Exposing and Addressing Security Vulnerabilities in Browser Text Input Fields

Asmit Nayak*, **Rishabh Khandelwal***, **Kassem Fawaz**

University of Wisconsin – Madison

Abstract

In this work, we perform a comprehensive analysis of the security of text input fields in web browsers. We find that browsers' coarse-grained permission model violates two security design principles: least privilege and complete mediation. We further uncover two vulnerabilities in input fields, including the alarming discovery of passwords in plaintext within the HTML source code of the web page. To demonstrate the real-world impact of these vulnerabilities, we design a proof-of-concept extension, leveraging techniques from static and dynamic code injection attacks to bypass the web store review process. Our measurements and case studies reveal that these vulnerabilities are prevalent across various websites, with sensitive user information, such as passwords, exposed in the HTML source code of even high-traffic sites like Google and Cloudflare. We find that a significant percentage (12.5%) of extensions possess the necessary permissions to exploit these vulnerabilities and identify 190 extensions that directly access password fields. Finally, we propose two countermeasures to address these risks: a bolt-on JavaScript package for immediate adoption by website developers allowing them to protect sensitive input fields, and a browser-level solution that alerts users when an extension accesses sensitive input fields. Our research highlights the urgent need for improved security measures to protect sensitive user information online.

1 Introduction

Browser extensions are small applications that enhance the capabilities of web browsers and improve the user experience. They can add new features, modify web page content and automate tasks. Canonical examples of extensions include password managers, productivity tools such as Pocket, and ad blockers that modify webpages through well-defined APIs to prevent advertisements. The extensions achieve this by accessing the contents of the webpages and manipulating the contents of the webpages by executing JavaScript code. This access to web page content also allows the extension to retrieve private content, such as emails or banking details.

Prior work has shown that it is possible to exploit this access to read sensitive user data such as emails [16, 24], passwords [2, 24], and even perform phishing attacks [2, 19, 24]. These attacks can either use: a) *Static Code Injection* where the attackers add the malicious code in the extension; or b) *Dynamic Code Injection* where the code is loaded dynamically from a remote server and executed at run time. Static code injection is impractical as they can be detected by static code analysis [4, 5, 26, 28]. Dynamic code injection bypasses the static security checks as the code is

---

*Equal Contribution

injected at run time, and thus, is harder to detect [13, 22]. To address this vulnerability posed by dynamic code injection, Google introduced new regulations that disallowed the execution of remotely injected code.

However, as we show in Section 4, bypassing the protective measures and extracting sensitive information using the extension is possible. The attack is feasible because the interaction between the extensions and the web pages has not changed. The extensions can still access entire contents of the web pages, including text input fields where users may enter sensitive information such as passwords, Social Security Numbers (SSN), and Credit Card information.

In this work, we contribute to understanding the security of text input fields after Google's new regulations by performing an experimental analysis of the interactions between browser extensions and web pages. We focus on text input fields because input fields are often used for sensitive data such as usernames, passwords, credit card numbers, and SSNs. Given the sensitive nature of the data, ensuring that malicious actors cannot access input fields is of utmost importance. If an attacker can access or manipulate the data in these fields, they can potentially steal private user information, impersonate the user, or commit financial fraud. Exposure of this data could potentially also be harvested by automated scripts or bots that scan webpages for such vulnerabilities.

To conduct a systematic experimental analysis after introducing new regulations, we first analyze the interactions between web extensions and web pages. We focus on the browser's permission model to isolate extensions (Section 3). We find that due to browsers' coarse-grained permission model, a lack of security boundary exists between the extension and the web page. This lack of boundary allows the extension to freely interact with and manipulate the HTML elements, including HTML input elements. Concretely, we have discovered that the extension permission model violates two security principles: 1) least privilege and 2) complete mediation. This allows a malicious extension to access sensitive user information without users' knowledge.

To demonstrate the concrete harm posed to users, we first identify two vulnerabilities with input fields, including a novel vulnerability where sensitive information like passwords is visible in plain text in the HTML source code. Next, we explore how an attacker can exploit these vulnerabilities and gain access to private user information by designing a proof-of-concept extension(Section 4). During the development of the extension, we introduce a novel attack to extract information from input fields. We leverage static and dynamic code injection techniques to create a hybrid attack that bypasses the webstore review process.

Next, we conduct measurements (Section 5) and case studies (Section 6) to understand the prevalence of these vulnerabilities. We observe that both vulnerabilities exist within the input fields of the login forms across the websites. Specifically, we can extract usernames and passwords from the login pages of 100% of the websites. Alarmingly, we find that on 15% of the websites, passwords are present in plain text in the HTML source code. This implies that any entity, including extensions and third-party JavaScript, that can view the page source can extract users' passwords. This vulnerability is present in popular websites with high traffic, such as Google, Cloudflare, and Doordash. We further analyze the permissions requested by the extensions and find that 12.5% of extensions have the necessary permissions to exploit the vulnerabilities discovered in this work. We also performed dynamic analysis by instrumenting the extensions and identifying 190 extensions directly accessing password fields.

Contributions. Our key contributions are summarized below:

- We perform an experimental security analysis of the security of text input fields and find that the permission model violates the principle of least privilege and complete mediation. We also find two novel vulnerabilities in input fields, including one where passwords are present in the HTML source of the webpage.

- We design proof-of-concept attacks to exploit these vulnerabilities and show the practicality of the attack by submitting the extension to the web stores.

- We perform measurements and case studies to show the prevalence of these vulnerabilities in the wild. We find the vulnerabilities across all websites indicating that websites fail to protect users' sensitive information.

Finally, we propose two countermeasures to mitigate the security risks from the observed vulnerabilities. In our *Bolt-on* solution, we provide a JavaScript package that website developers can adopt today to mitigate the attacks (Section 7.1). The package introduces a new input type *SecureInput* that uses WeakMaps[1] to store sensitive values in private variables. We also propose a more fundamental browser-level solution (Section 7.2) by instrumenting chromium to alert users when an extension accesses sensitive input fields.

Highlighting Severity of Vulnerability. To highlight the severity of the vulnerability, we provide an example. Consider an adversary to steal login credentials from the users. Assume that the adversary can acquire a popular browser extension (as shown possible in [1, 11, 13, 23]) - *Honey*[2] with over 17M users. We note here that popular websites such as `google.com` and `cloudflare.com` also have the vulnerability where the passwords are visible in plain text in HTML source. As the extension has permission to run on all websites, the adversary can inspect the source code of the Google login page and extract the login credentials for millions of users.

2 Background

We provide a brief background about the HTML fundamentals, including HTML input elements, HTML tree structure, accessibility of elements through JavaScript, and iframes. We also discuss the permission models for a browser extension, as well as the extension architecture.

2.1 HTML Fundamentals

HTML Input Elements: Input fields, marked by the HTML tag `<input>`, serve as the most basic avenue for users to input data into a webpage. These elements constitute a critical component of web forms[3]. The rendering of the input field on the screen is dependent on the type of input field. For instance, the checkbox' type establishes a checkbox, the text' type generates a regular textbox, and the 'password' type, generally used for sensitive content, conceals the text shown on the textbox. We note that ensuring that input fields cannot be accessed by malicious actors is crucial, as exposure of sensitive data can be harvested by automated scripts or bots.

DOM Tree: While rendering a webpage, the browser constructs a Document Object Model (DOM) of the page. This DOM, composed of nodes and objects, replicates the webpage as a tree structure, known as the DOM Tree. The tree's root initiates with the `<html>` element.

The DOM API allows JavaScript (JS) to access any component of the DOM tree and adjust its attributes or content. This involves selecting the HTML element, which can be accomplished in several ways, such as by *ID*, *tag name*, or *xpaths*. Moreover, the DOM API also enables JS to insert new elements into the DOM tree or discard existing ones.

---

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap`
[2] `https://honey.com/`
[3] `https://developer.mozilla.org/en-US/docs/Learn/Forms`

Eval Statements: JavaScript allows the execution of strings as JavaScript code using the `eval()` function. While `eval` can be legitimately used to generate code based on specific conditions dynamically, its use is generally viewed as a security risk due to its potent nature. Richards et al. [20] performed a large-scale study on the use `eval` statements in web applications.

Extensions have been known to use `eval` statements to inject code into webpages dynamically. Kapravelos et al. [13] found more than 400 Chrome extensions using `eval` statements with inputs exceeding 128 characters in length. Similarly, Wang et al. [25] discovered 145 extensions on the Firefox add-on store that contain the `eval` statement.

Inline Frames (iframe): An inline frame, or iframe, is a special type of HTML element that can load another HTML page. Each iframe runs on its own context and cannot directly interact with the parent object or other iframes. Iframes are majorly used to load content from another page, this functionality can lead to security risks as misuse of iframes leads to a security vulnerability for users.

2.2 Browser Extensions

Browser extensions provide a customizable user experience, adjusting web content to align with user preferences. Their functionalities range from aesthetic alterations like background color changes to ad removal and the automatic deletion of tracking cookies. To accomplish these tasks, extensions require access to browser resources, APIs, and webpage content.

Permission Models: Browser extensions request permissions for the resources they require for their functionality. requested via the manifest.json file, can be of two types: Host permissions and API permissions. Host permissions enable extensions to inform the browser about the websites they need to access, allowing extensions to access content from these specified sites. An extension can request access to a specific set of websites, or request access to all urls. API permissions, on the other hand, provide extensions with the capability to interact with specific WebExtension APIs, such as `browser.storage` or `browser.cookies`.

Content Scripts and Background Pages: Extensions are composed of two main components: content scripts and background pages (or service workers). Content scripts are static JavaScript files that are automatically loaded with a webpage. These scripts run in the webpage context as an extension to the DOM tree. Background pages, in contrast, are not loaded with each website; they react to browser events or carry out WebExtension API-based actions. Although content scripts have access to certain WebExtension API functions, their access is limited in scope. To leverage the full extent of the APIs, content scripts communicate with the background page via message passing.

While extensions can load static JavaScript as content scripts, they can also use a mix of host permissions and browser APIs to inject JavaScript into webpages programmatically. For example, an extension can request no websites under content scripts but then request `scripting` and host permissions on all websites to inject content scripts on websites dynamically. Furthermore, content scripts, without host permissions, must comply with website-defined cross-origin restrictions, unlike scripts injected via host permissions and API. This compliance limits their interaction with external entities, although they can still send and receive messages from the extension's background script.

3 Security Landscape of Extensions

We conduct a comprehensive security analysis to understand potential design issues in the accessibility of input fields by extensions in Google Chrome, Mozilla Firefox, and Safari browsers. We focus on these browsers due to their widespread popularity and usage. We introduce a practical

threat model and a systematic methodology for evaluating the potential risks posed by malicious extensions. We note here that while JavaScript running on the page can also access the HTML elements in a similar way, we restrict our analysis to extensions in this work as they operate within a controlled environment constrained by browsers' policies which allows us to identify and analyze potential security risks.

3.1 Extension Priviledges

An extension can request different permissions to perform its functionalities. The permission model for browsers is largely static i.e. users agree to an extension's host and API permissions at install time, granting the extension a pre-defined set of capabilities. The user can also approve the websites where the extension can inject content scripts at install time. However, through extension settings, the browser provides a high-level control that allows users to prevent extensions from injecting scripts on certain websites.

The existing permissions framework across all browsers exhibits a coarse-grained approach, particularly with respect to access to web page content. The interaction of extensions with the HTML DOM tree is shown in Figure 1. Once an extension is loaded on a webpage, it has unrestricted access to all elements on the page, including sensitive input fields. Such an extension, essentially a JavaScript program loaded into the DOM tree of the page, can access and potentially manipulate any data in the input fields on the page (Figure 1). This coarse-grained control contrasts with the fine-grained access control for certain software and hardware resources, such as location information or file storage.

One consequence of this coarse-grained model is the absence of a security boundary between the extension and the HTML elements (Figure 1). This contrasts iframes governed by strict cross-origin policies that restrict access to the parent DOM tree and thus lie outside the security boundary. We examine this coarse-grained permission system in detail in Section 4 and show that it has security design issues that can violate security design principles and lead to unwanted access by extensions. **Isolating Extensions.** Browsers follow a set of rules to isolate extensions to their own environment. Before December 2020, Manifest V2 (MV2) governed the extensions' interactions within the browser's boundaries. Previous research [] identified MV2's limitations and showed how extensions can bypass it, raising security issues. One significant MV2 security loophole was allowing `eval()` statements, enabling extensions to execute any external JavaScript without any checks. This lead to attacks such as iframe-based phishing and password stealing [19].

3.2 Existing Threats

Over the last decade, researchers have described various attacks employed by malicious extensions to extract sensitive data from input fields [6, 18, 24, 25]. These attacks can be broadly classified into static and dynamic code-based attacks. A summary of these works is shown in Table 1.
Static Code Attacks: In these attacks, the malicious code is statically present in the extensions' source code. Prior works [18, 24, 25] have proposed numerous attack vectors to exploit access of extension to the webpage and extract private information. For instance, Varshney et al. [24] have added an `eventListener` to log all the keystrokes. Similarly, Eriksson et al. [6] modify the password field to reveal the password values. We note that the underlying cause for these attacks is the lack of security boundary between the extension and the webpage, as discussed in Section 3.1.

While these attacks are theoretically possible, they are also known to be impractical as they can be detected via static code analysis [5, 8, 24, 26, 28]. This is evident because none of the existing works performing static code attacks have submitted the extensions to the web store. Therefore,
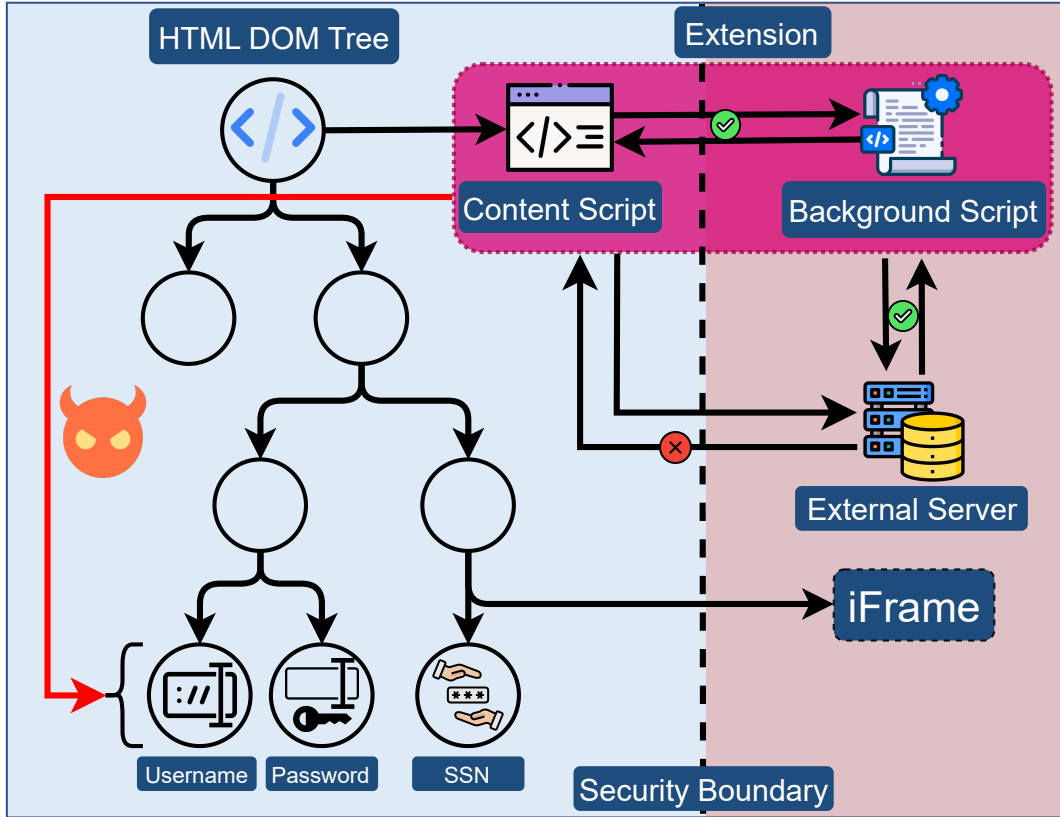
Figure 1: The figure illustrates the different contexts in which extensions and iframes exist with respect to a webpage. An extension's content script exists in the same context as the webpage, while an iframe is separated from the webpage by the website's Content Security Policy (CSP). Additionally, the content script can access any element of the DOM tree, including sensitive user data

these approaches do not test the viability of using extensions to compromise actual users that install extensions via web stores.

```
1  ...
2  $(document).keypress(function(e) {
3     ...
4     keyHistory += String.fromCharCode(e.which);
5     ...
6  });
7  ...
```

Figure 2: Static Code Example

Dynamic Code Attacks: Dynamic code attacks involve retrieving the malicious code from an external server and then executing it into the target webpage [18, 19, 25]. For example, Perrotta et al. [19] inject malicious remote code to perform a phishing attack.

```
1  ...
2  fetch('evil_server_url')
3    .then(response => response.text())
4    .then(maliciousCode => {
5      eval(maliciousCode);
6    });
7  ...
```

Figure 3: Dynamic Code Example

Figure 3 shows a skeleton code snippet where a malicious code hosted at a remote server is executed using the `eval()` function, which allows for the execution of JavaScript code represented as strings. These attacks can also include obfuscated code to prevent detection against dynamic analysis, such as remaining dormant and capturing data after a certain interval.

Dynamic code attacks primarily exploit the remote code execution privileges provided to the extension. The remote code can not be vetted using code analysis methods, and thus are harder to detect as the injected code cannot be analyzed during the review process []. To address this vulnerability, Chrome recently introduced new regulations that ban remote code execution. We discuss this in Section 3.3.

3.3   Security landscape after Manifest V3

In December 2020, Google Chrome introduced Manifest V3 (MV3), bringing substantial changes in privacy, security, and performance. From a security standpoint, MV3 introduced `declarativeNetRequest` API for network request modifications and discontinued the `webRequest` API, disallowing extensions to modify network requests in real-time, closing a major loophole [7]. MV3 also prohibited the execution of remotely hosted code and the use of eval statements. This vulnerability was exploited by attackers [19] to extract sensitive user data, as discussed in Section 3.2.

Despite MV3's intended advancements in user privacy and security, content scripts' operations remain unchanged. This maintains the lack of security boundary between the extension and web page and allows an extension to be loaded on the DOM tree and gain unrestricted access to the webpage, posing security risks for the users. We use this vulnerability to design our attack in Section 4.

Adoption of MV3. Following Google's lead, Mozilla updated its browser ecosystem to MV3. By May 2023, Safari, Firefox, and all Chromium-based browsers, such as Microsoft Edge, had integrated support for MV3. However, only Chromium browsers have stopped accepting MV2 in their respective web stores, with Safari and Firefox continuing to allow MV2-based extensions.

Impact on Review Process. Before MV3, Chrome's web store review process involved using malicious extension-detecting systems like WebEval [12]. WebEval utilized both static and dynamic analysis along with developer-centric heuristics (like email, age, etc.) to determine if an uploaded extension was malicious. However, as demonstrated by [19], extensions can bypass this system, enabling the successful upload of a malicious extension to the web store. After MV3, Google prohibited all remote code execution and mandated that all code be included within extensions as this permits more reliable and efficient reviews of extensions submitted to the web store [7].

3.4   Threat Model

In our threat model targeting browser extensions, we assume that the adversary adheres strictly to the guidelines provided in Chrome's MV3, and does not rely on side-channel attacks. Consequently,

our assumption restricts the analysis to Chrome and Chromium-based browsers. This is a reasonable assumption because MV3 provides the latest framework, with other browsers adopting it as the standard framework (Section 3.3). We also assume that the extension is uploaded to the webstore, and passes the Chrome review process. All changes made to the extension go through the review process of the webstore. For example, if an adversary acquires an existing extension and adds malicious code while updating the extension, the updates are passed through the review system. Finally, we assume that users directly download and install these extensions from the Chrome Web Store, as opposed to third-party installations. This ensures that the attack is practical, bypasses Google Chrome's review process, and the extension is available to users via the website, Once installed, users interact with these extensions in their regular browsing environment, unaware of the underlying threat.

3.5 Security Analysis Methodology

We conduct an experimental security analysis on browsers to examine how malicious extensions (as defined by our threat model) can potentially exploit the input fields. Specifically, we look for potential vulnerabilities in the latest browser extension regulation, MV3, and analyze if an extension with minimal permissions under MV3 can access sensitive input fields. Should such a possibility exist, we further explore if and how these extensions might access the DOM without the need for concealed code injections. With the attack, we discuss how it stems from the security design issues in the permission system, how it violates the security principles, and how it can lead to the exposure of sensitive information. We present our findings in Section 4.

To evaluate the *practicality* of extensions exploiting the vulnerabilities, we construct a proof-of-concept malicious extension to extract login credentials on websites. The extension requires enabling the content script to run on all web pages. Once installed, this extension accesses the input fields of login pages and extracts the username and password. We then submit the extension to Chrome Webstore where it bypasses the security checks and the review process. Finally, we run this extension on the top 10K domains from Tranco and extract login details. We discuss the measurement study in Section 5.1. We also perform case studies to understand the implications of the vulnerability on input fields consisting of SSNs and Credit Card information. We discuss these results in Section 6.

We measure *prevalence* of malicious extensions in two ways. First, we analyze the extensions' potential ability to access sensitive input fields. We collect the requested permissions of the extensions and examine how many apps have the necessary permissions to extract sensitive information. Second, we perform static and dynamic analyses of extensions to identify if the extension is actually accessing the input fields. We present a detailed analysis of this measurement in Section 5.2. It's important to note here, that this analysis does not aim to identify malicious extensions but to highlight that extensions currently have unrestricted access to sensitive fields, thereby potentially exposing sensitive data.

4 Attack Design

We analyze the interaction of browser extensions with sensitive input fields to identify potential vulnerabilities. We then design attacks to exploit the observed vulnerabilities and develop a proof-of-concept extension capable of extracting sensitive information within our threat model. We also upload our extension to the web store, bypassing the security measures in place during the review process. Finally, we analyze the root causes of attack success, mapping the vulnerabilities to violations of security principles.

Paper	Attack	Summary	Permissions	Content Script	Uploaded to Web-Store	Type of Attack
Liu et al. [16]	Email Spamming	Snoop on users' legitimate email credentials to send them spam emails.	tabs, http://*/*, https://*/*	http://*/*, https://*/*	✘	Static
	DDoS Attack	Perform cross-site HTTPS request to launch DDoS attack on a victim webpage	tabs, http://*.yahoo.com/*	http://*.yahoo .com/*	✘	Static
	Password Sniffing	Steal user's email ID and password from login pages	tabs, https://online. citibank.com/*	https://online. citibank.com/*	✘	Static
Bauer et al. [2]	iframe-based Password Stealing	Load a victim page in an iFrame and steal auto-filled credentials.	http://*/*, all_frames, webRequestBlocking	http://*/*, https://*/*	✘	Static
Perrotta et al. [19]	iframe-based Phishing	Inject malicious remote code into a website and perform a phishing attack.	N/A	http://*/*, https://*/*	✔	Dynamic (uses eval)
Varshney et al. [24]]	Keylogger	Log all keystrokes and send them via SMS	http://*/*, https://*/*	http://*/*, https://*/*	✘	Static
	Credential Stealing	Automatically record autofilled user login info	http://*/*, https://*/*	http://*/*, https://*/*	✘	Static
	Phishing	Stealthily redirect victim to a target website to perform phishing attack	tabs, http://*/*, https://*/*	Uses background scripts in conjunction with tabs API	✘	Static
	Email Spying	Malicious Extension's content script reads the content of emails	N/A	http://mail.google.com, https://mail.google.com	✘	Static
Our Paper	Sensitive Data Sniffing	Read the value of a specific input field, as specified by a remote server.	None	<all_urls>	✔	Hybrid

Table 1: Summarizing related works and their attacks against ours.

4.1 Identifying Vulnerabilities

As discussed in Section 2, when an extension is loaded onto a website, it is integrated into the DOM tree, obtaining unrestricted access to all DOM elements via the DOM APIs. This exposes a critical security issue – the lack of a security boundary between the extension and the rest of the DOM tree, as shown in Figure 1. Unlike iframes, which have isolated DOM trees, extensions face no restrictions once permitted to operate on a page, resulting in a broad attack surface. From this design, we discover two vulnerabilities, labeled as *Type-A* and *Type-B*. Illustrations of websites

(a) *Type-A* vulnerability on google.com
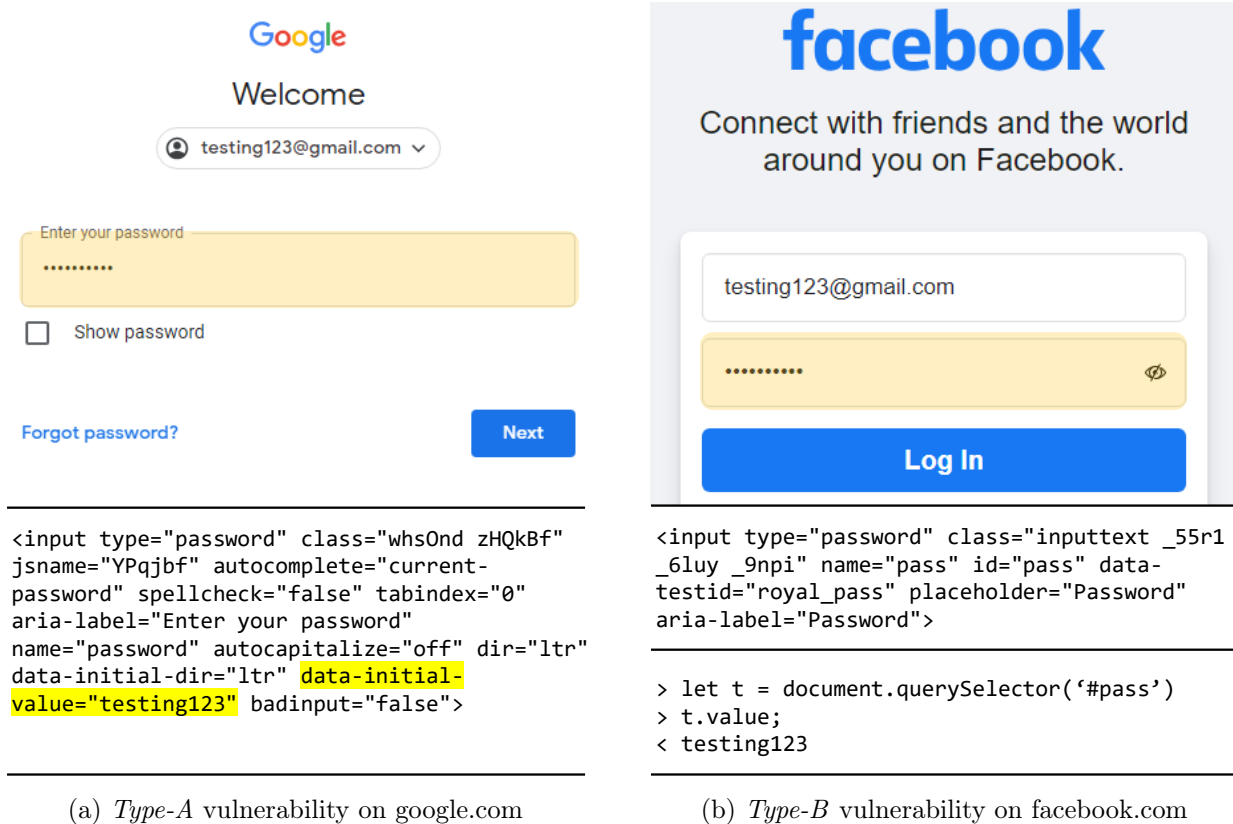
(b) *Type-B* vulnerability on facebook.com

Figure 4: Different types of vulnerabilities found in the wild. (a) The vulnerability allows a malicious extension attached to the DOM tree to extract login credentials. (b) The password is visible in the `outerHTML` of the element and can be extracted directly from the source code.

suffering from these vulnerabilities can be found in Figure 4.

In *Type-A* vulnerability, the sensitive values are visible in plain text in the source code of the webpage. For example, in the case of password fields, the password values are present in the `outerHTML` of the element, as shown in Figure 4a. This is particularly concerning as any entity with access to the source code, including the extensions, can extract the password values. This security lapse also has implications on possible defenses that we discuss in Section 7. We also note that, to the best of our knowledge, we are the first ones to highlight the *Type-A* vulnerability.

In *Type-B* vulnerability, the extension can gain access to the input elements via the DOM API, and extract the input element's value using the `.value` function. The values of sensitive input fields such as password fields are often masked to prevent shoulder surfing [15] or screenshot attacks [6]. This masking is done either via HTML input types (e.g. `type=password`) or through JavaScript-based obfuscation methods (e.g. private variables using WeakMaps[4]). Notably, accessing the value using `.value` function bypasses the HTML-based masking, as shown in Figure 4b.

4.2 Proof-Of-Concept Extension

Prior work has exploited the lack of security boundary between the extension and the rest of the DOM tree [3, 6, 18, 24, 25]. They either used static or dynamic code injection to extract sensitive

---

[4] `https://modernweb.com/private-variables-in-javascript-with-es6-weakmaps/`

data. Extensions with static code are impractical as the malicious code can be detected via code analysis [5, 8, 24, 26, 28]. On the other hand, dynamic code injection attacks are not feasible after the introduction of MV3. Thus, to build a practical extension and exploit the observed vulnerabilities, we need - a) to access the input elements without using dynamic code injection; b) to overcome any obfuscation on the values of input fields; and c) to submit the extension to chrome web store and clear the review process.

We note here that a malicious extension can also manipulate the elements and modify the content to perform other attacks such as screenshot attack [6] and phishing attacks [2, 19]. However, in this work, we focus on the security of text input fields and the sensitive information that can be extracted from them. We devise three attacks based on the observed vulnerabilities: *source extraction attack*, where we copy the sensitive values of input fields from the element's `outerHTML`; *value extraction attack*, where we select the target input field and read the sensitive values; and *element substitution attack* where we bypass JavaScript based obfuscation to extract sensitive values.

Our primary objective is to build a practical extension to pass the webstore review process and extract sensitive information. To achieve this, we propose a hybrid attack that leverages techniques from static and dynamic code injections. Specifically, we design our extension to include a benign code template that identifies an element with a given *CSS selector*. We dynamically retrieve the *CSS selector* string from a server which allows us to control the input fields at runtime. This technique is similar to that used by Khandelwal et al. [14]. We do not require additional permission to communicate with the server and retrieve the *CSS selector*. We instead use the background page to fetch the string and pass it through messages to the content script, as shown in Figure 1. Finally, we identify the type of attack to be used by analyzing the element's properties. A skeleton code of the extension is shown in Figure 5. Note that this is different than the code injection attacks as they obtain the code and execute them using `eval` statements. In our case, we only get the CSS selector string from the server, and we avoid using any eval statement. Additionally, our extension only requires minimal permission to run on the given page. Next, we describe the different attacks in detail.

```
1  ...
2  fetch('server_url') // Retrieve CSS selector
3    .then(response => response.text())
4    .then(data => {
5      var els = document.querySelectorAll(data); // Select the target element
6      for (let el of els) {
7        var outerHTML = el.outerHTML
8        var typeA = checkForTypeA(outerHTML);  // Determine if Type-A
9        if (typeA){
10            el.addEventListener(text, sourceExtractionScript)
11        }
12        else{
13            el.addEventListener(text, valueExtractionScript)
14  }}});
15  ...
```

Figure 5: Skeleton code showing how the extension extracts the content of sensitive fields by determining the type of vulnerability that can be exploited.

Source Extraction Attack: In this attack, we exploit the vulnerability where the sensitive values are present in the source code of the HTML. In this attack, we wait for the user to enter their credentials and capture the webpage's HTML when the user clicks the Login/Sign-in button, and

the page is redirected. At this instant, we copy the page's HTML source code, using the `outerHTML` property of the `window.document.body` object. Once captured, we can use regex to capture only the `password` type input field and send it back to the server. This is a novel attack leveraging the security lapse found in many websites, including `google.com`, and can potentially impact billions of users, as discussed in Section 5.1.

Value Extraction Attack: This attack involves identifying and selecting the target input field. Using JavaScript, this can be done in several ways, e.g. xpaths, CSS-selectors, etc. For our attack, we use the CSS selectors to identify the elements. Specifically, we use `document.querySelectorAll` method takes a CSS selector string and returns all the elements that match the selector. For example, for password fields, we use the selector `input['type=password']` and gain access to all the input elements with the password tag. Note that such selectors can be generated for other sensitive fields too. As discussed in Section 4.1, websites can use HTML-based or JavaScript-based obfuscation. With these obfuscation methods, the text entered is not visible on the screen. However, for HTML-based obfuscation, we can access the values using `.value` once we can access the elements.

Element Substitution Attack: JavaScript-based obfuscation prevents access to the information using the `.value` function. For example, for SSN fields, `chase.com` uses JavaScript-based obfuscation, which prevents access to the values. To extract values from these elements, we propose *Element Substitution* attack where the extension can leverage the lack of security boundary between the extension and the DOM tree and replace the protected input element with a simple password field. This new input field still appears secure, however, the values can be extracted using the `.value` function. We show a skeleton code corresponding to the element substitution attack in Figure 6.

```
1  ...
2  fetch('server_url')
3    .then(response => response.json())
4    .then(data => {
5      var old_element = document.querySelector(data.selector);
6      var new_element = document.createElement(data.tag);
7      new_element.setAttribute('type', data.type);
8      new_element.name = old_element.name;
9      ...  // Add other attributes
10     old_element.parentNode.replaceChild(new_element, old_element);
11   });
12 ...
```

Figure 6: Skeleton code showing the Element Substitution Attack, where an obfuscated input element is swapped for a basic text-input field.

4.3   Uploading to Web Store

Finally, we submit the extension to the web store to evaluate the web store's review process. The extension passed the review process on Google Chrome web store. To hide the extension's malicious aspects, we disguised it as a GPT-based assistant offering ChatGPT-like functions on websites. The extension asked for permission to run on all websites, which is reasonable as most extensions that offer assisting features ask for this permission.

Webstores' failure to identify the malicious extension highlights the need for more robust verification systems for browser extensions. The existing security checks may not be sufficiently comprehensive or effective in identifying potential threats. This is particularly concerning given the

potential for extensions to access sensitive user data, including passwords and other input field data, as shown in this work.

Ethical Considerations. We maintained ethical integrity throughout the process by adhering to the established guidelines from prior works [19]. Specifically, we ensure we do not collect sensitive information from manual testers during the review process. Our extension was engineered to interact with our servers, identify the type of HTMLElement we were targeting (in this case, input elements), monitor the values on those elements, and ultimately transmit the recorded values back to our server. To protect the privacy of the manual tester while not revealing the extension's malicious nature, we deactivated our data-receiving server, retaining only our element-targeting server online. Consequently, our extension would request the target element, acquire the CSS selector, and then attempt to send the recorded data to a non-existent server. This procedure ensured that the primary operation of the extension remained consistent with our original design. We uploaded the extension to the web store once, ensuring we did not waste testers' time during the manual review process. Additionally, once approved, we immediately removed the extension from the web store. We always kept extension in "unpublished" mode so the users could not find and install the extension.

4.4 Root causes for attack success

Next, we analyze the root causes for the success of our attacks. The success of our attack stems from a combination of factors, including the improper application of security principles, a trade-off between usability and security, bad practices by websites, and flaws in the online review process of extensions. In this section, we discuss these root causes and their implications for the security of sensitive input fields.

Improper Application of Security Principles. The success of our attack can be largely attributed to the improper application of fundamental security principles within the current design of web browsers and extensions. One of the key issues lies in the coarse-grained permission model at the HTML level. Once an extension is allowed to run on a page, it has unrestricted access to all elements. This unrestricted access is in direct violation of the Principle of Least Privilege, a fundamental security principle that advocates for limiting the permissions granted to a process to only those that are necessary for its function.

This unrestricted access also undermines the principle of Complete Mediation, which requires that every access to a resource be checked for appropriate permissions. In the current model, once an extension has been granted access to a page, subsequent accesses to elements on the page are not checked, allowing the extension to interact with all elements on the page freely.

These violations of fundamental security principles create an environment where sensitive data is vulnerable to unauthorized access and manipulation, highlighting the need for a more secure design that adheres to these principles.

Trade-off Between Usability and Security. In security systems, there is often a trade-off between usability and security. This trade-off is clearly evident in the current security landscape of web browsers and extensions. Websites often rely on browsers to provide the necessary security protections, placing trust in the browser's ability to safeguard sensitive data. However, this trust can lead to vulnerabilities if the browser's protections are insufficient or can be circumvented by malicious extensions.

An example of this trade-off can be seen in password managers. While they aim to improve convenience by storing and automatically filling passwords, they require access to password fields, which can compromise security measures. This creates a challenge in balancing between implementing strict security measures and ensuring the smooth operation of password managers.

Interestingly, we observed that many websites attempt to obfuscate Social Security Numbers (SSNs) but not passwords. This suggests a recognition of the need for protection of sensitive data, but an inconsistent application of it. The decision to obfuscate SSNs but not passwords may be driven by a desire to balance usability and security, but it also highlights the complexities and potential pitfalls of this trade-off.

Websites' Bad Practices. Our case studies revealed a range of security practices across different websites, with some leaving sensitive input fields unprotected or implementing only minimal protections. The reasons for these practices are not always clear, but they contribute to the overall vulnerability of these input fields. Even obfuscation, while better than leaving sensitive data in plain sight, is insufficient to fully secure these fields.

Flaws in Online Review Process of Extensions. Lastly, the online review process for extensions, particularly those with dynamically loaded selectors, has significant flaws. These flaws can allow malicious extensions to pass through the review process undetected, providing them with a platform to launch attacks. It's important to note that this is different from code injection, as the malicious code is part of the extension itself, not injected into the webpage.

These factors combine to create a landscape where sensitive input fields are vulnerable to attack. In Section 5 and Section 6, we measure the *practicality* and *prevalence* of this vulnerability on real websites by conducting large-scale measurements (Section 5) and case studies (Section 6).

5   Measurements

To evaluate the *practicality* of the vulnerability of sensitive text input fields, we analyze the login pages for vulnerabilities from the top 10K websites from Tranco[5]. We take password fields as a representative for sensitive input fields as we can automatically detect password fields, allowing us to scale the analysis. We analyze other sensitive input fields through case studies in Section 6. To measure the prevalence of potentially malicious extensions, we first examine the extensions' permission to understand their *potential ability* to access password fields. We then perform static and dynamic analysis to identify extensions accessing password fields using the vulnerabilities discussed in Section 3.

5.1   Websites' Vulnerabilities

We conduct a comprehensive measurement of the security vulnerabilities associated with password fields. Our infrastructure consists of a custom-built web crawler to navigate popular websites' login pages and inspect the HTML and JavaScript (JS) elements associated with password fields. The crawler is equipped with capabilities to handle different types of login forms, including both static and dynamic forms. It is also designed to detect and categorize the two types of vulnerabilities: Type-A, where password values are visible in plain text in the HTML source code, and Type-B, where password values are obscured but can be accessed via JavaScript. We ran the crawler from a controlled environment to ensure consistency in the measurements.

Methodology: We perform the measurement using a Chromium browser controlled via the Selenium library in Python. We also install the extension that performs the attack (Section 4) to extract the passwords. The overview of the measurement pipeline is shown in Figure 7.

We use the top-10K domains from the Tranco list generated on Feb. 2nd, 2023. We employ a two-tiered approach to identify and analyze the login pages of these domains. First, we attempt to locate the login button on the homepage of each domain by analyzing the text of all clickable

---

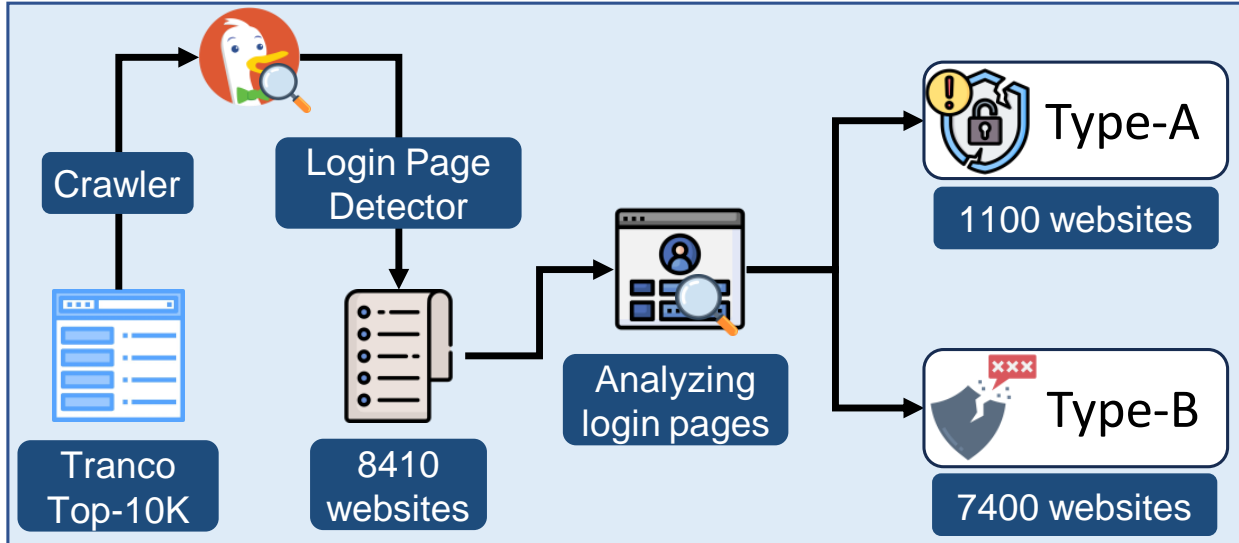[5]List available here: `https://tranco-list.eu/list/4K4GX/1000000`

Figure 7: Our website vulnerability measurement pipeline uses a custom crawler to identify login pages of websites and detect the type of vulnerabilities present.

elements on the page and searching for keywords associated with the login function. In case of failure, we perform a search on DuckDuckGo using the query `<domain name> log[-?]in`. We then select the top five pages from the search results as potential login page candidates and analyze each candidate page to determine if it is a login page. In particular, we treat a page as a login page if there is a *username/email* field or a *password* field.

After finding the login page, we automatically enter a unique username and password and attempt to extract them using the extension. We note that a login page can exist without password fields (e.g., `linkedin.com`). Specifically, there can be login pages where the password fields appear only after the email/username is entered. To capture the password field in such cases, we press ENTER after inserting the username and check if the password field is present. This allows us to capture login pages where the password fields are initially hidden.

Results: In our study, we identified login pages for 8,410 websites out of the top 10,000 domains. Among these, we found password fields present on 7,140 websites. The remaining 1,270 pages contained username or email fields but no password fields. We manually inspected a subset of these web pages to investigate the causes. We discovered that certain websites do not display password fields unless the provided email address corresponds to a registered account. For instance, on `quip.com`, the page redirects to third-party sign-in portals. Additionally, we came across cases where non-login web pages contained an email field, leading to false positives in our login page detection. It's important to clarify that these false positives do not affect the results presented in this work, as these pages lacked the password fields.

Notably, we could extract password data from all the websites that presented the password fields. Further analysis revealed that 1,100 websites exhibited *Type-A* vulnerability; the password values were displayed in plain text within the HTML DOM. Figure 4 shows snapshots of these vulnerabilities, depicting password values in plain text in the HTML. The underlying issue is that the *value* attribute of the *input* element is set to update at each keystroke. In most implementations of password fields, this value attribute is omitted or kept empty.

Alarmingly, we find that the *Type-A* vulnerability was present on several popular websites, including but not limited to `gmail.com` and `cloudflare.com`. Gmail has over 1.8B monthly active

15

users, whereas Cloudflare, a cybersecurity company, has over 4M active users. The results indicate that this security vulnerability can potentially impact billions of users. It is noteworthy that cloudflare.com is one of the popular cybersecurity firms used by services like OpenAI, Google, etc., for cybersecurity needs. The existence of such a basic security oversight exists on popular websites is concerning, as even websites with substantial resources are not immune to security lapses. This puts the user at significant risk as any JS code can access the HTML, send it to a private server, and extract passwords without running a JS code. We further discuss the implication of *Type-A* vulnerability on possible defenses in Section 7.
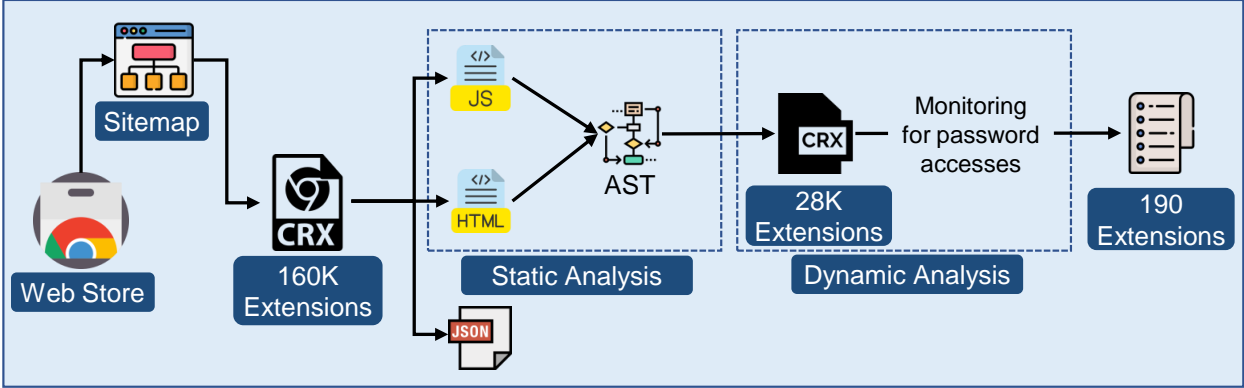

Figure 8: Our extension analysis pipeline uses a mix of static analysis, filtering out extensions that select input fields, and dynamic analysis to check if the password field's content is stored.

5.2 Extensions Vulnerability

5.2.1 Potential Ability To Exploit Vulnerability

We analyze the extensions on the Chrome store to identify how many extensions can potentially exploit the vulnerabilities discussed in Section 4. We first download the available extensions by scraping the Chrome web store. The downloaded extensions include the manifest.json file, HTML files, JavaScript files, and image files associated with the extension. We then analyze the manifest files and look for extensions that request the `scripting` permission, or that request the content scripts to be run on `all_urls`. Depending on the use case, an extension can request content scripts that have access to the HTML DOM tree to be run on specific web pages or all web pages. `Scripting` permission allows the extension to inject content script.

Analyzing the manifest files, we find that 12.5% (17.3K) extensions have the necessary permissions to extract sensitive information on all web pages. This includes popular extensions such as *AdBlockPlus* and *Honey* with more than 10M users. We also find that 33.6% (46.4K) extension request content scripts to be run on at least one website, whereas 55.2% (76.3K) extensions do not request `scripting` or content script permissions.

5.2.2 Potential Prevalence

Prior research has demonstrated the existence of malicious extensions in the webstore [6, 16, 26]. In this study, we focus on the potential for extensions to select and store password fields in a variable and aim to measure how many extensions access the password fields.
Methodology: Figure 8 shows the extension analysis pipeline. Our objective is to identify extensions that select any password fields. Identifying access to input fields is a challenging problem as

```
1  "CallExpression[callee.type='MemberExpression'][callee.property.name=/
     querySelector/] > Literal": function(node){
2    if (node.value.toString().toLowerCase().includes("input")) {
3      context.report({
4        node: node,
5        message: "Found input string"
6      });
7    }
8  }
```

Figure 9: ESLint code snippet to detect the use of `querySelector` to select elements based on CSS selectors having 'input' as a substring.

JavaScript provides numerous methods to select a `HTMLInputElement`. Thus, filtering extensions using all possible selection methods is infeasible [17]. Therefore, we perform static analysis and create custom ESLint rules to filter extensions that include a function containing the `querySelector` or `getElement` keywords and include `input` as its function parameter, as shown in Figure 9. This selects extensions that are selecting input fields. This filtered list contains some extensions that do not perform any input field selection, but their function call matches our filtering criteria. Conversely, our filters may fail to capture extensions that use alternative forms of element selection.

Next, we perform dynamic analysis to identify extensions that select and store password-type input fields. Following prior works [6, 28], we instrument the extension to determine whether the passwords are stored in a variable within the extension code. Specifically, we insert a `console.log` below the variable declaration to print its value.

Upon instrumenting the filtered set of extensions, we recompress them into CRX files and then use Selenium to load them automatically into a Google Chrome instance. We then visit the login pages of Facebook and Citi Bank, input a unique string in the username and password field, and verify whether these strings appear in the console window. If they do, we flag the extension as selecting and storing password-type input fields in variables.

Results: Our scraping of the web store resulted in 160K extensions. After applying our static analysis filters, we retained 28K extensions. Dynamic analysis of these 28K extensions flagged 190 extensions storing password values in a variable. Of these 190 extensions, 12 had more than 10K downloads, and three had more than 100K downloads. While some flagged extensions functioned as password managers, many were random extensions that selected and stored password fields. For example, Remote Torrent Adder's extension, with over 40K downloads, accesses input fields and stores them in a variable.

5.3 Takeaways

Systemic Issue. Our measurement studies on the top 10K websites show that we could extract passwords from all the login pages with passwords. The widespread presence of these vulnerabilities indicates a systemic issue in the design and implementation of password fields.

Need for Stringent Security Measures. The presence of *Type-A* vulnerabilities, where passwords are visible in plain sight in the HTML source code, is concerning. This severe vulnerability bypasses any browser protections, even the ones presented in this paper (Section 7), leaving sensitive data exposed and easily accessible to anyone viewing the source code. This highlights the need for more awareness of security measures.

Role of Password Managers. The widespread use of password managers may partially explain

the prevalence of *Type-B* vulnerabilities, where password values are obscured but can be accessed via JavaScript. These tools enhance the user experience by automating the process of entering passwords, storing the encrypted passwords, and later auto-filling these fields when required [27]. This functionality reduces the cognitive load on users and encourages the use of complex, unique passwords for each site, thereby enhancing overall security [10].

However, for password managers to function effectively, they require access to password fields via JavaScript. This necessity creates an inherent security vulnerability. While the password fields may appear obscured to users, any JavaScript code running on the page, including potentially malicious scripts, can access these fields and read their values. This interaction between password managers and *Type-B* vulnerability presents a trade-off between usability and security. While password managers improve usability and promote better password practices, their operation necessitates JavaScript access to password fields that inherently creates a security risk. We discuss potential solutions to maintain usability while providing security in Section 7.

5.4 Limitations

Website Measurements. We note two main limitations associated with our methodology. First, we may have missed dynamically loaded pages that rely on user interaction to reveal login forms. Second, our method for identifying login pages relied on the presence of certain HTML input fields (such as email and password fields). However, some websites may employ unconventional methods or unique identifiers for their login procedures, making it difficult to identify all login pages correctly.

Extension Analysis: In our extension analysis, we use a combination of static and dynamic components to identify problematic extensions. During the static analysis phase, we only include extensions that select input fields with methods like `querySelector`, `querySelectorAll`, `getElementBy`, and `getElementsBy`. However, our static analysis can't include every extension that selects input fields due to the numerous ways to select elements.

In dynamic analysis, we modify the extensions to automatically insert a log statement into the variable holding a selected element. This lets us track extensions that store input data in a variable but misses extensions that process the input data directly without storage. Some malicious extensions activate after a time delay, which our method also misses. Finally, our dynamic analysis does not detect extensions that add an event listener to input fields instead of simply reading the values.

6 Case Studies

Text input fields like Social Security Numbers (SSNs) and Credit Card details may expose private data. Misuse of this data could lead to identity theft or financial fraud. We study the possible leakage of such data via input fields. We note that carrying out large-scale investigations to find such leaks is difficult, as this data is typically input after the user logs in. Therefore, we opt for a case study approach. In particular, we focus on popular websites where such information is commonly entered, aiming to understand how websites protect input fields. In this section, we present our findings from these case studies, providing insights into the current state of security for SSN and credit card input fields on the web.

6.1 SSN Leakage

Social Security Numbers (SSNs) are unique identifiers assigned to individuals in the United States for identity verification, tracking earnings, and credit checks. Unauthorized access or exposure of

the SSNs of an individual can result in fraudulent loans or criminal charges being falsely attributed to the victim. According to the U.S. Department of Justice, millions of Americans fall victim to identity theft yearly, resulting in significant financial loss [9]. As more financial institutions require users to enter their SSNs for various transactions (e.g., tax returns or credit checks), the security of these input fields becomes increasingly critical. We examine websites that implement protections for SSNs and those that do not.

Websites Protecting SSNs. Several financial institutions, including Chase, Wells Fargo Bank, and Bank of America, have implemented measures to protect SSN input fields. These measures involve using JavaScript-based obfuscation to protect the values entered into these fields. When a user enters their SSN on these websites, the input field displays obfuscated characters (such as asterisks or dots) instead of the actual SSN. This means that the extension can only see the obfuscated characters, providing a layer of protection against malicious extensions.

However, our research indicates that even these protections are not foolproof. Our element substitution attack, discussed in Section 4, can bypass the security measures and access the information entered by substituting the HTML element containing the obfuscated SSN. This demonstrates that while obfuscation can provide a level of protection, it is not sufficient to secure sensitive input fields fully.

Websites Not Protecting SSNs. On the other hand, websites such as IRS.gov, Capital One, and USENIX security do not implement any protections. In these cases, SSNs are visible in plain text. This presents a significant security risk, as any malicious extension could potentially access and steal this sensitive information. Furthermore, the lack of protection on these websites is particularly concerning, given their nature. For instance, IRS.gov is the official website of the U.S. Internal Revenue Service, and Capital One is a major financial institution.

These case studies highlight the variability in security measures implemented to protect SSN information across different websites. While some websites have protections in place, others exhibit significant vulnerabilities that could be exploited to gain unauthorized access to sensitive data.

6.2 Credit Card Information Leakage

Credit card information is another type of sensitive data frequently entered into website input fields. Unauthorized access to this information can lead to financial fraud, such as unauthorized transactions. Furthermore, credit card information is often used without additional authorization (such as two-factor authentication), making it a particularly attractive target for attackers. Here, we conduct case studies on a selection of websites to understand how they handle the security of credit card information in input fields.

Websites Protecting Credit Card Information. One example of a website that implements measures to protect credit card input fields is `silvercar.com`. They add the attribute `type=password` to their credit card input fields. This obfuscates the information entered, similar to password fields, providing a layer of protection against unauthorized access. However, this protection is relatively weak, and our attack (Section 4) can bypass this and access the information entered by calling the `.value` method on the input element.

Websites Not Protecting Credit Card Information. On the other hand, major online marketplaces such as Google and Amazon do not implement any protections for credit card input fields. In these cases, credit card details, including the Security Code and zip code, are visible in plain text on the webpage. This presents a significant security risk, as any malicious extension could potentially access and steal this sensitive information. The lack of protection on these websites is particularly concerning, given their scale and the volume of transactions they handle daily.

6.3 Takeaways

Our case studies on the handling of sensitive information in input fields across various websites have yielded several key insights. We observed different security practices, from unprotected input fields to fields with minimal protections or JavaScript-based obfuscation.

Unprotected Input Fields. In the absence of any protective measures, as seen on websites like IRS.gov, Capital One, USENIX, Google, and Amazon, sensitive data such as SSNs and credit card information are immediately accessible to all extensions running on the page. This presents a significant security risk, as private data is left vulnerable.

Minimal Protection. Some websites, like `silvercar.com`, attempt to protect sensitive input fields by using the `type=password` attribute. While this visually obfuscates the entered data, it only offers minimal protection. Value extraction attack described in Section 4.2 can bypass this protection.

JavaScript Based Obfuscation. A more sophisticated approach is seen on websites like Chase, Wells Fargo Bank, and Bank of America, where JavaScript-based obfuscation is used to protect the input elements. However, even this measure is not foolproof. Our *element substitution* attack, which involves substituting the protected HTML element containing the sensitive information, can effectively circumvent this protection.

The success of our attack across all these scenarios highlights the fundamental issue: the extension's unrestricted access to all HTML elements on a page and the ability for them to manipulate these elements. This unrestricted access allows a malicious extension to bypass even the most sophisticated protections currently in place, highlighting the urgent need for more robust and comprehensive security measures. Notably, third-party JavaScripts loaded on the webpage are also attached to the DOM tree and have the same privileges as a browser extension. Therefore, the findings discussed here also apply to them, further exacerbating the security vulnerabilities of sensitive input fields.

7 Remedies

As we have shown in this work, the lack of security boundary between the extension and the webpage can allow a malicious extension to extract sensitive user information entered in input fields. In this section, we propose a two-fold approach to address these vulnerabilities. The *Bolt-on* solution serves as an add-on package that web developers can use to secure sensitive input fields. In contrast, the *Built-in* solution proposes a more fundamental browser-level solution that involves instrumenting the browser to alert users when sensitive fields are accessed.

7.1 Bolt On

In the *Bolt-on* solution, we provide a JavaScript package that the developers can use to protect sensitive input fields. Specifically, we introduce a new `HTMLInputElement` type, `SecureInput`[6] that leverages WeakMaps to store the sensitive information as private data. Unlike previous solutions [6, 16], our solution is ready to use and does not necessitate a major revamp of the current browser extension architecture. Developers can simply import the secure-input library and designate any input they wish to secure as follows:

```
1   <input is="secure-input" type="password">
```

---

[6] `https://osf.io/nbdfj/?view_only=c496010851314a3299c9e816804aac52`

The `SecureInput` class inherits all the properties associated with the base HTMLInputElement or the input tag. We store the real value of the input field in the WeakMap while presenting a masked value to the value attribute of the `HTMLInputElement`. This approach effectively prevents *Type-B* leaks (Section 4.1). We note that the website retains full access to the input field and its methods as the `SecureInput` class is employed by the website.

Furthermore, using WeakMaps ensures that most known attacks fail to access sensitive information. For example, Eriksson et al.'s [6] methodology of changing the input type to text would only display the masked value. Additionally, the `secure-input` library employs a mutation observer that alerts the user in case the secure-input field is switched with any other input field, by an attacker or otherwise, protecting against element substitution attacks (Section 4).

We provide out-of-the-box support for authentication using `SecureInput` field. Specifically, we add a new `submit()` function that locates the username and communicates with the developer-specified server, returning the authentication status. To use `SecureInput` in other applications, such as password strength meter, the developers can customize the package to add the desired functionality. We note here that some features of password managers are not supported by `SecureInput`. For example, automatically reading and saving passwords from the login forms is not supported. We accept this as a limitation. However, this limitation can be addressed by asking the user to re-type the password in the password managers' input fields. The autofill feature of password managers remains unaffected.

7.2 Built In

The solution proposed above, `SecureInput`, acts as an add-on solution to prevent unrestricted access of sensitive input fields. However, this does not address the root cause of the vulnerability, i.e. lack of a fine-grained permission model for sensitive fields. Prior works [6, 16] have proposed modifying the browser architecture to address this vulnerability. Specifically, Liu et al. [16] suggest augmenting the existing privilege set to restrict Chrome extensions from accessing input fields of high 'sensitivity.' Concurrently, Eriksson et al. [6] propose including a new `captureVisibleTab` permission to prevent extensions from obtaining user passwords.

In this work, we take a different route and propose instrumenting Chrome to alert users whenever any JavaScript function accesses any password fields. We note here that instrumenting chrome is a big undertaking, and hence is out of scope for this work. Here, we present a proof-of-concept solution showcasing the necessary steps required to achieve the desired functionality.

Our key insight here is that to programmatically access the sensitive values, the adversary first needs to select the element. We aim to intercept this access flow, and alert users when the access originates from JavaScript or browser extension. To develop a proof-of-concept solution, we focus on the flow where `document.querySelector` is used. We notify users both when the sensitive input field is selected and when its value is read. To accomplish this, we update the compiler file responsible for managing the document object to log a message whenever a querySelector selects the sensitive element. Finally, we update the core compiler for HTMLInputElement to log when the value of the sensitive field is read. Figure 10 shows the the logggin functionality on `facebook.com`.

It's important to note that this represents only a proof-of-concept for a possible system that could be used to notify users. Updating Chromium to notify users with a more user-friendly design exceeds the scope of this paper. Furthermore, we only show the logging by intercepting one selection method `document.querySelector`, but the methodology can be extended to other selection methods as well.

Impact on Type-A Vulnerability. Recall that in *Type-A* vulnerability, the password values are present in plain text in the source code of the element. Both the defenses presented in this section

do not address this vulnerability as, if the password values are available in plain text, the adversary does not need to select the input element and read its value; they can simply copy the entire HTML and analyze the HTML on a remote server to extract the sensitive information.

7.3 Tradeoffs

The bolt-on solution comprises a JavaScript library that keeps the password variable private, preventing JavaScript from accessing password values. This solution, which can be added to existing systems without major alterations to the underlying architecture, is fairly straightforward. It offers protection against numerous attacks that exploit JavaScript's access to password fields. However, the solution has its shortcomings. It doesn't guard against attacks that tamper with the entire HTML element or place a decoy textbox over the actual one, thereby leaving potential opportunities for exploitation. For example, the keylogging attack will bypass this defense and leak user data.

On the other hand, the built-in solution proposes a change at the browser's OS level and alerts users whenever an extension or JavaScript tries to access a sensitive field. This solution provides a more all-encompassing defense, tackling various potential attacks. Since it operates at the OS level, it offers a more cohesive and constant layer of protection. This solution can also integrate the functionality of password managers by giving trusted extensions exclusive access to password fields, thereby preserving usability while ensuring security. However, there are several challenges associated with this solution. Implementing the solution is more intricate and resource-consuming as an OS-level change than incorporating a bolt-on solution. It might also necessitate collaboration from various stakeholders, including browser developers, which could make the implementation process more complex.

8   Related Works

Attacks involving Extension: Prior studies [6, 17, 18, 25] have detailed various techniques by which malicious extensions could leak sensitive information. Wang et al. [25] conducted an empirical study on over 2400 Firefox extensions, revealing numerous vulnerabilities threatening web sessions. Obimbo et al. [18] showed how external attackers could take advantage of the elevated privileges given to extensions by exploiting code vulnerabilities present in those extensions. Similarly, Carlini et al. [3] highlighted that due to the reality that not all extensions are developed by developers who are security experts, code vulnerabilities exist that can be exploited by malicious actors to expose user data potentially. Eriksson et al. [6] systematically identified multiple attack entry points that a malicious browser extension could use to steal user information. For instance, they created a malicious extension that would alter the text type of password fields to 'text' and then capture a screenshot. Varshney et al. [24] demonstrated a variety of attacks a malicious extension could conduct to steal user information. These attacks included keylogging, credential sniffing, phishing, and inbox spying. However, all these attacks were not submitted as extensions to the webstore, making them theoretical rather than practically feasible, especially considering the current static and dynamic analysis defenses against such malicious extensions.

Bauer et al. [2] developed an iframe-based attack to stealthily steal user credentials by leveraging the autofill functionality of password managers. On the same lines, Perrorra et al. [19] crafted an extension that performed an iframe-based phishing attack where their extensions would fetch dynamic codes from a server and execute them. They managed to bypass and publish their extension to the Chrome web store. However, as we discuss later in Section 3.2, this attack is no longer viable due to Chrome's ban on dynamic remote code execution.

In this work, we propose a new hybrid attack that leverages both static and dynamic attack techniques to extract sensitive user information. We further submit the extension to the Chrome web store and find that it bypasses the security checks, showing the practicality of the attack (see Section 4).

Detection of Malicious Extensions: Several previous studies have devised tools and frameworks for the detection of malicious extensions. Research conducted in [26, 28] combined both static and dynamic analyses to identify and flag extensions. While Zhao et al. [28] focused on the detection of information leaks via extensions, Wang et al. [26] emphasized tracking DOM changes to identify malicious extensions. Varshney et al. [24] also introduced a static analysis framework for detecting malicious code within an extension. DeKoven et al. [5] identified malicious extensions by flagging users who behave suspiciously on websites, subsequently scanning all loaded extensions for specific threat indicators. Shahriar et al. [21] utilized a Hidden Markov Model to analyze and detect vulnerable and malicious extensions. Toreini et al. [22] created DOMtegrity to monitor and flag malicious DOM changes like 'document.write' or swapping child nodes. Previous research proposed dynamic analysis frameworks that analyze runtime code bases of extensions and match them to set heuristics to flag them as malicious [4, 13].

Our work complements this line of study, offering a security analysis of vulnerabilities affecting browsers and generic solutions to address these vulnerabilities.

Browser Architecture: Prior research has investigated how modifications to browsers' underlying structure can enhance user privacy and security. Louw et al. [17] suggested incorporating a new runtime monitoring framework to observe an extension's access to sensitive APIs, such as adding an event listener to secure fields like passwords. Guha et al. [8], and Liu et al. [16] recommended adding new permissions to access specific DOM elements. Bauer et al. [2] explored how extensions could bypass the existing Chrome permission structure to execute a range of attacks. Some, though not all, of the changes advocated by these studies have been implemented in the latest iteration of the Chrome extension platform (Manifest V3), such as banning `eval` statements.

Unlike the aforementioned research that calls for a complete overhaul of the extension permission systems, we propose both a JS-based bolt-on solution and a Chromium patch that alerts users of sensitive field access (see Section 7). The chromium patch, distinct from the solution proposed by [2, 3, 8, 16, 17], does not necessitate any redesign of existing frameworks. Furthermore, the patch addresses threats from any malicious JS script and is not restricted solely to extensions.

9   Conclusion

In this paper, we have presented a comprehensive analysis of the vulnerabilities associated with text input fields in web browsers, focusing on the exposure of sensitive information such as passwords, Social Security Numbers, and credit card details. We find that the lack of security boundary between the browser extension and the webpage results in novel vulnerabilities. Our case studies and large-scale measurements highlight the extent of these vulnerabilities, with alarming findings such as the exposure of passwords in plain text on over a 1000 websites, including popular ones like Google and Cloudflare. We also demonstrate the feasibility of a malicious extension bypassing existing protections and accessing sensitive data, underscoring the urgent need for more robust security measures.

To address these vulnerabilities, we propose two solutions: a JavaScript library that makes password variables private and a modified version of Chrome that notifies users when a password field is being accessed. While these solutions address some of the issues, they also highlight the need for a more comprehensive approach to securing sensitive input fields.

References

[1] Ron Amadeo. Adware vendors buy chrome extensions to send ad- and malware-filled updates. https://arstechnica.com/information-technology/2014/01/malware-vendors-buy-chrome-extensions-to-send-adware-filled-updates/, 2014.

[2] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan Tian. Analyzing the dangers posed by chrome extensions. In *2014 IEEE Conference on Communications and Network Security*, pages 184–192, 2014.

[3] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. pages 97–111, August 2012.

[4] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1687–1700, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Louis F. DeKoven, Stefan Savage, Geoffrey M. Voelker, and Nektarios Leontiadis. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. August 2017.

[6] Benjamin Eriksson, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Hardening the security analysis of browser extensions. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1694–1703, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Google. Overview of manifest v3. https://developer.chrome.com/docs/extensions/mv3/intro/mv3-overview/, 2023.

[8] Arjun Guha, Matt Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. *2011 IEEE Symposium on Security and Privacy*, pages 115–130, 2011.

[9] Erika Harrell. Victims of identity theft, 2018. 2018.

[10] Cormac Herley and Paul C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10:28–36, 2012.

[11] Honey. I am one of the developers of a popular chrome extension and we've been approached by malware companies that have tried to buy us. ama! https://www.reddit.com/r/IAmA/comments/1vjj51/i_am_one_of_the_developers_of_a_popular_chrome, 2014.

[12] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 579–593, Washington, D.C., August 2015. USENIX Association.

[13] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. pages 641–654, August 2014.

[14] Rishabh Khandelwal, Asmit Nayak, Hamza Harkous, and Kassem Fawaz. Cookieenforcer: Automated cookie notice analysis and enforcement. *ArXiv*, abs/2204.04221, 2022.

[15] Arash Habibi Lashkari, Samaneh Farmand, Dr Omar Bin Zakaria, and Dr Rosli Saleh. Shoulder surfing attack in graphical password authentication. *arXiv preprint arXiv:0912.0951*, 2009.

[16] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *Network and Distributed System Security Symposium*, 2012.

[17] Mike Ter Louw, Jin Soon Lim, and Venkat Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4:179–195, 2008.

[18] Charlie Obimbo, Yong Zhou, and Randy Nguyen. Analysis of vulnerabilities of web browser extensions. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 116–119, 2018.

[19] Raffaello Perrotta and Feng Hao. Botnet in the browser: Understanding threats caused by malicious browser extensions. *IEEE Security & Privacy*, 16(4):66–81, 2018.

[20] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, page 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, and Thibaud Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 47:66–84, 11 2014.

[22] Ehsan Toreini, Maryam Mehrnezhad, Siamak Fayyaz Shahandashti, and Feng Hao. Domtegrity: ensuring web page integrity against malicious browser extensions. *International Journal of Information Security*, 18:801 – 814, 2019.

[23] Steven Ursell and Thaier Hayajneh. Desktop browser extension security and privacy issues. *Lecture Notes in Networks and Systems*, 2019.

[24] Gaurav Varshney, Manoj Misra, and Pradeep Atrey. Detecting spying and fraud browser extensions: Short paper. pages 45–52, 10 2017.

[25] Jiangang Wang, Xiaohong Li, Xuhui Liu, Xinshu Dong, Junjie Wang, Zhenkai Liang, and Zhiyong Feng. An empirical study of dangerous behaviors in firefox extensions. pages 188–203, 09 2012.

[26] Yao Wang, Wandong Cai, Pin Lyu, and Wei Shao. A combined static and dynamic analysis approach to detect malicious browser extensions. *Security and Communication Networks*, 2018:1–16, 05 2018.

[27] Rui Zhao, Chuan Yue, and Kun Sun. Vulnerability and risk analysis of two commercial browser and cloud based password managers. *Science*, 2:183–197, 2013.

[28] Rui Zhao, Chuan Yue, and Qing Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1384–1394, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

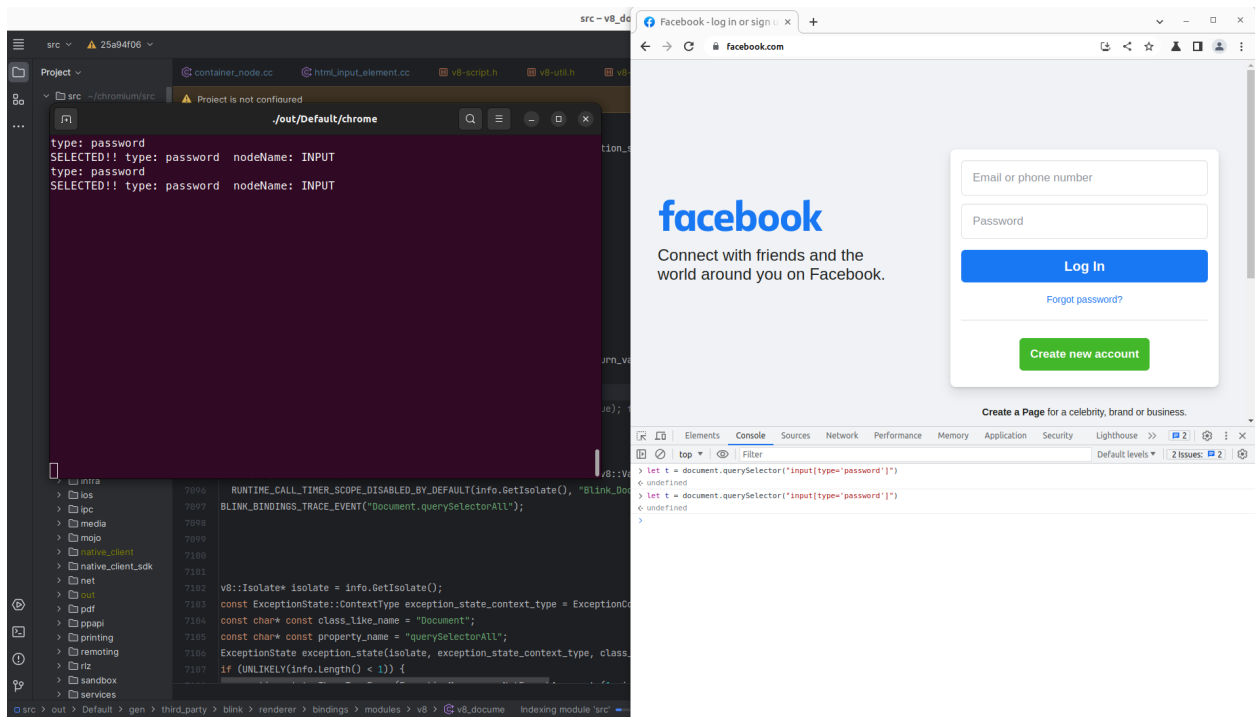# A Appendix

A.1 Chrome Instrumentation

Figure 10: The output of the logging code as a part of our chrome instrumentation to intercept sensitive element selection and notify users.