin tainting all sensitive elements on the web-pages they provide, we currently warn users about suspicious operations performed on any element, sensitive or not. We also chose to display additional information on eligible pop-up dialogs that informs users of the sensitivity of an element.

Additionally, when users are prompted to allow or prevent operations on non-sensitive elements, we let them always allow the operation for non-sensitive elements (see Section 6.3 and Figure 6.3). We do not give our modified browser the ability to always allow suspicious operations on sensitive elements as sensitive elements are more prone to misuse by definition and this therefore makes little sense.

On the other hand, it might make sense for users to be given the ability to prevent all operations on sensitive elements but not operations on non-sensitive elements (i.e. an "Always prevent for sensitive elements); we leave investigating this further for future work. We chose not to add this extra checkbox to prevent the dialog from becoming too cluttered and therefore more confusing for the everyday user.

6.4.3 Tracking Sensitive Data

Similarly to how we consider certain elements as sensitive and present this information to users, we also consider certain data originating from sensitive elements as sensitive. As sensitive data can leak out of the web-page via network requests[3] and since users may base their decision to allow or prevent requests on this information, we wanted to inform them on the sensitivity of information leaving their browser.

One option was to taint any data associated with sensitive elements in Chrome's V8 JavaScript engine and propagate it through the system similar to the work of Egele et al. [7]. Although this solution

[3]Sensitive data can also leave the web-page via chrome API methods, (e.g. `chrome.runtime.sendMessage`) but these are not currently intercepted and we therefore ignore these cases.

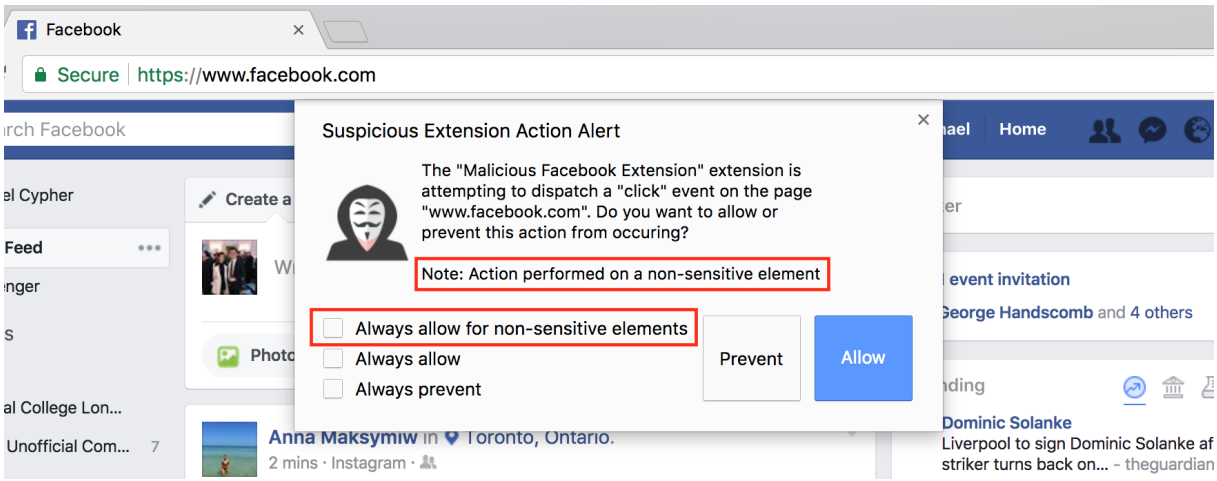

Figure 6.3: A suspicious action alert dialog that includes information about the sensitivity of the element and lets users request that the browser remembers their decision for non-sensitive elements.

would correctly propagate sensitive information through the system, it would likely result in a significant memory and performance overhead. Furthermore, we expect the implementation of this feature to take considerable time and since tracking sensitive data is not a high-level goal of this project, we opted for a simpler solution.

We took inspiration from the BrowserFlow plug-in [34] and decided to compare the data an extension is uploading in a network request with sensitive data on the page that has previously been read by any script. However, unlike BrowserFlow, our current comparison is *precise* in that we look for exact matches of sensitive data in network requests. We decided against *imprecise* data flow tracking as, according to research by Papagiannis et al., it does not work well comparing the similarity of small text segments, resulting in many false positives [34]. Most sensitive data (e.g. secret tokens, identifiers, passwords) will be smaller in length than an average paragraph and BrowserFlow text segment.

We considered only comparing the network request data with sensitive data that has been read by the extension sending the request, but this would expose a vulnerability. Other scripts or extensions could strip the sensitivity of data by leaking it from a sensitive to non-sensitive element. A malicious extension could read the non-sensitive data from the non-sensitive element and successfully send it in a network request without having Chrome alert the user of sensitive data leakage.

Limitations

This approach suffers from two large limitations that make it impractical in practice. Firstly, BrowserFlow's imprecise data flow tracking only worked because their threat model constitutes of *trusted-but-careless* users. In our threat model, we assume extensions to be malicious. A sophisticated attacker with knowledge of the data flow tracking model can encrypt sensitive data before uploading it via a network request, thereby evading our trivial sensitive detection. Secondly, our approach does detect sensitive data that has been slightly modified.

Furthermore, similar to the description attribute (§3.8.2), we only know an element's sensitivity if the web-site explicitly sets this attribute. For many small online companies, marking every sensitive element may not be feasible. Additionally, we also place responsibility on service providers to *correctly* mark elements as sensitive. Service providers that mark too few elements as sensitive may let malicious actions

go unnoticed, and service providers that mark too many elements as sensitive may risk flooding the user with benign action alerts. The latter would increase the false positive rate and reduce the likelihood the user spots a malicious behaviour.

Another disadvantage of our implementation is that we currently only use a binary model of sensitivity (i.e. elements are sensitive or not). However, this does not take into account the fact that elements are generally sensitive to only certain extension operations. For example, a "like" button may be sensitive to click events and reads of its form's action URL and secret tokens[4] but not be sensitive to style changes. Our approach does not take into account extension operations when determining the sensitivity of an element or its data.

6.4.4 Implementation

Similarly to element descriptions, we decided to use a custom, this time Boolean, attribute `data-sensitive` to mark sensitivity. We implemented a simple `Element.isSensitive` method that checks the current element's sensitivity or recursively checks its ancestors (see Listing 16). This implementation has a nice feature whereby service providers can mark non-sensitive elements within (i.e. as descendants of) sensitive elements by setting the sensitivity attribute to false (i.e. `data-sensitive="false"`). We also exposed the `Element.isSensitive` method to all scripts by adding it to the Elements interface (Web IDL).

```
// Returns true if the element is sensitive and false otherwise
bool Element::isSensitive() const {
  if (hasAttribute("data-sensitive")) {
    return true;
  }
  Element* parent = parentElement();
  if (parent != nullptr) {
    return parent->isSensitive();
  }
  return false;
}
```

Listing 16: The `Element.isSensitive` method returns `true` if the element in question is sensitive.

We used a static `HashSet` to keep track of sensitive data that has been read in the current renderer process. Whenever data is read from a sensitive element using the `Element.getAttribute` method, we add it to the set. In order to provide security guarantees about sensitive data read, we also wanted to mark element attributes as read whenever a script reads an element's raw HTML. If a script calls the `Element innerHTML` or `outerHTML` methods on a sensitive element, we mark all of the element's attributes as read and recursively do the same for all child elements that remain sensitive in a depth-first fashion using the `ContainerNode.markSensitiveDataAsRead` method (see Listing 29 in Appendix D).

The sensitivity of each suspicious action is sent, represented as an `ExtensionActionIsSensitive enum` (see Listing 17), from the renderer process to the browser process whenever we want to display suspicious extension action dialogs via an IPC message (§5.5).

Changing Sensitivity Warnings

We prevent any attempt by extensions to add or remove the sensitivity taint to or from elements. We currently do not inform the users of extensions attempting to do this but we propose this as a future

[4] Websites often attach secret tokens to forms to prevent Cross-Site Request Forgery (CSRF) attacks [38].

```
enum class ExtensionActionIsSensitive {
    TRUE,       // action performed on sensitive element
    FALSE,      // action performed on non-sensitive element
    NA          // action not performed on element (e.g. network request)
};
```

Listing 17: The `SensitiveElement enum` captures the possible sensitivities of an element.

extension. After all, extensions that attempt to mutate sensitivity are highly likely to be malicious and informed users may choose to uninstall them even if they are prevented from changing an element sensitivity.

6.5 Summary

In this chapter, we illustrate the challenge of reducing the number of dialogs with which users are presented. This problem, unsolved, would inconvenience users and consequently weaken their security as they would be less likely to correctly classify suspicious actions.

Furthermore, we outline many features implemented to alleviate the problem. The first two include implicitly allowing common benign events and allowing operations that are performed on elements not yet attached to the DOM tree. Neither appear to present prevalent threats.

We provide users with a tool to always allow or prevent certain suspicious extension actions with the aim of reducing the number of duplicate warning dialogs. We explore various solutions but choose to compare extensions based on their associated dialog warning messages.

Finally, a unique model of sensitivity is presented with which web-pages can mark elements as being more vulnerable to malicious extensions. Users are notified when suspicious operations act on or utilise suspicious extensions and their data. For the sake of simplicity, we adopt a precise data flow tracking model inspired by previous research.

Evaluation

In this chapter, we evaluate our modified Chrome browser against project objectives (§1.1). We describe the security guarantees we provide and evaluate our success in protecting users from malicious extensions by building one and analysing existing suspicious extensions available on the Chrome Web Store (CWS). A user survey is carried out to quantitatively determine the success of our user interface and popular existing benign extensions are analysed to understand how often suspicious operations are executed. Finally, Chrome benchmarks are compared to detect the performance overhead of our changes and we consider the feasibility of integrating our changes with the Chromium project.

7.1 Contrived Malicious Facebook Extension

To continuously evaluate the project during development against a malicious extension, we built a contrived malicious Facebook extension which we suitably named **Malicious Facebook Extension**. Users can ask the extension to perform operations, originating from the extension's content script, that would be considered malicious had they occurred without the users knowledge and consent. Furthermore, users can request that the extension executes operations within the main world by injecting scripts and using other JavaScript constructs like `Promise`s or `MutationObserver`s. We used this extension to ensure that our modified browser correctly intercepted all suspicious operations, even when executed within different worlds.

We took inspiration from Chipman et al. [4], and published[1] our extension as an educational one on the CWS [45] that would alert users of the capabilities of malicious extensions. The extension has also been open-sourced on GitHub [46] under a MIT License.

7.1.1 Implementation

We give an overview of the design of the contrived malicious Facebook extension. It is composed of two modules: a pop-up (background) page where users request operations be carried out and a content script injected onto Facebook pages that performs the suspicious actions.

Architecture

The architecture for the contrived malicious extension is extremely simple. A pop-up page that is displayed when users click on the extension icon to the left of the URL bar simply iterates through every open tab and sends a message to it using the `chrome.tabs.sendMessage` Chrome extension API method. The

[1] The extension has been privately published and is currently only accessible via the link provided.

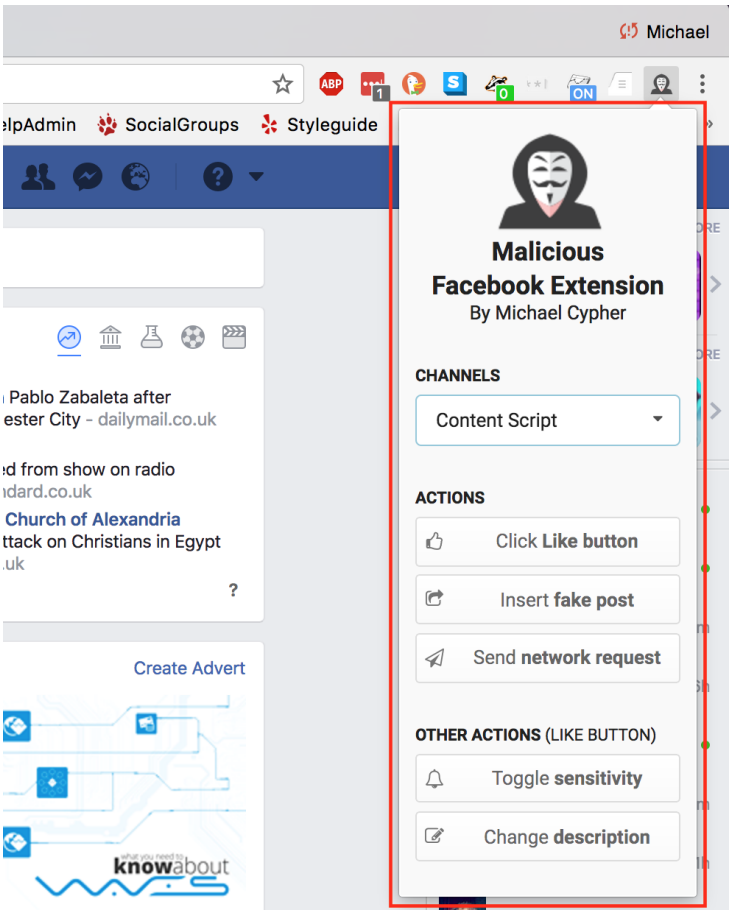

Figure 7.1: The pop-up page that is displayed when users click on the Malicious Facebook Extension button.

pop-up page and the permission to use the tabs API must be defined in the extensions `manifest.json` file (see Listing 18).

Each Facebook web-page has the malicious extension's content script injected into it and the content scripts register incoming messages handlers using the `chrome.runtime.onMessage.addListener` Chrome extension API method.

Triggering Suspicious Actions

Messages are sent whenever users click on "Action" buttons on the pop-up page (see Figure 7.1). These are several key operations we deem suspicious (§3) and includes clicking on the first "like" button on a page, inserting a fake post onto Facebook's "timeline" and sending a network request to an external domain[2]. We also include actions to toggle an element's sensitivity attribute and change its description attribute. For the purpose of this contrived evaluation extension, the extension does not currently perform operations that are not intercepted by default (e.g. reading a network request's response) (§3.10).

[2]Due to Chrome's extension permission model, we must include this domain, `https://www.reddit.com`, in the extension's `manifest.json` file. See Listing 18.

```
{
  "browser_action": {
    "default_pop-up": "pop-up.html"
  },
  "permissions": [
    "activeTab",
    "https://www.reddit.com/"
  ],
  "content_scripts": [
    {
      "matches": ["https://www.facebook.com/*"],
      "js": ["content_script.js"]
    }
  ]
}
```

Listing 18: A simplified `manifest.json` file for our contrived malicious extension with all the permissions needed to trigger suspicious actions.

Injected Scripts

Users can request that extensions perform actions within the main world by requesting the content script inject a `script` element according to the channel specified (see Figure 7.2). We provide this functionality to ensure malicious extensions cannot evade detection and interception by injecting scripts. At the moment, we support every method of injecting scripts that we are familiar with including `script` elements, event handlers, timers, `MutationObserver`s and `Promise`s (§4.5.3).

Each message is a string encoding of the selected action and channel. Whenever we receive a message, we call the function defined to carry out a specific action. If the channel requested is an injected script, we create and append a new `script` element with the string representation of the action function as its `textContent` (see Listing 30 in Appendix E).

7.1.2 Results

Our modified Chrome browser successfully intercepted (and blocks, in the case of the "Toggle sensitivity" operation) every suspicious operation that our extension fires. It does this whether or not the operation is executed within the Facebook page's main world or within the content script's isolated world. Furthermore, if we configure the browser to intercept attribute read operations, we also successfully intercept the extension attempting to find elements on the Facebook page such as the first "like" button or the "timeline" container.

7.2 Chrome Web Store Extension Analysis

In order to evaluate the effectiveness of the modified Chromium browser, we decided to manually run suspicious Chrome extensions that are publicly available on Google's CWS to see if the intercepted operations exhibit suspicious behaviour. Our goal was to find and flag extensions that perform malicious functionality or functionality that the user is unaware of.

As the CWS hosts thousands of extensions [47], with the majority benign, we focused on extensions that provide additional functionality to Facebook. Our reasoning was that these extensions are more likely to

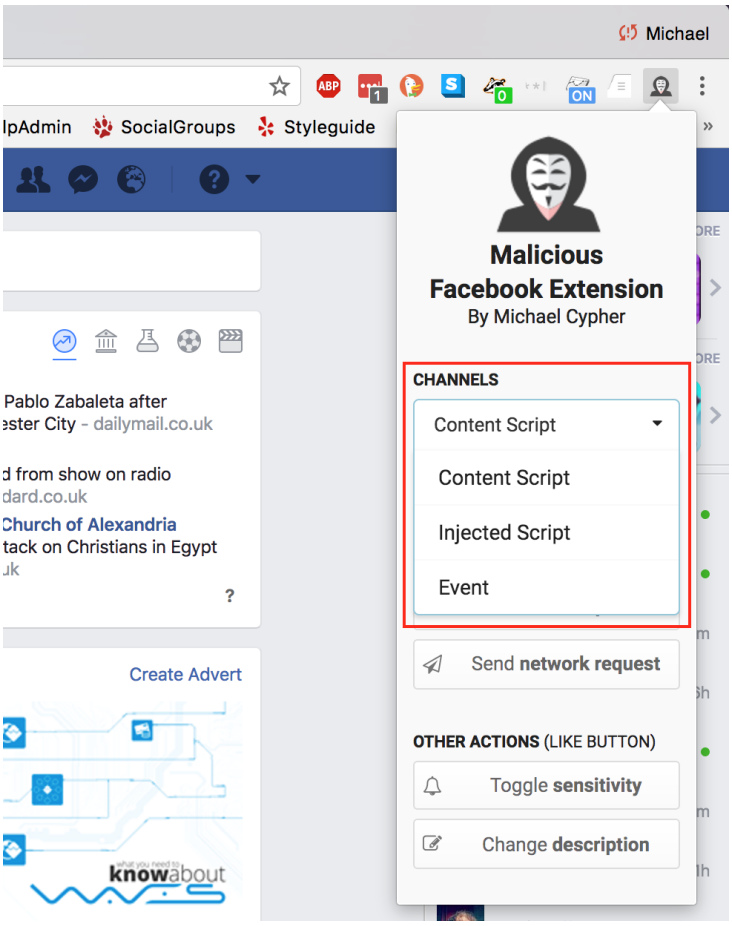

Figure 7.2: Users can request that the operation be executed within the Facebook web-page's main world by selecting different channels.

be malicious as malicious extensions that perform Facebook hijacking attacks often disguise themselves as legitimate Facebook extensions (§2.3).

7.2.1 Methodology

Finding Suspicious Extensions

To find publicly available suspicious extensions, we searched the Chrome Web Store for extensions that add functionality to Facebook using the search terms "facebook", "facebook theme" and "social". We favoured extensions published by individuals (as opposed to reputable websites or companies), with low ratings or with functionality that seemed unfeasible to add to Facebook.

For analysis, we installed one extension at a time and configured the browser to intercept all possible extension actions (§3.10). By doing this, the browser would provide more information about what the extension is doing at run-time and hopefully increase our chance of finding malicious behaviour. Additionally, this would also evaluate our choice of extension operations that we intercept by default.

Facebook Account

Before installing and analysing suspicious extensions, we created a new Facebook account for testing so to protect other non-testing accounts from being compromised via Facebook hijacking attacks. By creating a test account, we aimed to make our analysis more reproducible.

Malicious extensions normally react to specific elements on Facebook pages. For example, they may click the "like" button of posts from a particular publisher or user. In order to maximise the probability of triggering malicious functionality, we *seeded* the test account with Facebook data similar to that of typical users. This involved adding several users as "friends", "liking" the Facebook pages of various organisations and sharing posts on the test account's "timeline".

In order to follow Facebook's terms of service, the test accounts were registered with our real names and information. Unfortunately, this resulted in Facebook promptly blocking the test account. As a result, we instead reverted to using personal Facebook accounts.

Analysis

As our analysis requires running extensions to trigger malicious behaviour, we followed a similar testing script for each test. For each extension under analysis, we performed everyday, common actions on Facebook pages with the extension enabled for approximately 15 minutes.

The first step we took was to use the extension's stated functionality. For example, we experimented with changing the Facebook colour theme for theme-related extensions or moused-over images for extensions who increase the size of photos on Facebook pages. We did this for two main reasons. Firstly, we could compare pop-up dialogs displayed when performing core extension functionality to those displayed when surfing Facebook normally to detect abnormal behaviour. Secondly, we could determine if extensions were performing operations unbeknownst to users that potentially damaged their security or privacy in order to provide core functionality.

After testing the core functionality of the extension, we surfed various Facebook pages and performed the everyday actions below.

1. Scroll down the main Facebook "timeline".
2. Visit our Facebook account's profile page.
3. "like" various arbitrary Facebook posts.
4. Search for and visit the Facebook pages of various popular organisations and users. We specifically searched for "The Guardian", "Fox News", "Theresa May" and "Donald Trump".
5. Write an arbitrary Facebook post and share it[3].
6. Send an arbitrary private Facebook message to another user.
7. Click an arbitrary external link on a Facebook post shared by another user.
8. Log out of and back into our Facebook account.

[3]Posts were shared with the "Only Me" privacy setting to prevent other users from viewing them

Unlike our evaluation of benign, popular extensions (§7.4), we did not record every single action intercepted. Retroactively determining malice by inspecting intercepted action logs could prove incredibly difficult, and one goal of the modified browser and this project is to allow users to prevent suspicious extension operations at run-time. Consequently, we instead inspected each pop-up dialog manually and if malice was suspected, we investigated the operation further. We extensively used Chrome's inspector toolkit to view the extension's source code, JavaScript process' console and any network requests the extension sent or received.

7.2.2 Results

During our evaluation, we analysed 12 separate extensions publicly available on the CWS.

According to our modified Chrome browser, 9 extensions manipulated the DOM tree by creating, inserting or removing one or more elements on the page. We also found that 11 extensions read and 5 mutated element attributes. The most common attributes interacted with were: `type` (witnessed in 8 extensions), `href` (5), `disabled` (5), `value` (5) and `className` (4). The browser also alerted us of an extension reading the `src` attribute and of another interacting with custom attributes.

Two extensions caused the modified Chrome browser to crash with segmentation faults. Due to the multi-process architecture of the Chrome browser, we can deduce that the faults were caused in the browser process and likely caused by bugs in the pop-up views.

Only 2 extensions were caught sending network requests and reading their responses, and 3 were caught triggering events respectively; these turned out to be the most interesting operations intercepted. As users of the modified browser, pop-up alerts of these operations appeared most suspicious and we discuss the 3 extensions that executed them in the following case studies.

Case Study: Manipulating Facebook "likes"

We found two benign but suspicious extensions that allow users to automate Facebook actions. Similarly to the contrived malicious extension we built (§7.1), users can request that the extensions perform operations that would be considered Facebook hijacking attacks if carried out without consent.

The first extension, **ToolKit For Facebook** [48], describes itself as a "collection of Facebook automation tools that are created to save [users] time while using Facebook.". The extension is offered by `http://getmyscript.com` and, as of 26th of May 2017, has 258,406 users and a 4 star rating (based on 1830 user reviews) on the CWS, as well as 2.5k Google Plus recommendations.

Although the extension provided many tools, we tested only one, due to time constraints. The tool claims to remove all the current Facebook user's page "likes". We choose to test this functionality as "like" manipulation is a prevalent Facebook hijacking threat[4]. The modified Chrome browser successfully intercepted and alerted us of network requests to a Facebook URL and network response reads that the extension was performing in order to un-"like" pages (see Figure 7.3).

The second extension, **Like all, plus all, favorite all** [49], lets users "like" all Facebook, Google Plus or Twitter posts. As of 26th of May 2017, the extension, offered by Duc Nguyen, has 10,894 users and a 5 star rating (based on 473 user reviews) on the CWS, as well as 411 Google Plus recommendations.

[4]Although malicious extensions tend to "like" Facebook posts and pages, instead of un-"liking" them.

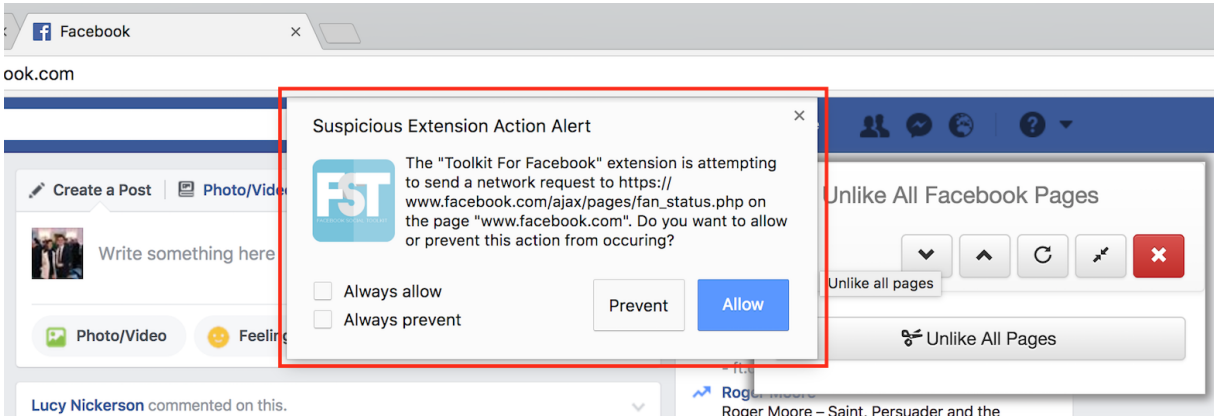

Figure 7.3: One of the many network requests the **Toolkit For Facebook** extension sent to `https://www.facebook.com` in order to un-"like" all of a user's "liked" pages.

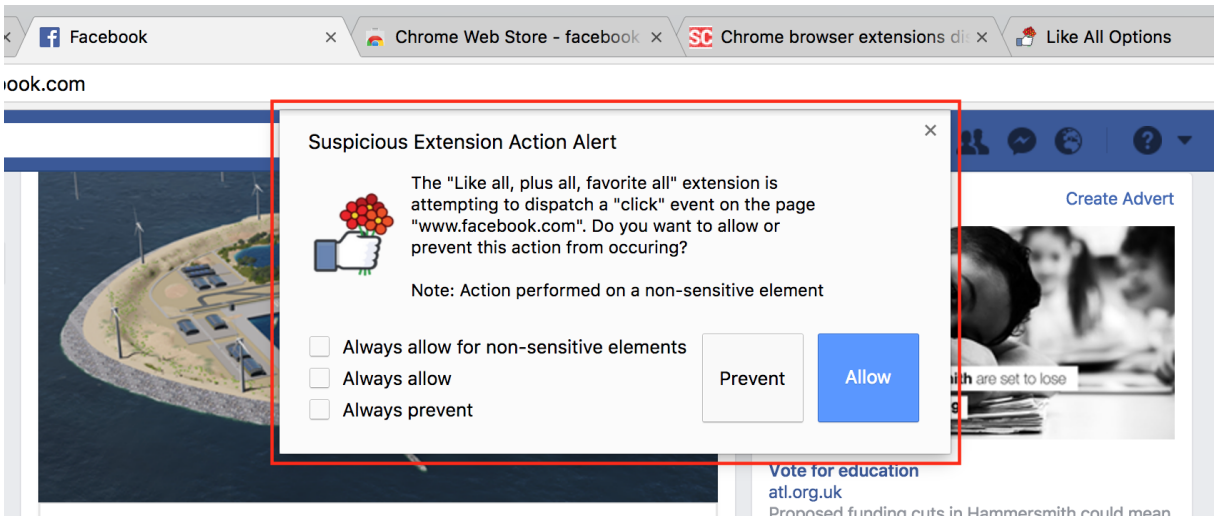

Figure 7.4: The **Like all, plus all, favorite all** extension fires events (intercepted by our modified browser) that click all of the "like" buttons on a Facebook time.

Similarly to the first, this extension manipulates "likes" on a Facebook page. However, unlike the first, it does so by firing `click` events on "like" button elements, instead of sending network requests. Despite this, our modified browser still successfully intercepts every event triggered by the extension (see Figure 7.4).

In our opinion, these case studies adequately demonstrate that our modified browser would successfully intercept Facebook hijacking attacks that attempt to manipulate "likes", whether by firing events or send network requests.

Case Study: Social Profile view notification

We discovered one extension, **Social Profile view notification** [50], that claims to "[send users] a notification when somebody views [their] social profile". The extension is offered by `http://fbpv.info` and, as of 24th of May 2017, has 156,791 users and a 3.5 star rating (based 3168 user reviews) on the CWS, as well as 1.9k Google Plus recommendations. In order to notify Facebook users when somebody visits their Facebook page, the extension would need to know what page each Facebook user is currently on. This functionality is not currently provided by a Facebook API.

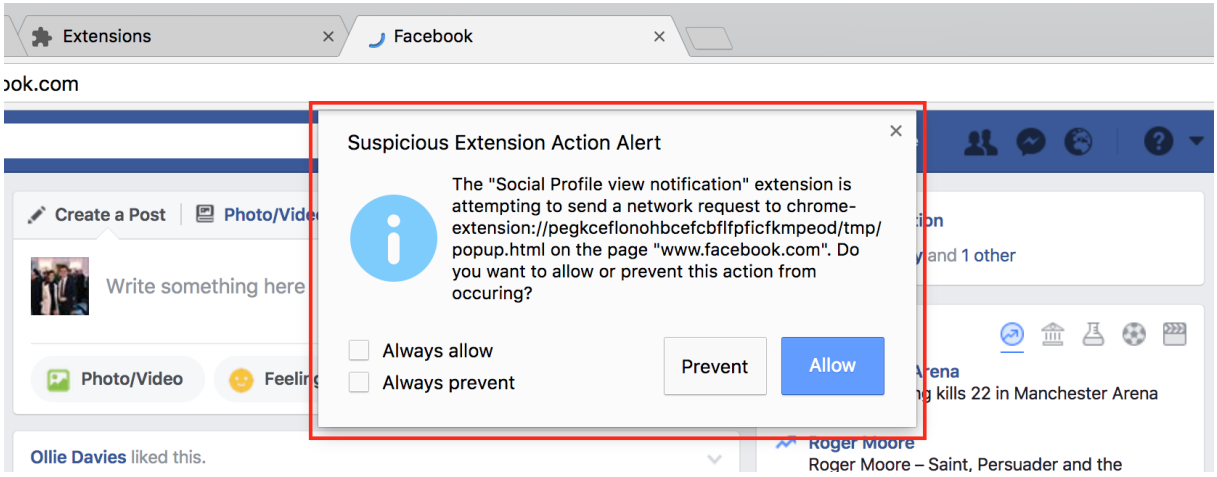

Figure 7.5: A browser pop-up dialog prompting the user to allow or prevent the Social Profile view notification extension from sending a network request to its background page.

Similar to other extensions tested, this extension created a new element and read various element attributes (including the `lang` and `value` attributes). Interestingly, the browser also alerted us (at regular intervals) that the extension was sending two network requests to its background pages `/tmp/pop-up.html` and `/tmp/user.html` (see Figure 7.5). By inspecting the network usage of the extension's background page, we noticed that it was essentially acting as a proxy and forwarding the requests to an (unresponsive) external server.

The first network request asks for a set of users currently viewing the extension user's Facebook profile. The second network request (see Listing 19) provides the external server with the identifiers of the Facebook profiles that the extension user has recently visited, as well as the extension user's Facebook profile identifier and full-name.

```
POST /v/add/ HTTP/1.1
Host: app.facebookprofileview.com
Origin: chrome-extension://pegkceflonohbcefcbflfpficfkmpeod
X-Requested-With: XMLHtpRequest
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

user_id:1031609757
fullname:Michael Cypher
ids[0][id]:100000626470156
```

Listing 19: A (simplified) version of the HTTP request that the Social Profile view notification sent which includes Facebook user identifier, full-name and the identifiers of Facebook profiles the extension user has visited. We have decoded and provided the content in a human-readable format.

According to our definition (§3.1.1), the second network request could be considered malicious. The extension does not suggest that by using it, users provide[5] the third-party service with information about what Facebook profiles they visit. Knowing this, many users would likely not enable the extension or at least prevent it from executing the malicious operation (sending the HTTP request in Listing 19). In this case, our modified Chrome browser successfully flagged malicious behaviour and gave users the ability to prevent it from occurring.

[5]This information is also sent using the insecure HTTP protocol.

7.2.3 Limitations

Although we found two benign but suspicious extensions that manipulated Facebook "likes" and one extension that we consider malicious, this evaluation surfaced serious limitations with our modified browser.

Lack of Information

The first and most noticeable limitation was the lack of information provided by the pop-up dialogs. In order for this project to be successful, users must be able to correctly distinguish between malicious and benign extension actions when alerted with a pop-up. Unfortunately, even when searching for malicious extensions, this task is not currently straightforward.

Currently, the only way we describe elements on a web-page is using the `data-description` and `data--sensitive` attributes. Unfortunately, this only works when the web-page has set the description and sensitivity of each element, which Facebook does not currently do. Consequently, we could not easily determine if operations performed (e.g. events dispatched or data read) on elements are malicious or not.

We also did not have enough information to determine if inserting, replacing, removing or creating elements was malicious. As the majority of extensions performed these operations, we assumed they were benign and necessary for core functionality and therefore we always allowed these actions.

Sending network requests and reading network responses also proved difficult to analyse with our pop-ups. We currently only display the URL of the request and do not show parameters or custom headers sent with HTTP POST requests. In order to determine the intent of the network requests, we inspected them manually using Chrome's built-in developer tools. However, non-technical users are extremely unlikely to do this themselves and we assume they would either allow or prevent all requests to external servers. In the case of the Social Profile view notification extension, this would result in allowing malicious behaviour (sending the user's Facebook profile views) or breaking core functionality (seeing who viewed the user's Facebook profile).

Finally, we struggled when analysing element attribute reads or mutations. Our modified browser intercepts and prompts users (if configured) whenever an extension attempts to remove an attribute or read and change the attribute's value. During the evaluation we only displayed the name of the attribute and not the actual value which did not provide enough information to determine malice.

Background Page Proxy-ing

When analysing the Social Profile view notification we also discovered another limitation with regards to network requests. The extension's background page was proxy-ing requests from its content script to an external server. All extension network requests intercepted on the Facebook page were being sent to the background page (see Figure 7.5), which appeared less malicious than if they were being sent to an external server. The fact that they are forwarded to the external server from the background page is hidden from everyday users (see Figure 7.6). This *disguise* makes network requests containing sensitive information look less suspicious as they do not appear to leave the browser.

This disadvantage exists because we currently only intercept operations executed within content scripts. We do this by only considering operations that are executed originally from within an isolated world and determining the extension from the isolated world's security origin (§4.5). As background pages run

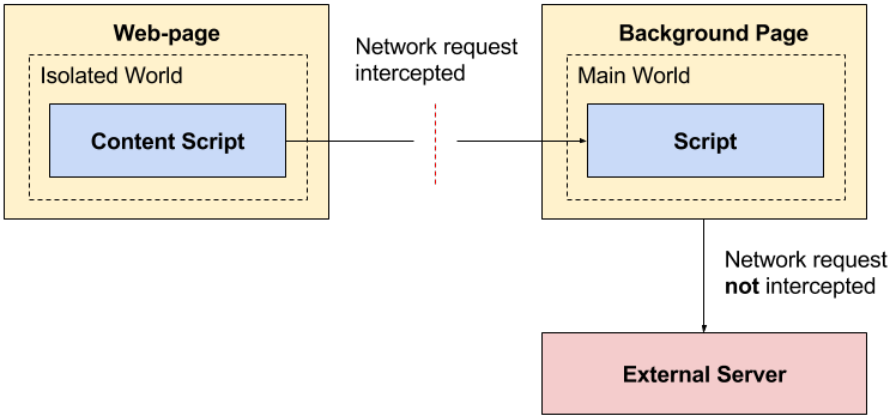

Figure 7.6: A malicious content script can *disguise* the recipient of a network request by proxy-ing it through its background page.

within their own process (similar to other web-pages), all actions executed within a background page operate within its main world.

7.3 Mechanical Turk: User Survey

To properly evaluate the user interface and experience with regards to our pop-up dialogs that warn users of suspicious extension actions, we carried out a user survey. In particular, we wanted to figure out whether users understood what our pop-ups were warning them of, whether they would allow or prevent extension actions in general and whether or not different extension actions would influence users differently.

Unlike our previous evaluation of malicious extension action pop-ups (§7.2), we decided to focus on just benign actions for the purpose of this analysis. We crafted a series of small surveys that asked 7 questions (see below) on anonymised[6] screenshots of 42 unique extension action warnings. The screenshots included pop-ups warnings of extensions attempting to trigger events, mutate the DOM (i.e. add, remove or replace elements on a page), create elements, send network requests and read their responses, read element attributes and change them.

1. **Question:** "Do you understand what the pop-up is telling you?"
   Answers: "Yes", "No" or "Kind of".
2. **Question:** In around 140 characters (length of a tweet), explain what you think this pop-up is doing?
3. **Question:** Would you click "Accept" or "Prevent"?
   Answers: "Accept" or "Prevent"
4. **Question:** In around 140 characters (length of a tweet), explain why you would click "Allow" or "Prevent"?
5. **Question:** Would you select "Always allow" or "Always prevent"?
   Answers: "Always allow", "Always prevent" or Neither

[6] All personally identifiable information was hidden from screenshots.

6. **Question:** Do you think the extension has done something malicious?
   Answers: "Yes", "No" or "Not sure"
7. **Question:** What would you expect would happen when you click the close ("X") button in the top right corner of the pop-up?
   Answers: "Allow action", "Prevent action" or "Not sure"

We distributed the survey through Amazon's Mechanical Turk [51], a crowd-sourcing Internet marketplace, and requested 20 responses for each screenshot, or equivalently 840 responses in total. To facilitate peer review, we have published survey results online [52].

7.3.1 Preventing Suspicious Actions

One of our goals was to reduce the number of warning pop-up dialogs so that users only need to make decisions infrequently (§6). As breaking few benign extensions is a project goal, we hoped this would reduce the frequency at which users prevent benign operations and break extensions. However, we still wanted to determine if respondents, would generally allow or prevent extension actions in order to quantify the number of benign extensions we break. We asked survey respondents if they would allow or prevent an extension action given a screenshot of its corresponding warning dialog. We found that 64% of respondents chose to prevent actions in general and that this varied based on the extension action (see Table 7.1). We attempted to confirm this hypothesis by running a Chi-squared test with 6 degrees[7] of freedom and computed a Chi-squared value of 7.964 and p-value of 0.241 but could not reject the null hypothesis[8] with enough certainty. Although we cannot guarantee that respondents did in fact read pop-up dialogs carefully, we believe the dialog message did have an impact and therefore we analyse the results for each action group.

We discovered that users are also more likely to prevent extensions attempting to change attributes (72.5% prevented) and trigger events (70.0% prevented). Since events are often necessary in Facebook hijacking attacks (§3.3), it is generally good that users are more cautious with them. However, we haven't found any extensions that carry out attacks by changing element attributes, although it is possible, and we would ideally like users to prevent these actions less frequently. This reinforces our decision to not intercept these actions by default (§3.10). On the other hand, respondents were less likely to prevent extensions creating elements (61.0% prevented) or reading element attributes. This may be because users share our belief that these actions as less likely to be malicious.

Since we only warn of benign extension actions in the pop-up dialogs we presented in the survey, we expect our modified browser to break at least 64% of benign extensions that require one suspicious operation to succeed in order to perform core functionality. This assumes that users have configured their browser to intercept all given extension action groups. If extensions require multiple suspicious operation approvals, or require changes to element attributes and events to fire, we expect this value to increase substantially.

7.3.2 User Understanding

We investigated how many people understood our pop-ups in order to determine whether they were too complicated for the everyday user. We discovered that in general, 53%, 28% and 20% of respondents

[7] We excluded the "Total" row in Listing 7.1.

[8] The null hypothesis in these tests is that external variables like extension actions or user understanding do not influence a user's decision to allow or prevent an extension action.

Action Group	Size	Prevented (%)	Bar Chart: Prevented (%)
Total	839	64.1	
Change Attribute	120	72.5	
Event	80	70.0	
DOM Mutation	280	63.6	
Network Request	80	62.5	
Network Response	80	62.5	
Create Element	100	61.0	
Read Attribute	99	56.6	
			40.0 60.0 80.0

Table 7.1: The percentage (1 decimal point) of respondents that chose to prevent the extension action for each action group.

understood, "kind of" understood or did not understand the suspicious extension action warning. This shows that there is huge room for improvement and perhaps teaching users about the implications and effects different groups of actions would help.

In Chapter 3, we decided which suspicious actions to intercept based on their likelihood to be used by malice. However, it is not clear if users understand the implications of suspicious actions and we therefore compared user understanding with extension action groups (see Figure 7.7). The extension actions described in the warning pop-ups did not seem to have a huge influence on whether users understood the pop-up or not. One takeaway is that more people understood what it meant to add, remove or replace elements on the page than what it meant to create an element or change an element's attribute.

In order to determine if user understanding had an influence on the decision to allow or prevent extension actions, we performed a Chi-squared test with 2 degrees of freedom. We found the chi-squared value to be 70.518 and p-value to be 0, and we therefore rejected the null hypothesis. Following on from this conclusion, we discovered that, according to our survey, 89.1% of the 166 respondents who did not understand the pop-up prevented it (see Figure 7.7). On the other hand, 67.8% of the 233 respondents who "kind of" understood the pop-up prevented it, and only 52.8% of the 441 respondents who understood the pop-up prevented it. With such large sample populations, it is clear that the less people understand our warning pop-up, the more likely they will prevent the associated action (i.e. respondents were more risk-adverse when they didn't understand the warnings).

We also discovered that 43.4% of the 128 users that considered an extension malicious also allowed the extension from performing the suspicious operation. Furthermore, 39.0% of 259 users that considered an extension benign also prevented it from executing a suspicious operation. We found this deeply worrying as these respondents were acting in a contradictory manner - either harming themselves by allowing (what they consider to be) malicious extensions or breaking benign extension functionality. This also suggests that these users don't understand our warning pop-ups. Finally, 87.5% of the 352 users that are not sure whether an extension is malicious or not, prevent the extension actions.

As pop-ups shown were describing benign extension actions, we hoped that users that understand the pop-up would allow the actions to occur. However, this proved not to be the case with 52.8% of respondents who claim to understand the pop-up still preventing actions. We believe this is because users that

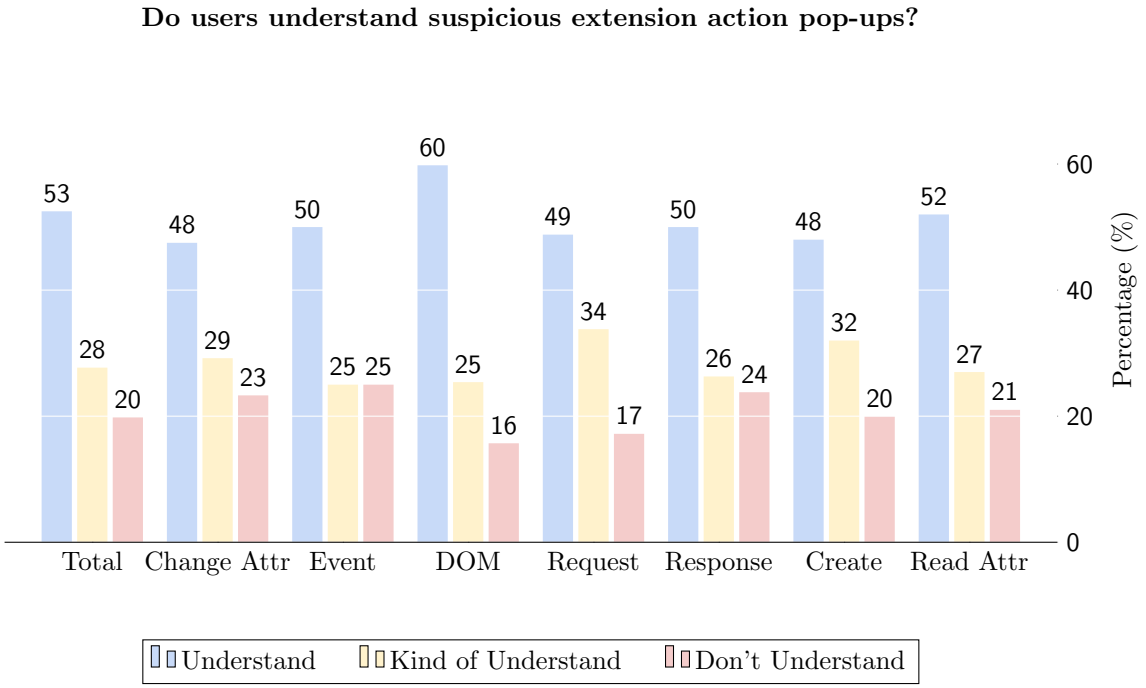

Figure 7.7: A bar chart explaining the percentage (to nearest integer) of users that understand, "kind of" understand or don't understand our warning pop-ups.

understand the pop-up warnings may still not fully understand what the extension is doing or are not sure whether the extension is benign or malicious and therefore act in a risk-adverse manner.

7.3.3 Remembering User Decision

Our modified Chrome browser can remember a user's decision to allow or prevent a certain group of actions[9] (§6.3) so that they do not need to approve or prevent the same extension multiple times. Instead of having one "Remember decision" checkbox, we opted for two[10], "Always allow" and "Always prevent", with the aim of clarifying the pop-up. To evaluate this decision, we included a question in our survey to determine if users will choose to have the modified browser remember their decision.

Although a majority of respondents choose to select "Always allow" (23.1%) or "Always prevent" (39.9%), this may be because more respondents prevent actions than allow them. Given that the user allowed the extension action, 63.1% chose to always allow the prompt and 33.9% chose to always display the prompt (neither always allow or always prevent). On the other hand, given that the user prevented the action, 60.6% chose to always prevent the prompt and 38.7% chose to always display the prompt. Users are therefore slightly less likely to request that the browser remember their answer if they decided to prevent the extension action. This may be because respondents view preventing all actions and subsequently breaking extension functionality as worse than allowing all actions and putting themselves at risk (i.e. users chose convenience over security).

We also found that only a negligible number of users selected "Always allow" and clicked the "Prevent" button (4) or selected "Always prevent" and clicked the "Allow" button[11] (9). This suggests that users

[9] In fact, we map the text displayed on a pop-up to a user's "remembered" decision.
[10] We show a third, "Always allow for non-sensitive elements", if the extension operates on sensitive elements or data.
[11] Our modified browser explicitly forbids users from selecting "Always prevent" and clicking the "Allow" button and vice versa (§5.5).

understand this feature and that the user interface correctly conveys the functionality.

7.3.4 Closing Pop-up Dialogs

We were also interested in discovering whether or not users knew what the default behaviour of warning dialogs is when they are closed using the `X` button in the top right corner. Due to project goals, we chose to allow suspicious extension actions by default to reduce the number of extensions we break (§5.5).

We found that 53.2% of respondents think clicking the close button prevents the extension action, 27% are not sure and 19.8% think it allows it. Only one in five people are actually correct which demonstrates a major flaw of the pop-up user interface and experience.

By reaching out to Chromium view developers[12] via IRC, we learned that our current dialog implementation is considered an anti-pattern and that "Cancel" buttons (i.e. our "Prevent" button) on dialogs should not perform any functionality but instead cancel the dialog. To conform to existing patterns, our dialog could instead have a "Prevent" button, where we now have the "Allow" button, that prevents suspicious extension actions and a "Cancel" button that does nothing (i.e. implicitly allows them). This would likely improve user-understanding but may result in users preventing more benign extension actions. We leave exploring this as future work.

7.3.5 Survey Limitations

Two limitations of this evaluation are apparent and we must therefore view the survey results with some scepticism.

First and foremost, our respondents were paid to answer the survey. Mechanical Turk allows requesters (i.e. those who issue survey jobs) to discard responses that seem incorrect and respondents may therefore read the questions more carefully than everyday users. Browser users would likely be surfing the web when our warning pop-ups are displayed and might want to close them as quickly as possible in order to continue browsing. Alternatively, we might in fact see the opposite result whereby respondents are less concerned about the content of warning dialogs than browsers users as their aim may be to finish the survey as quickly as possible.

Secondly, the survey respondents may not be a representative sample of all extension users and may be more technically inclined. However, as we neither have access to demographic information about Mechanical Turk or Chrome browser extension users, we don't infer anything from this.

7.4 Popular Benign Extensions Analysis

Building on our survey results, we wanted to empirically determine how many suspicious actions are executed by popular extensions. This would help us better predict the likelihood of breaking popular extensions given that 64% of users prevent suspicious actions on average (§7.3.1). Furthermore, this would substantiate decisions we made to not intercept potentially risky actions that we believed would be executed too frequently and thus inconvenience the user (e.g. reading or mutating element attributes).

[12] Chromium view developers focus on building and maintaining user interface related code including user dialogs.

7.4.1 Methodology

Before starting this experiment, we added additional logging to our modified browser and configured it so it would not display pop-up alerts for any suspicious actions (§3.10). This would allow us to run extensions without warning dialog interruptions.

For this analysis, we looked at 4 popular and recommended extensions within the "Getting Started" collection, which is on display at the top of the CWS. Each extension had close to or well over a million users according to the CWS and are listed below:

1. **Grammarly for Chrome** [53]. The extension is offered by `grammarly.com` and, as of the 14th of June 2017, has 10,000,000+ users and a 4.5 star rating (based on 20038 user reviews) on the CWS, as well as 4.9k Google Plus recommendations.

2. **LastPass: Free Password Manager** [54]. The extension is offered by `lastpass.com` and, as of the 14th of June 2017, has 5,712,137 users and a 4.5 star rating (based on 21537 user reviews) on the CWS, as well as 4.5k Google Plus recommendations.

3. **Google Dictionary (by Google)** [55]. The extension is offered by Google and, as of the 14th of June 2017, has 3,046,210 users and a 4.5 star rating (based on 12139 user reviews) on the CWS, as well as 10k Google Plus recommendations.

4. **StayFocusd** [56]. The extension is offered by Transfusion Media and, as of the 14th of June 2017, has 745,574 users and a 4.5 star rating (based on 5127 user reviews) on the CWS, as well as 4.6k Google Plus recommendations.

7.4.2 Results

The results substantiated our decision to not intercept all suspicious actions by default (§3.10). On average, extensions executed an order of magnitude fewer actions if we consider only default suspicious actions versus all possible suspicious actions (see Figure 7.8).

Unfortunately, we found that popular extensions that we analysed still executed far too many default suspicious actions. The **Grammarly**, **LastPass**, **Google Dictionary** and the **StayFocused** extensions each executed 1,135, 1,250, 229 and 1,632 defaults suspicious actions over a 15 minute period[13]. This would certainly overwhelm users if they decide not to use "Always allow" or "Always prevent" features. Furthermore, considering the rate at which users prevent extension actions (§7.3.1), we believe that our modified browser would break core functionality of each of the benign extensions analysed and this is a major disadvantage of our project.

7.4.3 Limitations

A limitation of this analysis is that it does not consider the fact that users can choose to have our modified browser save their preference (§6.3). Users are likely to do so and this would substantially reduce the number of dialogs they are presented with.

[13]The extensions executed 15,056, 15,278, 1,095 and 19,749 possible suspicious extensions respectively.

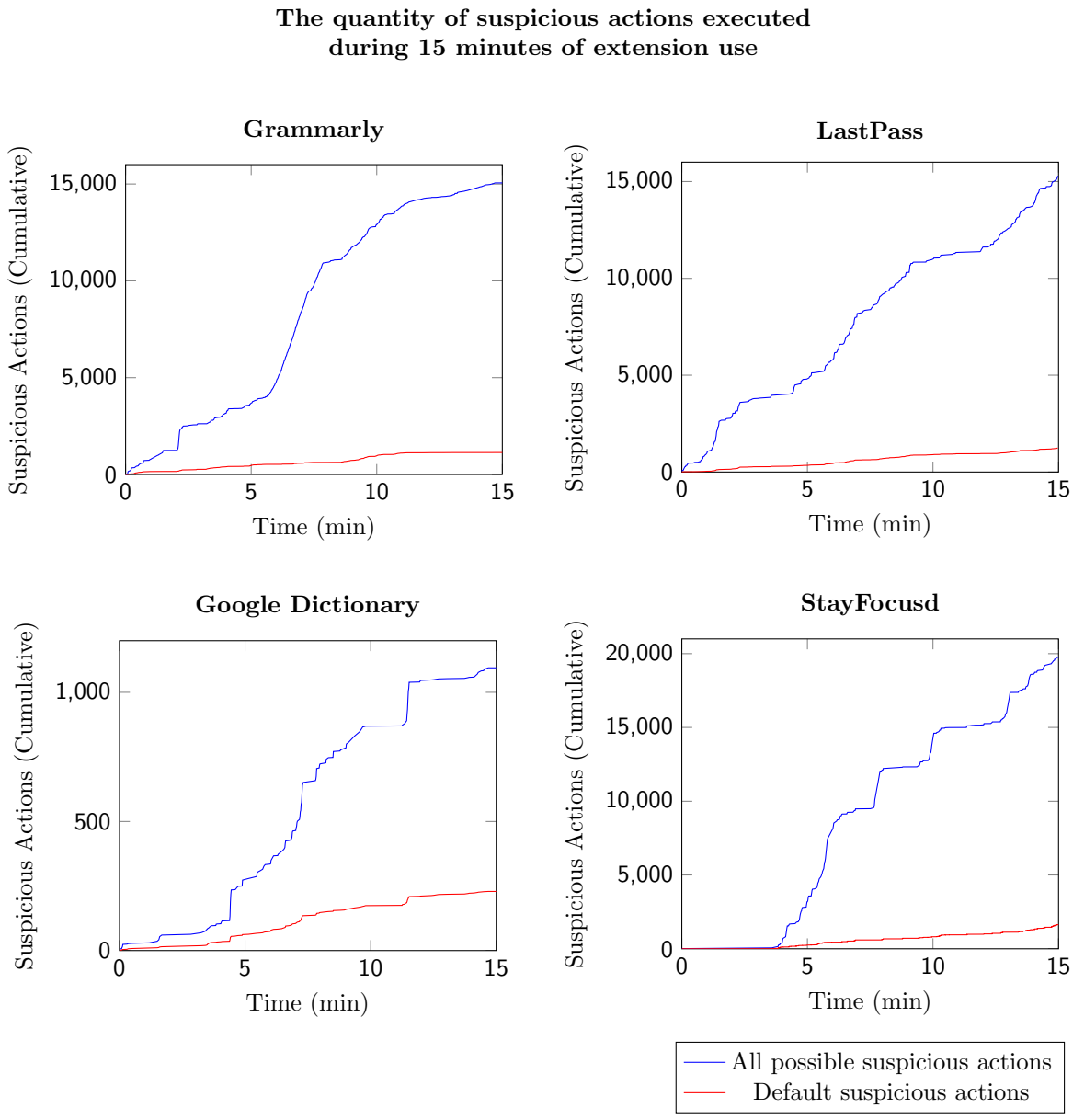

Figure 7.8: A plot of the cumulative number of suspicious actions that the Grammarly, LastPass, Google Dictionary and StayFocusd extensions execute during a 15 minute long browsing period. The blue line refers to all possible suspicious actions whereas the red line refers to only default suspicious actions (§3.10).

7.5 Performance

Companies heavily optimise their web-pages, scripts and servers to quickly deliver responsive web-pages to users. Consequently, users have come to expect responsive web-pages.

However, heavily optimised web-pages and scripts can still render and execute slowly if run with slow rendering and JavaScript engines. The Chromium project therefore invests a lot of resources in profiling the performance, memory and energy consumption of the Chrome browser. The project's perf regression sheriff bot [57] observes Chrome's continuous integration tests, which are run against every build, and alerts developers of performance regressions via a publicly view-able dashboard [58].

As a result, one project goal was to ensure any modifications to the Chrome browser do not incur a substantial overhead or negatively affect the browser's responsiveness or memory and power usage. We expect to see a performance overhead in intercepted operations when we check to see if execution is occurring within the content of an isolated world. In this section, we profile the performance of our modified Chrome browser, running without extensions, to ensure our changes to the Blink or JavaScript V8 engine do not negatively impact web-page responsiveness.

Our modified browser correctly blocks the renderer process' UI thread whenever pop-up dialogs are displayed (§5.3) as a result of suspicious browser operations. Web-pages and content scripts are therefore made unresponsive. In order to less frequently block web-pages, we worked on reducing the total number of dialogs (§6).

7.5.1 Telemetry Benchmarks

Telemetry is a performance testing framework used by the Chromium project. Developers can run automated tests that launch the browser, open windows and tabs, navigate to pages and perform arbitrary operations on the web-pages [59].

Fortunately, the Chromium source directory contains a bundle of Telemetry performance tests, located with the `src/tools/perf` directory. The test suites contain Blink and V8 engines performance tests, as well as browser power and memory usage tests. As our project mainly added code in the Blink engine code-base, the majority of which involves checking if execution is within an isolated world, we decided to benchmark our modified browser against the original in terms of Blink engine performance.

Although we have regularly merged in Chrome changes from the master branch, our modified branch was behind the master branch when evaluating it. As a result, when comparing the performance of our modified browser to the original, we checked-out the closest commit on the master branch to our branch and built it as a separate production browser.

7.5.2 Methodology

We executed Telemetry benchmarks on a MacBook Pro (Retina, 15-inch, Mid 2015) running macOS Sierra (version 10.12.5) with a 2.5 GHz Intel Core i7 processor, 16 GB 1600 MHz DDR3 of RAM and with a Intel Iris Pro 1536 MB graphics card. To reduce fluctuations in processing ability, no other applications were running during tests apart from those that macOS executes in the background upon start-up.

On test completion, Telemetry produces a report, in the form of a web-page, that displays various statistics including the mean, min, max, standard deviation, count sum for each benchmark test. These metrics are aggregated over a set of 50 tests runs in order to have a significant sample size. Telemetry's report can also compare test runs between two browsers (see Listing 20) and can display statistics of one test run relative to another. These include the absolute and percentage difference of all statistics (i.e. mean, min, max, etc), the p-Value[14], Mann-Whitney U statistic[15] and z-score[16]. We comment these statistics in this chapter.

For each benchmark (e.g. blink_perf.dom), we have produced a table comparing the average performance of the original Chrome browser to our modified one (in milliseconds). We calculated a 95% confidence interval for these timings given the mean and standard deviation and display this using the $\pm$ sign (where relevant). We also show the percent difference between the two average timings. Finally, we colour the background of every row (i.e. test) that has a p-Value greater than 0.05 as we cannot reject the null hypothesis that our modifications had no impact on the test.

As we added custom tests ourselves, we split the rows of results in each table in two with a solid line. The bottom half contains the results to our custom tests.

```
$ tools/perf/run_benchmark blink_perf.dom
> --browser-executable=out/Default/Chromium.app/Contents/MacOS/Chromium
> --reset-results --results-label="Modified Chrome Browser"
> --pageset-repeat=10

View result at file:///Users/cypher/Documents/chromium/src/tools/perf/results.html

$ tools/perf/run_benchmark blink_perf.dom
> --browser-executable=out/OldMaster/Chromium.app/Contents/MacOS/Chromium
> --results-label="Original Chrome Browser"
> --pageset-repeat=10

View result at file:///Users/cypher/Documents/chromium/src/tools/perf/results.html
```

Listing 20: The shell commands we used to execute the blink_perf.dom performance test suite on both the modified Chrome browser (Default) and original Chrome browser (OldMaster) compare the results.

7.5.3 Benchmark: Blink DOM Operations

The first benchmark we ran analysed the performance of DOM operations within the Blink engine (blink_perf.dom). This is particularly relevant as the majority of operations intercepted involved mutating or reading from the DOM.

According to the test results (see Table 7.2), modifying an element's class name, identifier or title resulted in a slight performance regression. As all of these methods involve modifying an element's attributes and as we intercept calls to `Element.setAttribute`, we expected to incur a performance overhead. We were pleased as the overhead was generally negligible. However, to confirm a small overhead, we created a new set-element-attribute test similar to the modify-element-classname test but that explicitly calls the `Element.setAttribute` method 1000 times and saw a substantial 383.731% increase in timings. Further investigation revealed that setting an element's class name by mutating the class name field (i.e. `el.className = 'class1';`) does not call into our modified `Element.setAttribute` method.

[14]The probability the execution time distributions are not significantly different.
[15]The U value produced and consumed by the Mann-Whitney hypothesis test.
[16]The difference between the execution time distribution mean and the reference distribution mean in number of standard deviations.

We noticed a small increase in timings when calling the `Selection.addRange` method [60]. We traced the Blink code executed in this test and could not find any calls to modified methods. Since the increase is so small and the timing confidence intervals nearly overlap, we also considered this increase negligible and likely down to chance (the 95% confidence intervals also overlap). Further profiling should be performed to confirm this.

Surprisingly, we saw a large decrease in timings (-49.854%) for the select-multiple-add test which "measures [the] performance of adding option elements to a multi-selection select element". This test appends 500 child nodes using our modified `ContainerNode.appendChild` method to a container node and then clears the container node's inner HTML. We also saw a large decrease in timings for the select-single-add (-17.241%) test and select-single-remove test (-28.031%). As we did not believe code introduced to DOM mutation operations increased performance, we created the append-child test which appends 1,000 `div` elements to a `div` container element and we saw a similar decrease in timings (-73.986%). We believe this performance improvement could either be due to compiling the original browser with incorrect optimization flags or as a result of an incorrect implementation in our modified browser. Due to time constraints, we did not investigate this further but this task for future work.

We also noticed timing reductions in the long-sibling-list and textarea-dom tests. Similarly to the selection-related tests, we did not establish probable cause and further investigation is necessary.

The remove_child_with_selection test displayed the largest performance hit of all tests. Upon inspection we discovered the test removed 1,000 elements from a container and appended them to another using the `ContainerNode removeChild` and `appendChild` methods respectively. As no test existed to measure the impact of our changes on the `Document.createElement` method, we created the create-element test which creates 500,000 elements in a loop. We discovered a 47.539% increase in timings for it which is a substantial performance regression.

We consider this first set of benchmarks as definitive proof that the changes introduced to our modified Chrome browser do have a performance overhead and future work should address this if necessary[17].

7.5.4 Benchmark: Blink Events

As we modified the `EventDispatcher::Dispatch()` method, we ran the Blink Events benchmark (blink_perf.events). Unlike DOM operations (e.g. `ContainerNode.appendChild`), we also create a new `HashSet` and subsequently delete it within the `EventDispatcher::Dispatch()` method. Therefore, we expected to see a larger performance impact on events than on DOM operations[18].

We were surprised to see reductions in timings for our modified browser for the EventsDispatcing test (-19.279%). This test fires 500 custom events on a DOM tree of depth 50 and we confirmed, by intentionally breaking the performance tests, that it executes the modified method. We investigated this further by creating the SimpleClickDispatch test that fires 1,000 clicks on a single element appended to the page's `body` and found a 69.522% increase in timings. This more closely correlated to our DOM operation benchmark results.

[17] Whether or not these changes are considered performance regressions is up to the Chromium team to decide.
[18] As mentioned, we ended up seeing timing reductions for the DOM operation benchmark tests.

Name	Original Browser Avg (ms)	Modified Browser Avg (ms)	Δ Avg (%)
modify-element-classname	4,489.4 ± 42.0	4,797.5 ± 100.4	+6.9
addRange	3,821.9 ± 13.8	3,852.6 ± 21.1	+0.8
modify-element-id	1,720.3 ± 11.6	1,764.1 ± 10.1	+2.5
modify-element-title	1,499.8 ± 11.1	1,512.1 ± 7.3	+0.8
select-multiple-add	141.3 ± 0.9	70.9 ± 0.5	-49.9
remove_child_with_selection	57.6 ± 0.7	140.8 ± 1.2	+144.6
select-single-add	26.0 ± 0.1	21.5 ± 0.1	-17.2
inner_html_with_selection	24.5 ± 0.6	24.9 ± 0.6	+1.2
select-long-word	14.4 ± 0.3	14.4 ± 0.2	-0.1
long-sibling-list	13.9	12.0 ± 0.1	-13.7
select-single-remove	8.5 ± 0.1	6.1	-28.0
textarea-dom	3.4	3.0	-14.0
div-editable	0.3	0.2	-4.7
textarea-edit	0.2	0.2	-3.5
create-element	4,418.0 ± 184.5	6,518.4 ± 77.9	+47.5
append-child	55.1 ± 1.2	14.3 ± 0.2	-74.0
set-element-attribute	14.5 ± 0.3	70.0 ± 0.6	+383.7

Table 7.2: A comparison of Blink DOM Operation benchmarks for both the original and modified Chrome Browser.

Name	Original Browser Avg (ms)	Modified Browser Avg (ms)	Δ Avg (%)
...ShadowTrees*	601.9 ± 2.9	579.5 ± 13.0	-3.7
...DeeplyNestedShadowTrees**	235.3 ± 0.9	233.3 ± 0.9	-0.8
EventsDispatching	25.2 ± 0.1	20.3 ± 0.1	-19.3
SimpleClickDispatch	56.2 ± 0.7	95.3 ± 0.9	+69.5

Table 7.3: A comparison of Blink Event benchmarks for both the original and modified Chrome Browser. The test EventsDispatchingInShadowTrees* and EventsDispatchingInDeeplyNestedShadowTrees** have been shortened and have a sample size of 30 and 45 in the Original Browser tests (due to test timeouts).

7.5.5 Benchmark: Blink Network Requests

Since we made changes to the `XMLHTTPRequest.send` method, we ran the associated performance test. We do not have enough information to determine if our modifications had a performance impact (the p-Value was 0.118) as the timing increase was only small (+2.954%).

One limitation is that we also changed methods to read a network response but the only test within this benchmark analyses the time to send network requests. To address this, we also added a read-response test which sends and reads 1000 network requests and responses respectively. Although we saw a small increase in the mean, it wasn't enough statistically significant due to the large standard deviations of test runs (202.140ms and 202.411ms respectively). Although we believe our changes had an impact, we suspect that they are insignificant compared with the costs of making a network request.

Name	Original Browser Avg (ms)	Modified Browser Avg (ms)	Δ Avg (%)
send	1,173.8 ± 57.1	1,208.5 ± 59.2	+3.0
read-response	1,208.1 ± 56.0	1,245.5 ± 56.1	+3.1

Table 7.4: A comparison of Blink XML HTTP Request benchmarks for both the original and modified Chrome Browser.

7.5.6 Benchmark Limitations

Telemetry and the Chromium source code have a wide list of other useful benchmarks that test the performance of the V8 engine as it browses desktop or mobile web-pages and memory or power usage across the top 10 and 25 web-pages. However, many benchmarks require Google accounts in order to, presumably, execute the tests on the cloud. Consequently, our Telemetry tests are limited to those able to run locally.

7.6 Integration With The Chromium Project

An indirect goal of this project was to ensure that any changes to the Chrome browser could be adopted and integrated with the Chromium project as a whole. As a result, we prioritised minimal and simple software solutions over complex ones.

Our modified browser consists of around 2650 and 100 lines of code added and deleted respectively to the Chromium browser. The majority of which were applied to the browser process and consist of new view classes for the warning dialog and "Configure Suspicious Actions" dialog. However, apart from necessary performance tests, no new tests were introduced to ensure correct execution. We therefore expect the quantity of code added to increase substantially before our browser is production ready. Although this project would produce a large patch, we believe it is acceptable due to the size of the complexity of the changes implemented.

We provide a generated git diff file between our modified Chrome browser and the latest (locally) merged commit from the Chrome browser master branch. This commit was published on the 18th of May 2017 and its associated code review is visible online [61]. Our changes have not yet been presented as a patch to the Chromium project.

7.6.1 Backwards Compatibility

As various JavaScript Web APIs now can potentially throw security exceptions, existing extensions are likely to break when users prevent them from executing malicious operations. Extensions that currently handle exceptions gracefully before attempting to call suspicious extensions should remain unaffected.

As a project goal was to not break existing browser functionality, web-pages or many benign extensions, we also aimed to break few existing tests. Of the 54,522 existing Blink Layout tests, our modified browser broke 374 (0.7%). We noticed all that 4 test timeouts and the vast majority of test crashes involved performing a suspicious operation (§3) within an isolated world (§4.5). According to test comments, the `chrome-extension://` scheme is an unknown scheme and as a result, isolated worlds created within

tests have their security origins set to `https://cloud-cuckoo-land.google:2112` while "pretending" the URL is a valid extension security origin. We believe our test crashes were a result of our assumption that all isolated worlds are associated with extensions that can be retrieved via their security origins. However, as previously mentioned, this assumption can be easily corrected.

Furthermore, extensions that actively use the `data-description` and `data-sensitive` attributes for unrelated reasons may find these actions are intercepted by default or prevented respectively. Web-pages that use these attributes for unrelated reasons may also unintentionally impact our modified Chrome browser. A potential solution to these issues could involve standardising these attributes.

7.7 Summary

In this chapter, we evaluated our modified browser against project objectives (§1.1).

We first built a test extension that executes suspicious operations, carries out Facebook hijacking attacks and performs the necessary actions to carry out ad injection and user tracking attacks. The extension also attempts to evade detection by executing operations within the page's main world. Our modified browser proved successful in intercepting all suspicious operations and guaranteeing sophisticated attackers cannot evade detection, whilst also protecting users against Facebook hijacking attacks.

We analyse a dozen potentially malicious extensions present on the CWS to evaluate the effectiveness of our project. Our browser intercepted suspicious operations that could have been malicious if executed without user permission from two extensions. Furthermore, we detect a broken, but potentially malicious extension that leaks user data. We discover our warning dialogs often do not provide enough information to correctly classify actions as malicious or benign.

As our project is heavily based on user participation, we evaluated both the user interface and warning dialogs by running a survey with nearly a thousand responses. We conclude that users do not have enough information or expertise to understand our warnings and correctly classify extension actions, and we find that the majority of users prevent benign actions. To determine the impact of this on existing extensions, we analyse 4 popular benign extensions and discover that most trigger thousands of suspicious actions. We believe our project would unfortunately break many benign extensions.

Finally, we evaluate the overhead of our changes and we observe large performance regressions. We describe the complexity of this project as a whole and mention the backwards compatibility of changes.

Conclusion

As discussed in Chapter 2, the Chrome extension security model has failed to prevent malicious extension behaviour within the Chrome browser. Existing analysis tools, whilst able to catch many malicious extensions, have not been able to provide security guarantees about extensions published on the Chrome Web Store (CWS) or extensions running within one's browser. The goal of this project was to guarantee the protection of browser users from popular threats without breaking many benign extensions or web-pages and without incurring unnecessary performance overheads.

We address the limitations of previous work and achieve most goals by proposing a novel extension to the Chrome security model that forces users to allow or prevent potentially malicious extensions actions that are intercepted at run-time. We relax a key goal so that instead of guaranteeing that malicious extension behaviour cannot occur, we guarantee that if it occurs, we prompt users to allow or prevent it. A modified Chrome browser successfully implements these suggestions but incurs substantial performance overheads. We also discover that users are not adequately equipped to classify suspicious extensions as malicious or benign which results in weakened protection and benign extensions breaking.

Achieving these goals was non-trivial due to the challenges of efficiently classifying actions as suspicious, guaranteeing the detection of extension operations, breaking minimal benign extensions and presenting suspicious extension actions in a user friendly manner. Modifying the Chrome browser also proved a challenging software engineering task that involved studying and contributing to complex segments of Chrome's C++ source code and multi-process architecture as a whole.

8.1 Summary of Work

We first summarise the work carried out in this project and compare it against our project objectives, which we presented in priority order in Chapter 1. Although equally important, this section excludes research work carried out Chapter 2, and instead focuses on project contributions.

We began by presenting a threat model where we assume extensions are attempting to execute attacks on web-pages via content scripts. We propose a novel solution that involves analysing extension behaviour at run-time and alerting users of suspicious extension actions. We contribute a list of Web API methods and general JavaScript operations which we deem suspicious and build a feature that allows users to configure this themselves.

We continue by describing the challenge of detecting and intercepting suspicious extension actions; we propose and implement modifications to the Chrome browser that solve this problem. Our solution guarantees sophisticated attackers cannot evade detection and guarantees we intercept all extension operations that we deem suspicious.

We design and actualise warning dialogs within the Chrome browser that alert users whenever an extension attempts to execute a suspicious operation and which provide users with the ability to allow or prevent these actions. We choose to block script execution while dialogs are displayed to prevent extensions from harming users. Furthermore, we propose multiple dialog features that provide additional information about actions in questions to help users better classify malice and therefore better protect themselves.

Warning dialogs have the potential to overwhelm users, thereby reducing their ability to correctly classify suspicious actions and limiting our browser's potential protection. We therefore spend considerable time white-listing benign events and operations performed on elements unattached to the DOM tree. We present users with the tool to always allow or prevent a certain type of action. Finally, we incorporate a simple sensitivity model whereby web-pages can mark elements and their data as more vulnerable to malicious extension attacks.

To conclude, we evaluate our modified Chrome browser against a contrived malicious extension and suspicious extensions publicly available on the CWS. We conduct a large survey of everyday browser users to assess the user interface and success of our dialogs and analyse popular extensions on the CWS to determine its impact. Subsequently, we evaluate the performance impact of our Chrome browser modifications and consider its backwards compatibility.

In terms of protecting users from Facebook hijacking, ad injection and user tracking threats carried out by sophisticated attackers, we have been largely successful. We guarantee our browser warns users of suspicious extension actions and provides them with the necessary tools to prevent malicious ones. Furthermore, our modifications do not directly break benign extensions.

However, our project has not been entirely successful as everyday users struggle to correctly classify suspicious extensions as malicious or benign. This results in weakened user protection and potentially breaks benign extensions. Although web-pages infrequently crash when executed within our modified browser, this is due to implementation errors and not a incompatible security model. Unfortunately, our modifications have incurred a significant performance overhead but we feel there is plenty of room for optimisations.

8.2 Future Work

We believe that this project has contributed significantly to the problem of detecting malicious extension behaviour in browsers and preventing it before it harms users. Nevertheless, the solutions we propose can be significantly improved in order to yield better results and better protect users. We suggest possible extensions to this project and list them in descending order of priority.

1. **Additional Information**. From our evaluation we found that survey respondents often didn't have enough information to classify extensions as malicious or benign. Future work should involve completing features to highlight elements (§5.6) and display additional information about operations (§5.8), or to introduce additional tools to help users classify suspicious operations as benign or malicious.

2. **User Interface**. As seen in our user survey (§7.3), many users don't fully understand and are risk-adverse to our warning dialogs. This resulted in far too many benign actions prevented and therefore

benign extensions potentially broken. Future work should be carried out to explore different dialog messages, user interface designs and mediums to better convey our warning message.

3. **Remembering User Decisions**. Currently, we provide users with the ability to have our browser remember their decisions to allow or prevent certain actions. However, this only lasts until the browser is restarted as these preferences are stored in-memory. This inconveniences users as they are re-asked to give permission to extensions every time they restart their browser. Similarly to how we configure suspicious actions (§3.10), we should persist user decisions to disk and give users more options for how long these wish to allow or prevent certain actions. This would reduce the number of warning dialogs displayed and should be explored further.

4. **Background Pages**. To prevent extensions proxy-ing malicious network requests through their background pages (§7.6), we should consider intercepting suspicious operations executed within extension background pages as well.

5. **Tracking Sensitive Data**. Our browser compares sensitive data previously read with data being send in network requests to detect sensitive data leakage (§6.4.3). Unfortunately, malicious attackers can trivially circumvent this by modifying or encrypting the sensitive data before uploading it. To prevent this from occurring, an extension to this project could track the data as it propagates through Chrome's V8 JavaScript engine by adding taint meta-data to every JavaScript `Value`.

6. **Sensitivity Model**. DOM elements represented within our modified browser can either be sensitive or not. However, elements are generally at risk from only a subset of suspicious extension actions. For example, a "like" button could be vulnerable to extensions triggering click events or reading attributes but not at risk from removal or style changes. Future work could fine-grain our model such that elements would declare the suspicious actions to which they are sensitive. This would in turn reduce the number of dialogs and false positives.

7. **Configurable Suspicious Actions**. Currently, we categorise a set of pre-defined Blink methods into sets that everyday users can (hopefully) understand. For example, the methods `Element` `appendChild`, `insertChild`, `replaceChild`, and `removeChild` are considered part of the "DOM mutation" set which has "Adds, removes or replaces an element" as a human-readable description. This provides a layer of abstraction and allows everyday users to decide to intercept all or none of the methods in the sets, without knowing the implementation details. A future extension to this project could be to give technical users fine-grained control over which Blink methods they intercept.

Bibliography

[1] W3Schools. Browser Information. http://www.w3schools.com/browsers/default.asp. [Online; accessed 10 January 2017].

[2] N. Carlini, A. P. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension Security Architecture. *Proceedings of the USENIX Security Symposium*, 2012.

[3] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. *Network and Distributed System Security Symposium (NDSS)*, 2010.

[4] R. Chipman, T. Ivonchyk, D. Man, and M. Voss. Security Analysis of Chrome Extensions. https://courses.csail.mit.edu/6.857/2016/files/24.pdf, 12 May 2016. [Online; accessed 8 January 2017].

[5] N. Jagpal, E. Dingle, J. P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. *Proceedings of the USENIX Security Symposium*, 2015.

[6] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. *Proceedings of the USENIX Security Symposium*, 2014.

[7] E. Egele, C. Kruegel, and E. Kirda. Dynamic Spyware Analysis. *Proceedings of the USENIX Security Symposium*, 2007.

[8] D. Hazal-Massieux. JavaScript Web APIs. https://www.w3.org/standards/webdesign/script. [Online; accessed 12 June 2017].

[9] Chromium Project. https://www.chromium.org/Home. [Online; accessed 18 January 2017].

[10] A. Laforge. Stable Channel Update. https://chromereleases.googleblog.com/2010/01/stable-channel-update_25.html, 25 January 2010. [Online; accessed 8 January 2017].

[11] Multi-process Architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture. [Online; accessed 19 June 2017].

[12] Site Isolation Project. https://www.chromium.org/developers/design-documents/site-isolation. [Online; accessed 17 May 2017].

[13] Inter-process Communication (IPC). https://www.chromium.org/developers/design-documents/inter-process-communication. [Online; accessed 19 June 2017].

[14] Blink. https://www.chromium.org/blink. [Online; accessed 27 May 2017].

[15] Web IDL interfaces. https://www.chromium.org/developers/web-idl-interfaces. [Online; accessed 27 May 2017].

[16] dpranke@chromium.org. Blink, Testing, and the W3C. https://www.chromium.org/blink/blink-testing-and-the-w3c. [Online; accessed 27 May 2017].

[17] J Robinson, A. Walker, V. Kokkevis, and N. Duca. Compositor Thread Architecture. https://www.chromium.org/developers/design-documents/compositor-thread-architecture. [Online; accessed 9 June 2017].

[18] A. Barth. How WebKit Works. https://docs.google.com/presentation/d/1ZRIQbUKw9Tf077odCh66OrrwRIVNLvI_nhLm2Gi__F0, 30 October 2012. [Online; accessed 10 June 2017].

[19] Overview. https://developer.chrome.com/extensions/overview. [Online; accessed 10 January 2017].

[20] Autoupdating. https://developer.chrome.com/extensions/autoupdate. [Online; accessed 10 January 2017].

[21] R. S. Liverani and N. Freeman. Abusing Firefox Addons. https://www.youtube.com/watch?v=vffa4FshXWY, 8 October 2009. [Defcon17; Online; accessed 8 January 2017].

[22] S. V. Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[23] P. L. Hgaret. Document Object Model (DOM). https://www.w3.org/DOM/#what, 19 January 2005. [Online; accessed 10 June 2017].

[24] Extension JavaScript APIs. https://developer.chrome.com/extensions/api_index. [Online; accessed 01 June 2017].

[25] Content Scripts. https://developer.chrome.com/extensions/content_scripts. [Online; accessed 17 January 2017].

[26] M. West. Content Security Policy Level 3. https://w3c.github.io/webappsec-csp/, 31 May 2017. [Online; accessed 17 January 2017].

[27] Content Security Policy (CSP). https://developer.chrome.com/extensions/contentSecurityPolicy. [Online; accessed 17 January 2017].

[28] Sandboxing Eval. https://developer.chrome.com/extensions/sandboxingEval. [Online; accessed 19 January 2017].

[29] M. West, A. Barth, and D. Veditz. Hash usage for script elements. https://www.w3.org/TR/2015/CR-CSP2-20150721/#script-src-hash-usage, 21 July 2015. [Online; accessed 19 January 2017].

[30] E. Protalinski. Malicious Chrome extensions hijack Facebook accounts. http://www.zdnet.com/article/malicious-chrome-extensions-hijack-facebook-accounts/, 26 March 2012. [Online; accessed 6 May 2017].

[31] L. Constantin. Google cracks down on ad-injecting Chrome extensions. http://www.pcworld.com/article/2904852/google-cracks-down-on-adinjecting-chrome-extensions.html, 1 April 2015. [Online; accessed 19 June 2017].

[32] I. Papagiannis, M. Migliavacca, and P. Pietzuch. Php aspis: Using partial taint tracking to protect against injection attacks. *2nd USENIX Conference on Web Application Development*, 2011.

[33] Common Vulnerabilities and Exposures. https://cve.mitre.org/. [Online; accessed 18 June 2017].

[34] I. Papagiannis, P. Watcharapichat, D. Muthukumaran, and P. Pietzuch. Browserflow: Imprecise data flow tracking to prevent accidental data disclosure. *Proceedings of the 17th International Middleware Conference*, 2016.

[35] Cat and mouse. https://en.wikipedia.org/wiki/Cat_and_mouse. [Online; accessed 04-June-2017].

[36] Google Chrome Web Store Developer Agreement. https://developer.chrome.com/webstore/terms#review. [Online; accessed 4 June 2017].

[37] The Navigator object. https://html.spec.whatwg.org/multipage/webappapis.html#the-navigator-object, 16 June 2017. [Online; accessed 16 June 2017].

[38] Cross-site request forgery. https://en.wikipedia.org/wiki/Cross-site_request_forgery. [Online; accessed 30 May 2017].

[39] A. Rybka and F. Rivoal. Browser Extension Community Group. https://www.w3.org/community/browserext/. [Online; accessed 5 May 2017].

[40] proberge@chromium.org. Chromium Origin World Code Review. https://codereview.chromium.org/1615523002/, January 2016. [Online; accessed 6 May 2017].

[41] chrome.webRequest. https://developer.chrome.com/extensions/webRequest. [Online; accessed 4 June 2017].

[42] Objective Development Software. Little Snitch 3. https://www.obdev.at/products/littlesnitch/index.html. [Online; accessed 15 June 2017].

[43] Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/. [Online; accessed 9 June 2017].

[44] S. Olsen. Nearly undetectable tracking device raises concern. https://www.cnet.com/news/nearly-undetectable-tracking-device-raises-concern/, 2 January 2002. [Online; accessed 11 June2017].

[45] Malicious Facebook Extension (CWS). https://chrome.google.com/webstore/detail/malicious-facebook-extens/nolbbgleoigmchiplhfjipbcboapdfba. [Online; accessed 14 June2017].

[46] M. Cypher. Malicious Facebook Extension (GitHub). https://github.com/mikeecb/malicious_facebook_extension/. [Online; accessed 26 May 2017].

[47] M. G. Siegler. Chrome Appears To Have Hit 10,000 Extensions, Inching Closer To Firefox. https://techcrunch.com/2010/12/10/chrome-extension-numbers/, December 10 2010. [Online; accessed 16 June 2017].

[48] http://getmyscript.com. ToolKit For Facebook. https://chrome.google.com/webstore/detail/toolkit-for-facebook/fcachklhcihfinmagjnlomehfdhndhep, 7 December 2016. [Online; accessed 26 May 2017].

[49] D. Nguyen. Like all, plus all, favorite all. https://chrome.google.com/webstore/detail/like-all-plus-all-favorit/eadbjcgjgmdijbagjakgemgkjaiadane, 27 May 2017. [Online; accessed 26 May 2017].

[50] fbpv.info. Social Profile view notification extension. https://chrome.google.com/webstore/detail/social-profile-view-notif/pegkceflonohbcefcbflfpficfkmpeod, 13 June 2017. [Online; accessed 25 May 2017].

[51] Mechanical Turk. https://www.mturk.com/mturk/welcome. [Online; accessed 04 June 2017].

[52] M. Cypher. Suspicious Extension Action User Survey. https://docs.google.com/spreadsheets/d/1gtH-zLfzBQue-50O-xc9j8cQh63izrLVtOQ951YQaLc/edit?usp=sharing. [Online; accessed 04 June 2017].

[53] grammarly.com. Grammarly for Chrome. https://chrome.google.com/webstore/detail/grammarly-for-chrome/kbfnbcaeplbcioakkpcpgfkobkghlhen, 14 June 2017. [Online; accessed 14 June 2017].

[54] lastpass.com. LasttPass: Free Password Manager. https://chrome.google.com/webstore/detail/lastpass-free-password-ma/hdokiejnpimakedhajhdlcegeplioahd, 14 June 2017. [Online; accessed 14 June 2017].

[55] Google. Google Dictionary (by Google). https://chrome.google.com/webstore/detail/google-dictionary-by-goog/mgijmajocgfcbeboacabfgobmjgjcoja, 19 April 2016. [Online; accessed 14 June 2017].

[56] Transfusion Media. StayFocusd. https://chrome.google.com/webstore/detail/stayfocusd/laankejkbhbdhmipfmgcngdelahlfoji, 17 May 2017. [Online; accessed 14 June 2017].

[57] Perf Regression Sheriffing. https://chromium.googlesource.com/chromium/src/+/master/tools/perf/docs/perf_regression_sheriffing.md. [Online; accessed 27 May 2017].

[58] Chrome Performance Dashboard. https://chromeperf.appspot.com/. [Online; accessed 27 May 2017].

[59] Telemetry. https://catapult.gsrc.io/telemetry. [Online; accessed 27 May 2017].

[60] Selection API. https://www.w3.org/TR/selection-api/, 9 May 2017. [Online; accessed 7 June 2017].

[61] treib@chromium.org. Chromium Review: e9399f008302fcb62a257e67951497290bd7e93e. https://chromium.googlesource.com/experimental/chromium/src/+/e9399f008302fcb62a257e67951497290bd7e93e. [Online; accessed 12 June 2017].

Suspicious Extension Actions

```
enum class SuspiciousExtensionAction {
   EVENT, // EventTarget.dispatchEvent, Element.click, ...
   NETWORK_REQUEST, // XMLHttpRequest.send
   DOM_MUTATION, // Node.appendChild, Node.removeChild, ...
   ELEMENT_DESCRIPTION_MUTATION, // Element.setAttribute("data-sensitive", ...)
   READ_ELEMENT,
   ELEMENT_MUTATION,
   CREATE_ELEMENT, // Document.createElement
   NETWORK_RESPONSE, // XMLHttpRequest.onreadystatechange
   NAVIGATOR // window.navigator
};

class SuspiciousActionConfig {
  public:
    typedef base::Callback<void(bool)> IsActionSuspiciousCallback;

    SuspiciousActionConfig(content::BrowserContext* browser_context);

    // Determine if an extension action should be considered suspicious.
    // Calls callback(true) if it should and callback(false) otherwise
    void IsActionSuspicious(
      const std::string& extension_id,
      SuspiciousExtensionAction action,
      const IsActionSuspiciousCallback& callback
    );

    void SetActionSuspicious(
      const std::string& extension_id,
      SuspiciousExtensionAction action,
      bool is_suspicious
    );

  private:
    std::string ExtensionActionToString(SuspiciousExtensionAction action);

    // This is a callback executed when IsActionSuspicious reads from data store.
    // Converts Values read from data store to bools and executes
    // IsActionSuspiciousCallback
    void IsActionSuspiciousReadCallBack(
      const std::string& extension_id,
      SuspiciousExtensionAction action,
      const IsActionSuspiciousCallback& callback,
      callbackstd::unique_ptr<base::Value> value
    );

    bool IsActionSuspiciousByDefault(SuspiciousExtensionAction action);

    extensions::StateStore* state_store_;

  DISALLOW_COPY_AND_ASSIGN(SuspiciousActionConfig);
};
```

Listing 21: The `SuspiciousActionConfig class` definition that wraps the extension state store.

```
wrapper.setupColumn('details', '.suspicious-actions-link', 'click', function(e) {
    if (!this.suspiciousActionsPromptIsShowing_) {
        chrome.developerPrivate.showSuspiciousActionsDialog(
            extension.id,
            function() {
                this.suspiciousActionsPromptIsShowing_ = false;
            }.bind(this)
        );
        this.suspiciousActionsPromptIsShowing_ = true;
    }
    e.preventDefault();
});
```

Listing 22: JavaScript snippet that triggers C++ code to open a custom dialog.

```
enum ReadResponsePermission {
    kPromptUser, // Ask user if extension has permission to read response
    kAllow, // Allow an extension to read the response without asking user
    kDisallow, // Disallow an extension to read the response without asking user
};
ReadResponsePermission can_extension_read_response_;
```

Listing 23: An `enum` that represents the possible states our browser can take when deciding to alert users of extension that attempt to read network responses.

```
// Read an extension action's current state and select or un-select the checkbox
void ExtensionSuspiciousActionsView::InitCheckbox(views::Checkbox* checkbox) {
  checkbox->set_listener(this);
  SuspiciousExtensionAction action = GetExtensionActionForCheckbox(checkbox);
  suspicious_action_config_->IsActionSuspicious(
    extension_->id(),
    action,
    base::Bind(&views::Checkbox::SetChecked, base::Unretained(checkbox))
  );
}

// Whenever a checkbox is selected or un-selected, update the configuration
void ExtensionSuspiciousActionsView::ButtonPressed(views::Button* sender,
                                                   const ui::Event& event) {
  views::Checkbox* checkbox = (views::Checkbox*) sender;
  suspicious_action_config_->SetActionSuspicious(
    extension_->id(),
    GetExtensionActionForCheckbox(checkbox),
    checkbox->checked()
  );
}
```

Listing 24: Whenever our suspicious action configuration dialog is displayed, we load the configuration using the `ExtensionSuspiciousActionsView::InitCheckbox` which reads from the `SuspiciousActionConfig` state store wrapper. Whenever users change the configuration, we write back to this state store.

Detecting Extension Actions

```
bool Document::isCallerExtension(LocalFrame* frame) {
  v8::Isolate* isolate = v8::Isolate::GetCurrent();
  if (isolate->InContext()) {
    v8::Local<v8::Context> context = isolate->GetCurrentContext();
    // If current world is an isolated world, assume the caller is an extension.
    if (!context.IsEmpty()) {
      return DOMWrapperWorld::world(context).isIsolatedWorld());
    }
  }
  // By default, we assume the caller is not an extension
  return false;
}
```

Listing 25: A method to determine if the caller is an extension by checking if execution is occuring within the context of an isolated world.

```
DOMWrapperWorld* originWorld() {
    if (isIsolatedWorld()) {
      return this;
    }
    if (!originWorldStack().isEmpty()) {
      return originWorldStack().back().get();
    }
    return nullptr;
};

bool isOfIsolatedWorldOrigin() {
    DOMWrapperWorld* origin = originWorld();
    return origin ? origin->isIsolatedWorld() : false;
};
```

Listing 26: Methods to determine the origin world and whether or not it is an isolated world.

```
class OriginWorldScope {

    public:
        explicit OriginWorldScope(PassRefPtr<DOMWrapperWorld> originWorld)
            : m_hasPushed(false) {
            if (originWorld) {
                // "Push" the origin world onto the origin world stack
                DOMWrapperWorld::originWorldStack().append(originWorld);
                m_hasPushed = true;
            }
        }

        ~OriginWorldScope() {
            // "Pop" the origin world to avoid tainting the entire document.
            if (m_hasPushed) {
                DOMWrapperWorld::originWorldStack().pop_back();
            }
        }

    private:
        bool m_hasPushed;
};
```

Listing 27: A class to safely push origin worlds onto the origin world stack.

User Prompts

```
// content/renderer/render_frame_impl.cc
bool RenderFrameImpl::DisplayExtensionActionPrompt(
  const GURL& extension_url,
  const blink::WebString& message,
  const SuspiciousExtensionAction& extension_action,
  const ExtensionActionIsSensitive& sensitive_element
) {
  bool success = false;
  // Send a message to the browser process to display a suspicious extension action
  // dialog and block until the user closes it.
  // If the user chooses to allow the action, return true. Otherwise, return false
  Send(new FrameHostMsg_DisplayExtensionActionPrompt(
    routing_id_, extension_url, message.Utf16(), extension_action.Utf16(),
    sensitive_element, frame_->GetDocument().Url(), &success
  ));
  return success;
}

// third_party/WebKit/Source/core/dom/Document.cpp
bool Document::userAllowsSuspiciousExtensionAction(
  LocalFrame* frame,
  const String& message,
  const String& additional_information,
  const SuspiciousExtensionAction& extension_action,
  const ExtensionActionIsSensitive& sensitive_element
) {
  v8::Isolate* isolate = v8::Isolate::GetCurrent();
  if (isolate->InContext()) {
    v8::Local<v8::Context> context = isolate->GetCurrentContext();
    if (!context.IsEmpty()) {
      // Determine what world we are currently executing in
      DOMWrapperWorld& world = DOMWrapperWorld::World(context);
      if (world.IsOfIsolatedWorldOrigin()) {
        // Need to save the result of utf8 on stack or it will be free'd
        CString cs = world.GetOriginWorld()
                          ->IsolatedWorldSecurityOrigin()
                          ->ToString().Utf8();
        GURL* extension_url = new GURL(base::StringPiece(cs.data()));
        Page* page = frame->GetPage();
        return !(page && !page->GetChromeClient().DisplayExtensionActionPrompt(
                                                *extension_url,
                                                frame,
                                                message,
                                                extension_action,
                                                sensitive_element))
      }
    }
  }
  // By default, we should allow operations that haven't been prevented by the user
  return true;
}
```

Listing 28: Intercepted suspicious operations execute the `Document::-userAllowsSuspiciousExtensionAction`, which sends an IPC message to display a dialog, to determine whether they should continue or throw an exception.

Reducing Dialogs

```
// Adds all sensitive data within the node (including its children) to the
// SensitiveData HashSet
void ContainerNode::markSensitiveDataAsRead(ContainerNode& node,
                                            bool only_children) {
  // If only_children is true, only add sensitive data from the node's children
  // and not the node itself.
  if (!only_children) {
    // If the element is not sensitive, there is no sensitive data
    if (!ToElement(node).isSensitive()) {
      return;
    }
    markAttributesAsRead(ToElement(node));
  }
  for (Node* child = node.firstChild(); child; child = child->nextSibling()) {
    if (!child->IsElementNode()) {
      continue;
    } else if (child->IsContainerNode()) {
      markSensitiveDataAsRead(*blink::ToContainerNode(child), false);
    } else {
      markAttributesAsRead(*ToElement(child));
    }
  }
}

// Adds all attribute values to the SensitiveData HashSet
void ContainerNode::markAttributesAsRead(Element& element) {
  for (AttributeCollection::iterator it = element.Attributes().begin() ;
       it != element.Attributes().end();
       ++it) {
    element.SensitiveData().insert(it->Value());
  }
}
```

Listing 29: The `ContainerNode::markSensitiveDataAsRead` method is called any time a script attempts to read an extensions inner or outer HTML.

Evaluation

```
chrome.runtime.onMessage.addListener(function (request, sender, sendResponse) {
  var splitRequest = request.split("-");
  var action = splitRequest[0];
  var channel = splitRequest[1];
  switch (channel) {
    case "contentScript":
      performAction(action); break;
    case "scriptTag":
      var s = document.createElement('script');
      s.textContent =
        "var s = document.createElement('script');" +
        "s.textContent = '" + getStringToPerformAction(action) + "';" +
        "document.body.appendChild(s);";
      document.body.appendChild(s); break;
    case "event":
      var s = document.createElement('script');
      s.textContent =
        "document.addEventListener(\"DOMContentLoaded\", function() {" +
          getStringToPerformAction(action) +
        "});";
      document.body.appendChild(s);
      document.dispatchEvent(new Event("DOMContentLoaded")); break;
    case "domTimer":
      var s = document.createElement('script');
      s.textContent =
        "window.setTimeout(function() {" +
          getStringToPerformAction(action) +
        "}, 1000);";
      document.body.appendChild(s); break;
    case "promise":
      var s = document.createElement('script');
      s.textContent =
        "Promise.resolve().then(function() {" +
          getStringToPerformAction(action) +
        "});";
      document.body.appendChild(s); break;
    case "mutationObserver":
      var s = document.createElement('script');
      s.textContent =
        "var target = document.getElementsByTagName(\"p\")[0];" +
        "var observer;" +
        "observer = new MutationObserver(function(mutations) {" +
          getStringToPerformAction(action) +
        "});" +
        "var config = { attributes: true, childList: true, characterData: true };" +
        "observer.observe(target, config);" +
        "target.setAttribute(\"data-random-attribute\", true);";
      document.body.appendChild(s); break;
    default: break;
  }
});
```

Listing 30: An extension message listener that executes suspicious actions in the isolated or main world.