Helping or Hindering? How Browser Extensions Undermine Security

Shubham Agarwal
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
shubham.agarwal@cispa.de

ABSTRACT

Browser extensions enhance the functionality of native Web applications on the client side. They provide a rich end-user experience by utilizing feature-rich JavaScript APIs, otherwise inaccessible for native applications. However, prior studies suggest that extensions may degrade the client-side security to execute their operations, such as by altering the DOM, executing untrusted scripts in the applications' context, and performing other security-critical operations for the user.

In this study, we instead focus on extensions that tamper with the security headers between the client-server exchange, thereby undermining the security guarantees that these headers provide to the application. To this end, we present our automated analysis framework to detect such extensions by leveraging static and dynamic analysis techniques. We statically identify extensions with the permission to modify headers and then instrument the dangerous APIs to investigate their runtime behavior with respect to modifying headers in-flight.

We then use our framework to analyze the three snapshots of the Chrome extension store from Jun 2020, Feb 2021, and Jan 2022. In doing so, we detect 1,129 distinct extensions that interfere with security-related request/response headers and discuss the associated security implications. The impact of our findings is aggravated by the extensions, with millions of installations dropping critical security headers like Content-Security-Policy or X-Frame-Options.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Client-side Security, HTTP Security Headers, Browser Extensions

ACM Reference Format:
Shubham Agarwal. 2022. Helping or Hindering? How Browser Extensions Undermine Security. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560685

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560685

1 INTRODUCTION

Browser extensions are an integral part of the modern-day Web that provides additional client-side features to their users, such as improving the appearance of Web sites and integrating with third-party services. They are more potent than native Web applications, owing to the feature-rich JavaScript APIs that allow them to intercept and control client-server exchanges, read and modify DOM content, and much more. Thus, they often mediate private information and perform sensitive operations for the end-users.

Web applications are regularly the targets of different attacks, from Cross-Site Scripting through framing-based attacks to TLS downgrading. Researchers and practitioners have developed various mitigations for these attacks, typically delivered through HTTP headers from the server and subsequently enforced by the client. For instance, the server may define the Content-Security-Policy header to control script inclusion and framing by third-party pages. However, given the capabilities mentioned above, extensions may tamper with such headers, effectively disabling well-configured security mechanisms and, thus, degrading the applications' security.

Prior studies have reported the abuse of extensions to perform nefarious actions such as history-sniffing, data theft, or ad-injection [4, 21, 46, 48, 55, 56]. Bauer et al. [7] showed potential threats posed by extensions that can modify HTTP headers. Kapravelos et al. [33] analyzed malicious behavior and reported 24/48,332 Chrome extensions that modified security-related headers. In 2015, Hausknecht et al. [27] showed that extensions might need to alter CSP to allow intended functionality and proposed an endorsement mechanism to enable these. However, any security header alteration may lead to potentially dangerous consequences due to disabled security mechanisms. A preliminary study by Agarwal and Stock [3] also indicated that extensions alter security headers. However, their work suffered from "critical errors" [2] that led to incorrect findings and their paper to be withdrawn. Hence, our community currently lacks any systematic study to assess the *inadvertent* modification or removal of security-critical headers and their impact on the Web.

To close this research gap and study such behavior at scale, we present an automated pipeline to detect extensions that alter security headers using a hybrid analysis technique. The framework statically analyzes the codebase and instrument extensions to observe their behavior at runtime and detect potentially harmful extensions. We also review the risks associated with this behavior and its impact on the Web's security. Our findings show that at least 1,129 extensions in Chrome inadvertently interfere with security headers, affecting millions of users on high-profile sites.

To summarize, the key contributions of this study are:

- We present our automated framework to detect extensions that alter security headers, ultimately undermining the security boundaries desired by the Web application.
- With our framework, we conduct a large-scale study with three snapshots of the Chrome Web Store and show that 1,129 extensions alter security-related headers, thus influencing or even undermining security mechanisms.
- We conduct an additional analysis between the three snapshots of the Chrome extensions and discuss the temporal evolution of the extension ecosystem.
- We outline the implications of modifying different security headers and discuss specific cases from our analysis.
- We show the versatility of our framework by applying it to Firefox and making our pipeline publicly available [1] to encourage extension stores to include it in their vetting.

2 BACKGROUND AND RELATED WORK

In this section, we describe the relevant security headers we consider in our work. Specifically, we introduce all headers that we observed in our analysis, which have a clear security relation and are not deprecated (such as the `X-XSS-Protection` header). Subsequently, we outline the general architecture of browser extensions and survey-related work in the space of extension security.

2.1 Security-related Response Headers

HTTP security headers are a subset of standard HTTP headers that mediate security-specific information between the server and the client. The Web server sends necessary security policies within the response headers to the browser when a user visits any Web site. Upon receiving the response, the browser enforces these security policies for the given site on the client. Altering these headers in-flight may potentially disable the desired defense mechanism, e.g., framing control, and expose an application to the associated types of attacks, e.g., clickjacking. In the following, we briefly outline the server-sent response headers that influence security mechanisms.

Content-Security Policy (CSP): Although initially introduced to mitigate cross-site scripting (XSS), this header has undergone several revisions over the years to control framing and enforce secure communication channels on the client [30]. The server controls how the browser should handle different content for a given Web site through various CSP directives, such as scripts, images, or forms. These directives contain the allowed origins and instruct the browser to load resources only from those origins. This primary use case was often the topic of modifications in CSP; e.g., to eliminate the dreaded `'unsafe-inline'` keyword, CSP's Level 2 introduced nonces and hashes to selectively allow the inline scripts from the developer. In Level 3, CSP added the `'strict-dynamic'` keyword to enable scripts trusted through nonces or hashes to programmatically add additional script resources while simultaneously disabling a host-based allowlist. Notably, CSP is designed in a backward-compatible fashion, i.e., combining nonces with `'unsafe-inline'` will enable modern browsers to rely on nonces (i.e., they ignore `'unsafe-inline'`), while legacy browsers still execute the inline scripts even without supporting nonces.

Orthogonally, `frame-ancestors` limits the URLs which can render the current Web page inside an *iframe*, thus mitigating click-jacking attacks. CSP also contains directives to enforce TLS over an insecure connection and prevent any *man-in-the-middle* attack. For instance, the `block-all-mixed-content` directive instructs the browser to block all mixed content on the Web page while the `upgrade-insecure-requests` directive forces the browser to upgrade all HTTP resources and links to HTTPS. Entirely removing CSP headers obviously disables all available use cases, yet manipulations within the directives may also expose the client to otherwise mitigated XSS attempts.

HTTP Strict-Transport-Security (HSTS): The server defines the HSTS header to ensure a secure communication channel between the client and the server and prevent any man-in-the-middle attack such as protocol downgrade attacks and cookie hijacking [28]. When the browser receives this header, it automatically upgrades any HTTP request to HTTPS for the given host. The browser can cache this setting for a given period, as specified by `max-age` directive (at least one year as per the recommended standards), while the `includeSubDomains` directive instructs the browser to load all the subdomains for the given host over HTTPS as well. Thus, altering this header can undermine HSTS enforcement, opening the client up to protocol downgrading attacks and potential cookie leakage.

X-Frame-Options (XFO): The X-Frame-Options header defends against click-jacking attacks on the client side. It allows the server to restrict their Web site to be framed inside another Web site (e.g., `iframe`) over same or different origin [29]. For instance, one can entirely disable framing for their site using `DENY` attribute or only allow framing from the *same-origin* pages using `SAMEORIGIN` attribute. While CSP's `frame-ancestors` is the desired way to prevent framing-based attacks, websites still deliver the XFO header more frequently than the better CSP-based alternative [49]. Furthermore, in the presence of `frame-ancestors`, the X-Frame-Options header is ignored by modern browsers; however, Web sites often use both, often with conflicting security guarantees [10]. Nevertheless, if only XFO is present, this is used to control framing, such that interfering with this may enable click-jacking attacks.

X-Content-Type-Options: This HTTP header aims to thwart the MIME-type sniffing vulnerability on the client side by instructing the browser not to determine the MIME type of the content and obey the values specified by the `Content-Type` header. Whenever a user accesses any uploaded file for which either the server sends no `Content-Type` header or with inappropriate values, the browser "sniffs" the resource to determine its content type. This can lead to dangerous situations, e.g., when the client detects the uploaded file as HTML. By setting the `nosniff` attribute for this header, the server instructs the browser not to sniff the MIME type of any resource. Conversely, dropping the header leaves the application vulnerable to content sniffing.

Cross-Origin Resource Sharing (CORS): CORS is a mechanism by which the server can instruct the browser to make content available to JavaScript even though the content comes from a different origin than the requesting page [42]. However, the server can use `Access-Control-Allow-Origin` with either an allowed origin or a `*` wildcard to control this interaction. This can be combined with `Access-Control-Allow-Credentials`, which enables access to *credentialed* requests, such as those with cookies

attached to them. Note that combining a wildcard with credentials does not work, as CORS-enabled browsers ignore the credentials flag in this case. Moreover, `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers` enables complex requests with non-standard methods or request headers. Notably, CORS is server-sent but enforced by the browser; hence altering them may allow cross-origin read access to otherwise unprivileged JavaScript.

Set-Cookie: Orthogonal to headers that explicitly control security mechanisms, cookies also carry security attributes. These range from `HttpOnly`, to disallow access to a cookie from JavaScript and mitigate XSS attacks, through `Secure` (to only send cookies on HTTPS connections) to the `SameSite` attribute. `SameSite` cookies intend to protect against Cross-Site Request Forgery (CSRF) attacks. Here, the browser only sends cookies on same-site requests. If any of these properties are altered, cookies may be prone to stealing through an XSS from JavaScript or accidental leakage in unencrypted requests. Moreover, if cookies are set to `SameSite=none`, they will be sent along with cross-site requests, potentially leading to exploitable CSRF flaws.

Additional Security Mechanisms: The Web servers can further rely on several recent additions, such as `Referrer-Policy` (to control when the referrer header is sent [57]), `Permissions-Policy` (to selectively disable browser features like geo-location [58]), as well as `Cross-Origin-Resource-Policy`, `Cross-Origin-Opener-Policy`, and `Cross-Origin-Embedder-Policy` to mitigate Spectre-like attacks and XS-Leaks [39–41].

2.2 Security-related Request Headers

Like server-sent response headers, *request* headers sent by the client can also have a security impact. In the following, we provide an overview of relevant headers and explain how they can combat specific threats on the server.

Referer and Origin: Browsers have two ways to communicate to a remote server from where a request was made. They can either use the `Referer` header, which sends the entire URL of the referring document to the server. In modern browsers, though, only the origin of a referring resource is sent when a request is made cross-origin [43]. This can be used on the server to make a security decision; notably, while an attacker can strip the `Referer` header, it cannot be modified freely. Hence, a server could allow a particular request only if the `Referer` header is present and has a specific value. Since this potentially leaks sensitive information (e.g., session identifiers in the URL), it has a privacy-friendly counterpart in the `Origin` header. This header, sent automatically by browsers on CORS-governed requests and cross-origin form posts, only contains the origin of the referring document. Relying on the `Origin` header can be a meaningful defense against CSRF, as modern browsers only omit it on *same-origin* form posts. Hence, if either of the values is modified, this potentially undermines the server-side defense.

Fetch Metadata: Cross-site leaks [53] are a relatively new class of attacks. Similar to Cross-Site Request Forgery (CSRF) attacks, these occur because the server delivers content to the browser without knowing a) who made the request (the user or a script), b) the origin of the request, and c) as what the resource was included. To rectify this, the W3C has proposed the so-called *Fetch Metadata Request Headers* [59]. The core idea is for the browser to indicate to the server the information it needs to make security decisions. This is done by sending multiple HTTP request headers, which the server can take into account. If implemented in the browser, `Sec-Fetch-Site` indicates whether a request is coming from the same origin (`same-origin`), the same site (but different origin, `same-site`), from an entirely different site (`cross-site`), or is triggered by a user typing a URL into the address bar (`none`). This header enables the server to, e.g., mitigate CSRF by simply ignoring requests that are not (at least) `same-site`. `Sec-Fetch-Dest` is used to inform the server about the type of resource requested. This header is helpful for the server to decide whether a certain document should be returned, e.g., when an image is loaded as an iframe to measure load time for an XS-Leaks attack [53]. Furthermore, `Sec-Fetch-Mode` communicates which mode is used for a particular request, such as `navigate` to inform the server that a request is the result of top-level navigation. The last relevant header is `Sec-Fetch-User`; it informs the server whether a user action caused a request or not, which can be used by the server to detect a potential CSRF attack.

upgrade-insecure-requests: With this header, the client can inform the server about using its preference to use an encrypted connection. While, fortunately, HTTPS is increasingly used [25] with full browser support, this header nevertheless informs the server that the client can handle secure connections. Hence, while arguably, it might not serve much of a purpose in the modern era, it should not be dropped to give a false impression that the client does not fully support HTTPS.

2.3 Extension Architecture

Browser extensions are lightweight add-ons that enhance the user experience on the client side by utilizing the extension-specific APIs exposed by the corresponding browser.

Seminal works by Barth et al. [6] and Carlini et al. [11], respectively, argued for fine-grained privilege separation based on the principle of least privilege and component isolation between the extensions and native applications within the browser ecosystem. Modern browsers enforce these principles as follows: each extension has a *manifest* where the developer defines the metadata of the application, such as the API permissions required to perform privileged operations, the target hosts on which it should operate, and other operational configurations. It also consists of different script components that contain its core logic. If appropriately configured, their cross-origin capabilities are not governed by the Same-Origin Policy (SOP). Instead, the extensions can make authenticated fetch requests to allowed hosts and execute JavaScript within Web applications. The background script, often called the extension core, runs in a single isolated process as a separate component. It has access to the highly privileged, feature-rich Browser Extension APIs to perform a multitude of functionalities. These functionalities include accessing browser history, periodic script execution in the background, managing the list of extensions installed, and launching or uninstalling them as desired. It can also passively intercept or actively drop, issue, or modify requests for any given Web site. Hence, extensions are more potent than regular Web applications, given the wide range of operations they can perform on the client, such as making cross-origin requests. Each content script runs as a

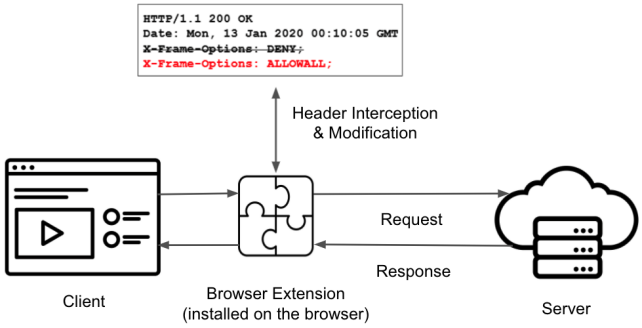

Figure 1: Extension with header-modification capabilities

separate instance and within the context of any given Web site. It can directly interact with the page, modify its contents and further execute its script from within the context of the webpage unless this is forbidden through per-site security mechanisms such as CSP.

To access any extension API, browser API, or the Web site's content they intend to use in their components, the extension developers must declare its corresponding permissions in the manifest. Upon an extension's installation, the user must grant all the required permission to enable the extension. For example, Figure 1 illustrates an extension that holds the privilege to intercept the client-server exchange in its background. In this example, the extension changes the value of the X-Frame-Options header from `DENY` to `ALLOWALL` to override the server's security decision to disallow framing. While it is necessary to declare permission to use privileged APIs, it must also declare the host permissions on which the extension core should operate. For instance, if an extension intends to intercept and modify request headers on `example.com`, it needs to a) declare `webRequest` and `webRequestBlocking` as API permissions and b) `example.com` as a host permission.

Although it is recommended for the developers to request minimal permissions for their target functionality, prior studies have shown that only a few extensions adhere to this policy [7, 32, 33]. Moreover, once the user grants the required privileges to these extensions at install time, the user does not control how and when the extension utilizes the granted privileges to carry out their functions in the background and within what context. Thus, an extension with elevated privileges can be harmful to the security of the application and the sensitive data processed on the client side.

2.4 Related Work

Malicious Browser Extensions: Given the higher privileges of extensions, they are a prominent vector for attackers to abuse them for nefarious purposes. Thomas et al. [55] and Xing et al. [60] individually conducted a longitudinal study and showed that over 300 extensions inject unwanted ads from illegitimate sources into the webpage for monetary purposes. Around the same time Kapravelos et al. [33] conducted a large-scale study with over 48K Chrome extensions to identify *malicious* extensions using dynamic analysis and *HoneyPages*. They found 24 such extensions that altered security-related headers but did not elaborate on the impact of these modifications. While they explicitly labeled 11 of them as malicious for injecting security-related headers, we instead measure this behavior from a different vantage point where these actions do not necessarily indicate malice and may be required for benign reasons for the extensions. We further open-source our pipeline and contact extension developers to better understand the underlying issue pertaining to this behavior. Bauer et al. [7] statically examined the privileges of top 1,000 Chrome extensions and described threat scenarios where they could be abused as a potential attack vector for a multitude of attacks on the client. They highlighted that extensions could alter security headers to bypass security restrictions but did not measure if this occurred in the wild.

Perrotta and Hao [48] and DeKoven et al. [21] further affirmed in their respective works that extensions are often abused to install malware on the client and may serve as an integral component of a botnet framework over the Internet, owing to the capabilities that they enjoy on the client side. Jagpal et al. [32] proposed a fine-grained screening process that combines both static and dynamic analysis techniques to distinguish rogue extensions from benign ones. Another line of work by the authors in [4, 46, 56] reported that extensions also observe users' activity on the Web and may steal sensitive information and send it to third-party domains to create unique user profiles. Chen and Kapravelos [12] used taint-propagation techniques on 180K Chrome extensions and identified 2.13% among those which leaked any private data. Recently, Pantelaios et al. [47] also analyzed the updates received by these extensions and reported that 143 (0.09%) of them were initially benign but later turned malicious by receiving updates, thus bypassing the initial screening.

Vulnerable Browser Extensions: Contrary to extensions actually being malicious, they may also be the victim of attacks through vulnerabilities in their code. Bandhakavi et al. [5] proposed an automated vetting tool to statically analyze the Firefox extensions' source code to detect potential vulnerabilities that it may pose before submitting them to the store. Somé [52] analyzed extensions from different stores and found 197 among them that could be exploited due to insecure message handling, leading to SOP bypasses, cookie stealing, or history sniffing. Recently, Fass et al. [24] further reported 278 such extensions in the Chrome ecosystem alone.

Header Interference by Extensions: While the studies on malicious extensions show that the adversaries abuse extensions as an attack vector on the Web for various nefarious purposes, they do not cover the seemingly necessary modification of security headers for the proper function of the extension itself. Moreover, design choices and coding mistakes in the extension code may not only threaten the integrity but also inadvertently degrade the security of individual sites for visitors with such extensions. Thus far, there have been two works that attempt to shine a light into this space of potentially necessary yet security-degrading header modifications. In 2015, Hausknecht et al. [27] argued that extensions often need to modify CSPs. In particular, this is necessary to implement proper functionality for the extensions that, e.g., need to inject their code into the page. They moreover proposed a mechanism for extensions to communicate their CSP needs explicitly. Agarwal and Stock [3] conducted a preliminary analysis of Chrome extensions focusing on four security headers. However, they relied on a network-based approach, which required two requests. As noted by the authors [2], this may lead to false positives given random server responses.

Therefore, basing the analysis on a network-based approach is error-prone, and we choose a different approach instead.

Given the unreliable results reported by Agarwal and Stock [3], although we observe that the above studies provide evidence that extensions do modify security headers, they do not provide an in-depth analysis of this behavior. Our work addresses this research gap. Specifically, we focus on a large-scale analysis of (most likely) benign extensions to understand how they *inadvertently* undermine security mechanisms. Moreover, we target *all* relevant security headers in our analysis, i.e., both request and response headers.

3 RESEARCH METHODOLOGY

We now describe our pipeline to detect extensions that modify request or response headers. We use our pipeline to answer the following fundamental questions in line with the discussed threat as follows:

(1) How many extensions hold the privileges to intercept or modify Web requests and responses, respectively?
(2) How many of the above also utilize the requested capabilities to modify HTTP headers at runtime?
(3) How many of them actively inject, drop, or overwrite security-related headers on respective target hosts?
(4) Which security headers are most often altered by these extensions? Do these modifications degrade the client-side security of Web applications in the wild?
(5) Does this trend of security header interception and alteration among extensions change over time?

To answer these questions, we implement an automated framework to detect extensions that interfere with security headers. Figure 2 illustrates the various stages of our framework. The framework begins by parsing the manifest of all downloaded extensions and looks for permissions required to alter the HTTP headers. If no such permission is requested, we discard the respective extension. Then, it parses the background script(s) to locate the APIs used to intercept headers at the source-code level and extract the host permissions from each selected extension. The last static step constitutes instrumentation, where it rewrites the native definition of target APIs in all background scripts and stores it on the disk. This allows us to record the headers before and after the execution of the APIs for extensions within a *single run*. Given this hook, we do not suffer from false positives through server-side randomness or race conditions, as mentioned by Agarwal and Stock [2].

During dynamic analysis, the framework visits target hosts for each instrumented extension and stores the original and modified set of headers captured at runtime. Lastly, it analyzes the modifications made by the extensions and further flags them as either benign or potentially dangerous.

3.1 Extension Analysis

Our framework utilizes a static analysis approach to precisely identify those extensions that can potentially intercept client-server exchanges in the first stage of analysis. An extension is required to hold both *webRequest* and the *webRequestBlocking* permission to intercept and modify request and response headers synchronously at runtime [19]. Thus, we first shortlist all those extensions that request both the above permissions, declared in their *manifest*.

```
window.browser = window.browser || window.chrome;
var oneMoreDomain = "*://*.bar.com/";
browser.webRequest.onBeforeSendHeaders.addListener(
    function(details) {
        //core logic
        return {
            requestHeaders: details.requestHeaders
        };
    }, {
        urls: ["*://*.foo.com", "https://*/*", oneMoreDomain]
    }, ["blocking", "requestHeaders"]
);
```

Listing 1: Host permissions specific to the event listener.

In addition to the permission to use the APIs, an extension must request host permission. Hence, we need to identify the hosts for our analysis that successfully trigger the core logic for any given extension. There are multiple ways by which an extension can achieve this. *(i)* The first obvious way is to list all the target hosts in the *manifest* [13]. In this case, the extension may have specific URLs in their manifest such as http://www.example.org or may contain wildcards such as http://*.example.org that allow them to operate on all subdomains of `example.com`. *(ii)* Majority of extensions are domain-agnostic and operate on all domains and thus, declare *<all_urls>, http://*//* or https://*/** [16]. *(iii)* Many extensions declare *<all_urls>* in their manifest yet operate only on certain domains by specifying hosts within the `webRequest` API definitions to intercept requests/responses, specify additional hosts in the manifest, or perform domain checks in their code execute operations, as shown in Listing 1.

An extension may also modify headers on any active tab of the browser, irrespective of the hosts declared in the manifest, using the *activeTab* privilege. This is when the user allows the extension to operate on any host loaded on the currently active tab [15] at runtime. Once permitted, the extension continues to operate until the page closes or the user navigates to any different origin. Since the use of *activeTab* necessitates user intervention at runtime, we need to rely on extension rewriting to analyze such extensions automatically. Thus, we flag them for further steps, which we discuss in the next section.

Based on the list of extensions that have the proper API permissions, we can now extract URLs/hosts we need to visit for our analysis. Our framework starts with gathering all the wildcards and host-related permissions from the manifest and then extracts the URL literals from the arguments of the *webRequest* APIs within the code. Since our static analyzer is lightweight, it only extracts literal values from the target APIs and does not resolve pointer to any variables. In such cases, the framework falls back to the hosts specified in their manifest of the extension. It further resorts to Tranco Top 100 domains if no hosts are extracted from either of the sources. Extracting target hosts from the code also allows us to detect extensions that contain the definition of any of the target APIs. This enables us to later investigate cases where our dynamic analysis did not yield any findings, yet the extension had the permissions *and* defined the logic to intercept headers.

The framework now processes the extracted hosts to replace wildcards with its valid counterpart and store them for dynamic analysis. For example, `*://*.foo/*`, and `http://*.foo/*/*/*` is processed and resolved to `http://www.sub.foo.com`. This is in

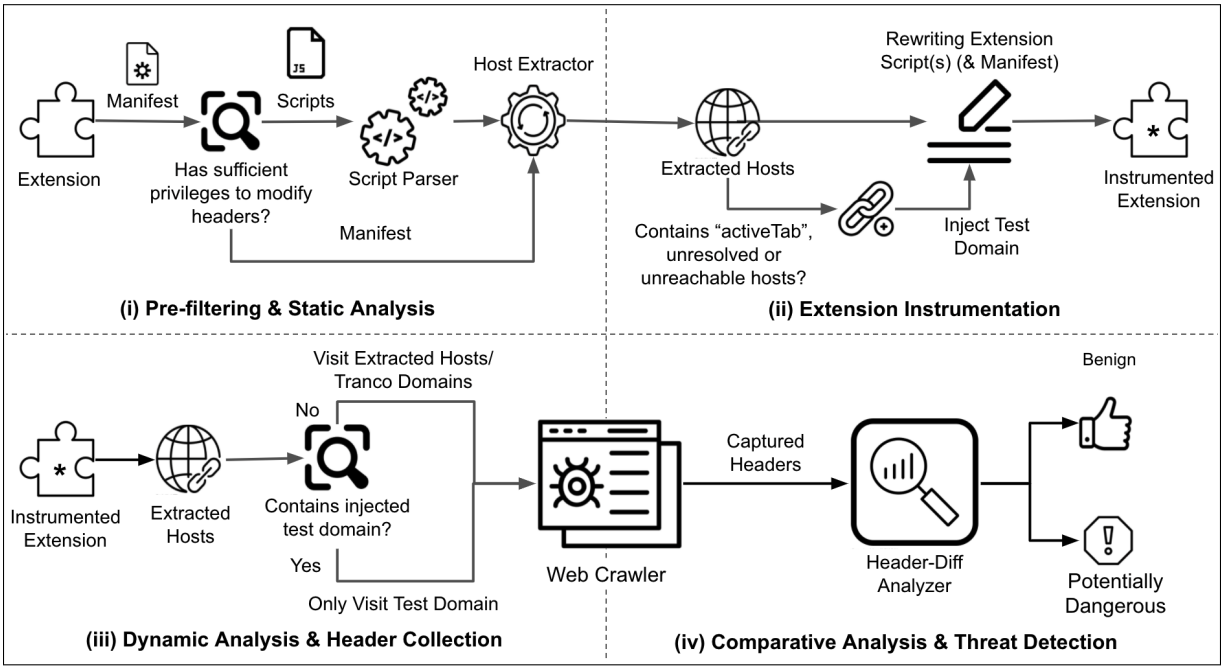

Figure 2: An overview of the automated framework for analysis.

line with wildcards matching patterns for extensions [14]. However, there exist cases where this transformation would not yield accurate results. For example, `https://github.com/*/*/issues/*` and `*://*/*/*.pdf` would not yield a valid URL.

3.2 Extension Instrumentation

Once all the extensions with sufficient privileges to alter HTTP headers and their corresponding hosts are identified, the next step constitutes the instrumentation phase. The framework instrument the extensions by overwriting the native definition of the corresponding `webRequest` APIs within the extension-space to enable an additional header-capturing mechanism while also preserving the intended functionality. This way, it enables us to capture the originally received headers by the API (as an argument) and their modified counterparts returned by the callback method within a single visit to the target host. For example, Listing 2 shows the instrumentation of the `onHeadersReceived.addListener` API, which intercepts the response headers, which now record the headers right before and after the *callback* method executes (as indicated in line 6). In this case, the *callback* method (in line 5) is the original callback method. While we capture the header data only from the relevant APIs, i.e., `onBeforeSendHeaders` and `onHeadersReceived`, we similarly overwrite other `webRequest` APIs to record their invocation (but not log any headers). This helps us to identify the runtime usage for the requested `webRequest` privilege.

Concretely, the instrumentation proceeds as follows: The framework identifies the background script(s) for each extension - from the manifest and background pages, if any. Within these, it first filters out `"use strict"` and non-ASCII characters that may interfere with the execution of our "API hooks" without making any

```
let _onHeadersReceivedListener =
↪  browser.webRequest.onHeadersReceived.addListener;
function _hookResponseListener(callback, filters, opts) {
    function _hooked() {
        let originalHeaders = arguments;
        let newHeaders = callback(arguments);
        recordHeaders(originalHeaders, newHeaders);
        return newHeaders;
    };
    _onHeadersReceivedListener.apply(this, [_hooked.bind(this), filters, opts]);
}
browser.webRequest.onHeadersReceived.addListener = _hookResponseListener;
```

Listing 2: Instrumented API to collect response headers.

changes to the core logic. It then prepends the hooks for the `webRequest` APIs within these scripts and overwrites them on disk. We also modify the CSP defined by the extensions, wherever necessary, to enable the XHRs necessary for our framework to collect header data (for details, please refer to Appendix A).

While running our initial experiments, we noted that our framework did not capture invocations of the hooked event listeners even though the code clearly indicated this was implemented. On further investigation, we observed the following classes of reasons: *(i)* Many target domains are unreachable from certain geographic locations. *(ii)* The target domain (or the client) does not serve all the headers (or even a randomized set of headers) that the extension could potentially alter. *(iii)* Few extensions declare wildcard hosts and modify headers only on them, unresolvable by our framework (e.g. `*://*/*/*.pdf`). Further, as noted in the previous section, an extension may request `activeTab` permissions (which is equivalent to all_urls but requires user intervention).

To cover these special cases, we run a *separate* analysis: we modify any such extension to *(i)* have host permissions for a test

domain we own, and *(ii)* adjust event handlers to react to the said domain. We then schedule the modified extensions to run against our test domain, which serves all headers we consider.

3.3 URL Scheduling & Header Collection

The next stage constitutes dynamic analysis, where we load instrumented extensions separately in the browser and visit each of the corresponding hosts. This enables us to capture the original and modified set of headers at runtime within a *single visit* to the site, meaning our approach is not prone to server-side randomness, which might influence the results. Although the extension stores do not accept extensions with obfuscated code [26, 37], we stress that obfuscation also plays no role in our approach, as we hook native APIs, which must be used to modify HTTP headers. For each instrumented extension, we visit the associated target hosts: any specific host extracted from the code or manifest, or Tranco top 100 domains for extensions that operate on *<all_urls>*. For any extension which falls under the aforementioned special cases (e.g., `activeTab`), we separately schedule the modified extensions to run on our test domain.

We rely on *puppeteer* [18] to load the extensions and launch browser instances. The framework launches the browser for every extension and successfully visits each URL. It stays on the page for six seconds after the page load to record the client-server communication. The hooked target APIs, on invocation, collect the original and modified set of headers and then relays them to our logging server. After every page visit, it clears the cache before visiting the following URL. This is to avoid headers altered by an extension in the current iteration being retrieved from the cache, possibly tainting the analysis for the next extension.

3.4 Detecting Potentially Harmful Extensions

After capturing the extensions' behavior at runtime, we have the original headers and their modified counterpart available for analysis. Now, we analyze the potential security impact of the modification for each extension/URL pair where we observed a difference before and after the invocation of the extension's callback function.

The framework retrieves the original and modified set of headers for an extension, parses them, and compares one set of headers with another to identify the differences among their values. If the server sends multiple instances of the same header within a response, or a single header holds multiple comma-separated values, the framework groups all the distinct values for the given header in a serialized structure and compares them accordingly. For instance, in the case of the `X-Frame-Options` header, when an extension modifies the header to enforce a relaxed version of the server-sent policy, e.g., replace `DENY` with `ALLOWALL`, the framework identifies the change and flag the extension as *modifiedHeader*. If an extension adds any security header for which there exists no ground truth header, the analyzer labels the extension as *injectedHeader*. In contrast, if any ground-truth security header is removed or replaced with an empty string, the extension is marked *strippedHeader*. While we only analyze modifications among headers that we discuss in Section 2, it is possible to extend the analysis to other headers as well since the framework collects all the headers at runtime. Once the framework detects all the modified, injected, and dropped headers, it then reports whether an extension exhibits benign modifications or any suspicious behavior based on the above operations on the target security headers.

	2020	2021	2022
Total downloaded extensions	186,434	174,355	180,361
Actual extensions	166,932	154,415	147,334
Extensions with `webRequest`	17,536	10,620	9,298
Extensions with `webRequest` & `webRequestBlocking`	14,821	7,972	6,720
Extensions for dynamic analysis	14,052	7,660	6,505
- targeting <all_urls>	11,824	5,312	4,659
- targeting specific hosts	2,228	2,348	1,846
Extensions with relevant API calls	3,049	3,147	3,145

Table 1: Overview of chrome extensions from each snapshot

4 CHROME EXTENSION ANALYSIS

We report on a large-scale analysis with three snapshots of all the downloadable Chrome extensions in June 2020, February 2021, and January 2022, respectively. These datasets, as shown in Table 1, consist of 186,434, 174,355 and 180,361 extensions, respectively. Notably, there is an overlap of 109,311 extensions, among 92,441 of which were not updated between the three snapshots. To analyze each extensions' modifications, we employ the Tranco Top 100 domains based on the list of November 1, 2021 (ID: *Y3JG*), as well as those URLs we extract from the manifest or code (as in Section 3).

4.1 Permissions & Source Code Analysis

We download the three snapshots of Chrome extensions by extracting all the extension IDs from the Chrome sitemap [20] and download them from a publicly accessible link.Similarly, we download Firefox extensions by extracting the information from the extension search [45]. Our framework begins by extracting the *crx* packages from the downloaded sets of extensions separately. It filters out those extensions, which are essentially themes or invalid extensions, in its preliminary phase of screening (as shown in Table 1). It then checks for the required privileges, and host permissions declared in the manifest. Doing so, we identified a total of 17,536, 10,620 & 9,298 extensions from the respective snapshots, which requests the `webRequest` privilege to intercept headers. We further narrow it down to extensions with both the required permissions. In the second round of screening, our framework then filters out those pre-selected extensions which do not define any background script and page in their manifest or if the scripts are missing from the directory. At this stage, we obtain a total of 14,052, 7,660 & 6,505 valid extensions, which hold the privilege to synchronously intercept and modify HTTP headers at runtime and consider them for the next stage of analysis.

For the above-selected extensions, the static analyzer component now parses the manifest to identify the background script(s), inject the hooks that overwrite the `webRequest` APIs, extract host URLs from them, and additionally determine the usage of the relevant APIs at source-code level. Here, we identify 3,049, 3,147, and 3,145 extensions for the three snapshots, which provide some evidence of header interception based on the static analysis. We categorize

Operational Stage	2020	2021	2022
Considered extensions	14,052	7,660	6,505
Successfully loaded	14,030	7,643	6,486
Registered any handler	9,842	5,172	5,031
Registered relevant handlers	2,735	2,785	2,713
- *onBeforeSendHeaders*	1,695	1,744	1,672
- *onHeadersReceived*	1,639	1,607	1,598
– *both*	599	566	557
Triggered relevant handlers	2,499	2,577	2,553

Table 2: Overview of dynamic extension analysis

them based on their target hosts, i.e., whether they operate on all hosts or a particular set of the host(s). As listed in Table 1, we observe that a majority of extensions can operate on *all_urls*.

The framework now injects the instrumented hook, as described in Section 3.2, within all the background script(s) for each of these extensions. At this stage, we identify a total of 3,294, 2,723 & 2,266 extensions among the three snapshots, respectively, which contains `content_security_policy` definitions. The framework then adds the remote server's location to the `connect-src` directive to allow XHRs among them and, thus, header storage for further analysis.

4.2 Instrumentation Analysis

Before we describe our findings from the dynamic analysis of the extensions, we first measure and discuss the operation of the instrumented hooks within these extensions at runtime.

Table 2 summarizes the operation of the instrumented hook injected into the extensions during dynamic analysis. For the three sets of extensions, we observe that the hook is successfully injected into ~99% of the extensions at runtime, and the overwritten functionality is executed without any error. For the rest, 26 extensions fail to load at runtime. However, this was not an artifact of our injection but somewhat related to syntax errors in the original extensions across the datasets.

We observe that a total of 9,842, 5,172 & 5,031 extensions from the respective snapshots register at least one event listener for different *webRequest* APIs. However, we only focus on the two relevant APIs that allow request or response header interception. We find that around 2,735, 2,785 & 2,713 extensions from the respective sets register an event listener for either of the two APIs as mentioned above. Notably, this number is well below the one shown in Table 1 for the static analysis. We manually sampled extensions containing invocations in the JavaScript code, but for which our analysis did not trigger. Doing so, we found the discrepancy primarily comes from extensions that only register these event handlers on certain conditions, e.g., configuration by the user.

Finally, the number of extensions for which a relevant handler was triggered is also below those that registered an event handler in the first place. To understand this, we sampled 20 extensions for manual analysis from each set. In all cases, the analysis did not yield results either because our framework failed to extract hosts from the code, since we do not resolve pointer to the variables, or the extension only triggered its functionality on certain runtime conditions. Note also that given the overlap between the datasets, 151 non-triggering extensions are present in all three datasets.

4.3 Results and Overall Impact

In total, 1,129 unique extensions across the three datasets interfere with any of the security headers we analyzed. Of these, 867 tamper with server-sent headers, whereas 315 extensions tamper with client-sent headers; i.e., 53 extensions alter both server- and client-sent security headers. Further, 94 of them alter any of the security headers only on our test domain, highlighting the necessity for the rewriting discussed in Section 3.3. While 10/94 rely on *activeTab*, the target hosts were unreachable or unresolvable for the rest by our pipeline before. In the following, we provide an overview of the most prevalent headers and their respective modifications.

4.3.1 Response Headers. Table 3 shows the overview of our results for response headers, i.e., those sent by the server and enforced by the client. The data is split between extensions that altered headers in 2020, 2021, and 2022 datasets, respectively. Naturally, the extensions overlap across the dataset, i.e. extensions that are available on the store in 2022 and modified headers since 2020 are included in all three snapshots. We find that XFO and CSP are the most affected server-sent security headers for each group. This comes as no particular surprise, given that extensions often attempt to load content in frames or inject their scripts into pages, both of which are actions that are made more complicated (or impossible) through XFO and CSP, respectively. However, we also find that they frequently inject CORS' `Access-Control-Allow-Origin` to allow for cross-origin read access from JavaScript. For the remaining security headers, X-CTO and HSTS are only interfered with by 13 and 15 extensions, respectively, across all groups. For `Referrer-Policy`, we only find alterations by eight extensions. We also note that neither `Permissions-Policy` nor `COOP`, `COEP` and `CORP` is particularly popular targets for tampering, most likely because they are only rarely used in practice. One notable example is *FasterTabs* (60K+ installs) that drops all the headers ending with *"icy"*. In the following, we dive deeper into the more frequently modified headers and the associated security impact.

Content-Security-Policy: Table 4 shows an overview of the top 10 most injected, dropped, and modified directives for different extensions across all the snapshots. Note that if an extension drops an entire CSP header, we count every directive in the original CSP as implicitly dropped. This is because by dropping the entire header, the extension undermines the control of each directive in the original policy. Overall, 205 extensions strip the entire CSP header, and 68 among them span the three datasets. Almost all directives are also injected by at least one extension. Our analysis shows that extensions such as *NoScript* make use of CSP to block content entirely, which is why these and other similar extensions cause several injected directives. On the other hand, we observe that the relevant directives for XSS mitigation (i.e., `script-src` and `default-src` are dropped by 274 and 263 extensions on at least one domain. The most notable case with over 1M installations is the 2020 version of *Fair AdBlocker*, which we observed to be dropping CSP headers in their entirety for Google, eBay, and many others. For 2021, this version has been modified, though, and no longer drops the CSP headers. Another notable example is *HIRETUAL* (30K+ users), which drops CSPs on Twitter, Facebook, Google, and others. This tool, which is meant to help recruiters source contact information, also drops the `X-Frame-Options` header to enable framing. However,

Security Headers	2020 (N = 14,052)				2021 (N = 7,660)				2022 (N = 6,505)				Total
	Inj	Del	Mod	Any	Inj	Del	Mod	Any	Inj	Del	Mod	Any	
content-security-policy	29	189	134	319	23	204	149	339	24	225	157	376	507
content-security-policy-report-only	4	39	24	65	2	47	28	76	1	53	30	83	98
x-frame-options	2	266	13	281	3	300	15	317	4	303	17	321	469
access-control-allow-credentials	7	1	2	10	9	1	4	13	14	1	4	18	22
access-control-allow-headers	14	2	3	16	15	1	5	18	20	1	8	22	29
access-control-allow-methods	24	1	7	26	31	1	10	34	38	1	16	41	46
access-control-allow-origin	57	3	33	66	82	3	38	91	91	4	45	101	121
access-control-expose-headers	3	2	4	6	4	2	4	6	7	2	5	9	11
set-cookie	5	2	4	8	4	2	11	15	8	8	20	28	31
x-content-type-options	1	3	-	5	-	8	1	8	1	9	-	10	13
strict-transport-security	-	7	4	9	-	6	2	6	-	9	2	9	15
referrer-policy	-	1	2	3	-	4	2	6	-	5	2	7	8
permissions-policy	-	1	-	1	-	2	-	2	1	4	1	5	5
cross-origin-opener-policy	-	1	-	1	-	3	1	4	1	4	1	6	6
cross-origin-resource-policy	-	1	-	1	-	2	-	2	2	6	-	8	8
cross-origin-embedder-policy	-	-	-	-	-	1	-	1	1	2	-	3	3

Table 3: Distinct extensions that target different security headers sent by the server-side along with responses.

Directive	Injected	Dropped	Modified	Any
script-src	273	274	263	297
object-src	286	290	208	297
frame-src	264	267	262	306
base-uri	284	290	208	293
worker-src	283	289	206	294
connect-src	258	264	247	293
img-src	258	264	233	285
require-trusted-types-for	266	274	207	274
default-src	258	263	239	282
style-src	258	265	226	278

Table 4: Top 10 most frequently impacted CSP directives

this way, it undermines the security of all targeted sites even if the tool is idling in the background. We further analyze the nature of *modifications* within the `script-src` directive on different sites. We observe that while the majority of them make identical changes to the values across different websites, others modify values according to what is sent by the Web server. For instance, *DWStory - Work Acounts Quick Access* adds "thebodyofchrist.us" into the directive on all websites that send CSP. Over 80 extensions just replace the original directive's value with an empty string. Notably, the extensions that drop CSP altogether amount to at least[1] 3,279,417+ installations, and those only modify CSP (in some form) have a combined count of 3,202,903+ installs.

X-Frame-Options: Although framing-control can be implemented through CSP, the vast majority of sites still rely on `X-Frame-Options` [49]. Chrome only supports two modes: `SAMEORIGIN` & `DENY`. When Chrome observes any other value, it fails insecurely, i.e., setting the value to `ALLOWALL` has no meaning to the browser, ignoring the header altogether. Hence, dropping the header or setting to an unsupported value has no effect, i.e., no protection against framing-based attacks. In our dataset, we found three notable extensions: *Dark Mode for Chrome* drops XFO on all domains (500K+ users). *Eno® from Capital One* (600K+ installs) in turn drops the header on `amazon.com`. Finally, *Notifier for WhatsApp Web* (400K+ installs) drops the XFO header on `whatsapp.com`. Generally speaking, we observe XFO as the header targeted as often as CSP. While this may be necessary for extensions to operate smoothly, none of the descriptions of the highly ranked extensions alludes to the undermining of clickjacking protection. We further analyzed 17 extensions in 2022 that modified XFO on different domains and found that all of them relaxed the original policy by either setting to "allowall" or values essentially discarded by the browser. 3/17 also replaced "deny" with "sameorigin". 6/17 extensions showed this behavior on all domains. This behavior is consistent with the other two datasets and does not change with websites. Such manipulations are equivalent to dropping the headers and thus, disabling any framing-protection altogether. In total, the number of installations of available extensions that drop XFO at least once is 5,372,019+.

CORS: Dangerous misconfigurations for Cross-Origin Resource Sharing require setting both `Access-Control-Allow-Credentials` to true and `Access-Control-Allow-Origin` to a single specific origin, in the presence of an `*` for the allowed origin, CORS fails securely and does not allow credentialed requests. In our data, we found a total of 121 extensions that altered the `ACA-Origin` header. However, only *Flexible Access Helper*, combined the explicit addition of an origin with allowing credentials. Notably, though, CORS can still cause problems if an extension blindly injects an `ACA-Origin` header with `*` into every response. In particular, this enables an attack to access devices that are in the local network of their victim, such as routers [23]. This careless injection is done by numerous extensions, though, with *Video Downloader Plus* being the most popular example with 800K+ installs.

Cookie Security Attributes: We found 31 extensions that modified the security properties of cookies, particularly the `SameSite` flag. One notable example is *HiFrame*, which allows any page to be loaded in an iframe. This extension, which only has around 500 users, also removes framing control directives (both CSP and XFO) and sets all cookies to `Secure; SameSite=none`. Note that Chrome forbids SameSite cookies to lack the Secure flag; hence this is explicitly added by the extension as well. Surprisingly, the documentation states, "This approach led to a simple plug-and-play product that just works for most cases, and does it with minimum

[1] install numbers are only accurate until 1000, then 1000+/2000+/...

Security Headers	2020 (N = 14,052)				2021 (N = 7,660)				2022 (N = 6,505)				Total
	Inj	Del	Mod	Any	Inj	Del	Mod	Any	Inj	Del	Mod	Any	
origin	45	2	11	50	67	1	8	70	83	3	11	92	133
referer	105	18	24	124	146	7	32	158	134	10	27	147	240
sec-fetch-dest	-	1	1	2	3	1	8	9	5	1	12	15	16
sec-fetch-mode	-	1	1	2	3	1	4	6	9	1	6	15	16
sec-fetch-site	-	1	1	2	5	1	11	12	12	1	22	32	34
sec-fetch-user	-	1	-	1	2	1	-	2	1	1	-	2	3
upgrade-insecure-requests	-	2	-	2	1	1	-	2	1	-	-	1	3

Table 5: Distinct Chrome extensions that target different security headers sent by client along with requests.

side effects." — however, actually disabling the protection against CSRF that SameSite cookies entail. Other examples include *DuckDuckGo Privacy Essentials* (5M+ users) that blocks all attempts to set cookies and *Multi Chat - Messenger for Whatsapp* (40K+ users) that sets `SameSite=none`. In total, the reported extensions amount to at least 5,120,425 users.

HSTS: Across our three datasets, we found 15 extensions that mangle with the `Strict-Transport-Security` header. *Multi Chat - Messenger for Whatsapp* (40K+ installs), among many other popular extensions, indiscriminately drops HSTS on all page loads. Overall, however, HSTS does not appear to be a popular target, which is not particularly surprising since enforcing HTTPS connections does not usually hinder the necessities of extensions (such as loading cross-origin resources, rendering pages in *iframes*, or injecting scripts into pages with CSP). However, the user base for currently available extensions is still significant, with 153,199+ installs.

4.3.2 Request Headers. Table 5 shows the overview of our analysis for request headers, split into extensions that we observed to interfere with certainty in 2020, 2021, and 2022 set. In total, 315 extensions interfered with any of the given security headers.

Generally speaking, by far, most changes occur for the `Origin` and `Referer` headers, respectively. This is more pronounced among recent extensions (i.e. after 2020), where 25% more extensions tamper with either of these headers. We did not find significant evidence for interference with Fetch Metadata by extensions from before our 2021 snapshot, which is expected given their recent introduction. Finally, we observe that `upgrade-insecure-requests` is only negatively affected by two extensions (in the 2020/2021 intersection set). Hence, this header remains largely unimpacted by the extensions we tested. In the following, we dive deeper into an impact analysis, showing how the modified, injected, or dropped headers affect the security of each extension's users.

Origin/Referrer: The extension *Pure motion* (40,000+ installations) which labels itself as a "video ad remover" injects fake `Origin` and `Referer` headers into requests towards the supported video platforms such as Dailymotion. Since the `Origin` header may be used to defend against POST-based CSRF, this makes these platforms susceptible to attacks. Our analysis also flagged *Dictionary all over with Synonyms* as dropping the `Referer`. Assuming any server-side check for the `Referer` could also take the security decision based on the origin, this is not detrimental to security. Notably, the *CyDec Platform Anti-Fingerprinting* extension (3K+ installs) seemingly randomized the `Referer`. This could deter the functionality if security checks are conducted based on the `Referer`.

Fetch Metadata: Since the Fetch Metadata specification is quite new, we did not expect many extensions to manipulate this. However, newer extensions tend to be more aggressive towards these headers, as shown by the number of modifications in 2021 & 2022. In total, we found that 15 extensions in the 2022 dataset modified `Sec-Fetch-Dest`. The one with the highest installation number (7000+) was Find website used fonts. This not only set the dest from `iframe` to `main`, thereby faking a top-level load of an actually framed page, but also modified the `Sec-Fetch-Site` header to set it to `none` (the value used for user-invoked top-level navigation by typing in a new URL). This clearly undermines the security goal of Fetch Metadata by pretending to make user-invoked requests.

4.4 Temporal Evolution

Our three datasets allow us to investigate the longitudinal evolution of Chrome extensions over 19 months. Of the 647 extensions which interfered with any security header in 2020, 143 had stopped doing so in 2021. Of these, 115 had been removed from the store, 8 had lowered their privileges, and 25 still had the capabilities but did not use them in our automated tests. Manual checking indicated that these changes related, e.g., to the header interception being controlled through configuration options, as is the case for *Eno® from Capital One*. We find similar trends for extensions that modify headers from the 2021 dataset, where 177 stopped doing so in 2022. Among these, 125 extensions were removed from the store; we do not observe modifications by 38 extensions during our analysis, while 14 of them reduced their permissions. We analyzed these 14 extensions to understand why they no longer need to modify headers. Interestingly, 4 of these now define static rules, and the other 4 define dynamic rules to modify headers in-flight, as per MV3 standards. For instance, *DSM Auto-Paste Chrome Extension* now defines static rules to drop CSP and XFO, as before, instead of using the traditional `webRequest` API. Similarly, *HTML Content Blocker* now defines dynamic rules to modify headers [17]. 6/14 extensions completely stopped modifying headers. *Kurator* used to drop XFO on all domains but stopped doing so after our notifications and reported this as an unintended action in their response.

Of the 406 overlapping extensions that interfered with headers in all three datasets, 55 of them remained entirely unchanged. Among 14,052 extensions in 2020 with the permissions to modify headers, 7,466 were removed from the store altogether, whereas 137 no longer requested permissions in 2021. For 7,660 extensions in 2021, 2,122 were removed from the store in 2022, while 176 reduced their permissions and do not modify headers anymore. To understand if the large number of removals in 2021 were related to the recent

deletion of malicious extensions [34], we randomly sampled 20 of them from the 2021 dataset. Out of those, none were malicious, but the majority (17/20) were wallpaper extensions, which seemingly originated from the same vendor. We, therefore, believe the drastic change to result from such extensions. Further, from a list of known malicious extensions [9], we found that only 30/7,466 removed extensions were listed there, i.e., the vast majority were benign.

Moreover, we found that 56 extensions already were in the 2020 dataset *and* had relevant permissions but only started modifying security headers in 2021. Similarly, 49 extensions with the privileges to modify headers in 2021 only started doing so in the 2022 dataset. We sampled 10 of these 49 extensions and analyzed the usage of header-modifying APIs. An *Online Shopping extension* started injecting `Referer` and `Origin` within request headers, potentially to bypass server-side filtering. Similarly, another such *assistant* now started altering ACA-Origin to access cross-origin resources. Four other extensions (e.g. *IDA* and *RSS Feed Reader*) made logical changes while the rest also added new features that required relaxed security restrictions imposed by CSP, XFO and ACA-Origin, in particular (e.g. *DuckDuckGo !bangs but Faster* and *D&D*).

Notably, only 1,103 extensions with relevant permissions were added between 2020 and 2021, and 160 elevated the required permissions (i.e., they were present before but lacked permissions). Of these 1,263 extensions, 182 actually used their privileges to manipulate headers. Similarly. for the 2022 dataset, we observe 892 new extensions since 2021 that now could modify headers at runtime, whereas 259 already existing extensions upgraded their permissions to modify headers. 188 of these 1,151 extensions in 2022, also modified headers at runtime. We sampled and analyzed ten extensions to understand the need to upgrade their privileges and modify headers. *Video Downloader for Vimeo* set `Origin` to www.vimeo.com to all outgoing requests to this domain, potentially to bypass the newly added filtering mechanism on the server-side. Another such extension, *MarkDown Viewer* dropped CSP on all domains to inject and execute the script and present markdown in a readable format.

Given that we cannot rely on a snapshot from early 2020 to compare the evolution, it is hard to judge whether new extensions make more excessive use of interception capabilities than older ones. However, it is notable that considering only newly added extensions from 2021 and 2022, 14.4% and 16.3% of them, respectively, used their permissions to modify security headers, whereas, in the 2020 dataset, we observed only 647/14,052 (4.6%) extensions which utilize their capabilities in our automated tests.

5 DISCLOSURE & NECESSITY FOR MODIFICATIONS

Here, we discuss our disclosure to have the issues, answers from developers to explain their reasoning for changes, and our manual analysis of sampled extensions.

Disclosure & Developer Responses. From our preliminary analysis in July 2021, we sent notifications to the developers of 327 Chrome and 47 Firefox extensions, which were live on the store, altered security headers, and could be contacted. This was to allow developers to address the underlying problem and to understand the design choices behind such actions. We received 30 responses, categorized as follows: *(i)* The developers acknowledged our report and the potential threat of altering security headers and claimed to either take down the extension or fix the issue. *(ii)* They are unaware of potential security implications of altering certain headers. *(iii)* 24/30 of them also explained the benign usage behind altering security headers such as injecting external images or *iframes* and allowing requests. *(iv)* They have an unidentified bug or legacy code.

We observe that 220/327 chrome extensions still modify headers in the 2022 dataset. 13 extensions for which we received feedback were either removed or do not modify headers anymore. We have sent out notifications to 379 more developers from the 2022 dataset who were not contacted before. While the above responses represent only <10% of extensions, we infer that most extensions indeed exhibit such behavior to enable benign functionalities yet are unaware of its security implications on the application in many cases.

Manual Analysis. To better understand why extension developers modify security headers, we manually sample and investigate 50 extension, spanning across 8 categories, reported by our pipeline. We find that 44 of them modify headers for their desired functionalities based on the description provided by them. Some of these functionalities include injecting scripts and resources to highlight texts, solve captchas and provide other text-based references to third-party domains such as *Wikipedia. uMatrix development build & CSP Safe Browsing* replaces the server-sent CSP header with all-blocking policy to prevent any and all content-based injection attacks. Other extensions often inject CORS-related headers to access cross-origin resources or go after CSP and XFO to bypass framing-based restrictions and allow functionalities such as dashboards to store and access resources, open multiple pages in a single tab, and provide additional tools on social media platforms such as *WhatsApp*.

While a majority of these extensions alter headers for benign reasons in some sense, these alterations are often extreme or could be avoided without any security degradation. For instance, extensions that intend to inject script from a single remote location should not drop the entire CSP header, as seen in many cases, but instead, make bare-minimum changes to the original policy. Similarly, if an extension intends to operate only on specific hosts, such as social media platforms, it should restrict its actions only to those hosts instead of *all_urls*. For example, *Loop11 User Testing* states in its description - *"The Loop11 extension will lay dormant and will only be active during a usability test which you have opted in to"*. However, this is not the case as it alters headers on all the pages when active. Extensions that drop CSP & XFO to allow dashboards and other in-context features may instead use window or extension pop-ups.

6 SECURITY IMPACT OF ALTERATIONS

While more than 1000 extensions modify security-related headers, these changes may not always degrade the client-side security, as with NoScript, AdBlocker, and other privacy-preserving extensions. To better assess the overall impact of these modifications, we perform additional analysis to detect changes for individual headers. While detecting modifications among single-valued headers is seemingly straightforward, we apply extended parsing techniques for multi-valued headers (e.g., HSTS & CSP), as also shown by Roth et al. [50], and discuss popularly targeted headers here [2].

[2]Please refer to Appendix C for details on other security headers.

CORS Headers. We observed 110 extensions that injected `ACA-Origin` in our experiment. While 84/110 only set the header value to wildcard, the rest 26 set it to the requesting domains. Four of them dropped this header, and 53 also modified the existing header value. 44/53 of these replaced the existing values with wildcards, while 6 of them set it to the value of requesting domains. Injecting or modifying the header with value set to wildcard may allow unintended access to credential-less resources to all the domains, while appropriate non-wildcard values, when combined with `ACA-Credentials`, may allow unintended access to credentialed resources as well. 6/26 extensions that inject `ACA-Origin` with value set to the domain also inject the `ACA-Credentials` header, and three others also modify this header, when already present, to allow the credentialed request.

XFO. 446 extensions across all datasets dropped XFO, while 21 modified the header by replacing the value either with its relaxed counterpart (e.g., `deny` to `sameorigin`) or with garbage value. Thus, this clearly indicates lowering security restrictions by extensions. On the contrary, 4 extensions injected XFO with `sameorigin` while one of them replaced `allow-from https://www.icloud.com` to `sameorigin` on apple.com. These changes indicate enhanced security but may also break functionality.

CSP. While *dropping* the entire CSP header shows apparent security degradation, we discuss the impact of other alterations here. We first present the overall impact of injections and then show directive-level modifications using the `default-src` directive, as it serves as a fallback to other content-based directives when not specified.

Injections: 40 extensions injected CSP while 4 also injected CSP-RO. While it may seem that the injections may enhance security, this is not always the case. For instance, the policy injected by 3 extensions on theguardian.com, as in Appendix B, shows no improvement as it specifies trivially bypassable scripting restrictions.

Modifications: 19 extensions explicitly stripped the `default-src` directive from within the CSP header. This may be dangerous if other content-based directives (e.g. `script-src` and `style-src`) are not specified. 8/19 of these also strip script-src, disabling all script-based protections originally desired by the server. 11 other extensions inject this directive. As before, this may not enforce additional security. For example, we observe one extension injecting `default-src` with insecure values (*unsafe-inline* and *unsafe-eval*). Lastly, 45 extensions modified values for the existing `default-src` directive by either adding or removing allow-listed sources. 6/45 extensions also replace none with arbitrary hosts, e.g., one of them replaces `'none'` with `"* data: blob: 'unsafe-inline' 'unsafe-eval'"` on GitHub domains. The exact impact of these modifications depends on other directives and replaced values.

To summarize, most extensions alter headers, leading to relaxed security restrictions. While some also attempt to enforce security by modifying or injecting related headers, improperly configured security policies in these cases, such as with CSP and CORS headers, could still prove to be fatal. We assess the impact of these alterations purely based on the header-based changes during our analysis. The real-world impact of these changes may differ and vary with applications. For instance, a Web site may not consider the `Origin` header while processing the request; thus, altering them will cause no harm. Similarly, any alterations with the HSTS header for domains would depend on their presence on the preload list.

Header	Injected	Dropped	Modified	Total
content-security-policy	13	16	29	45
...-report-only	1	2	4	6
x-frame-options	1	19	4	23
ACA-origin	11	0	7	11
ACA-credentials	2	0	1	2
ACA-headers	6	0	5	6
ACA-methods	5	0	3	5
access-control-expose-headers	2	0	2	2
x-content-type-options	0	0	1	1
origin	4	0	2	5
referer	4	6	18	23
sec-fetch-site	0	0	1	1

Table 6: Firefox extensions that modify security headers

7 DISCUSSION

In this section, we discuss the prevalence of underlying issues beyond the Chrome ecosystem and standards, the limitations of our framework, and possible measures to alleviate the problem.

Extensions Beyond Chrome & Standards. To apply our framework to another ecosystem, our versatile design allows for minimal changes: we merely need to change from Puppeteer to *web-ext* [36] for Firefox extensions. Hence, contrary to a patched browsing engine, our approach is agnostic to newer Chrome versions and allows transfer to other ecosystems. Doing so, we could extend our analysis to all 23,363 Firefox extensions, as of April 2021, and analyze the header-modifying behavior among them. Of the 2,551 extensions that held the privilege to modify headers, 1,253 of them also had the target API definitions at the code level. Overall, we identify 84/2,551 Firefox extensions which tamper with at least one security header on their respective target hosts. Please refer to Table 6 for detailed result for each headers. Unsurprisingly, these extensions also find CSP and XFO among response headers as their biggest roadblock for their operation. Although the total number of flagged extensions is low, they affect a total of 504K+ users by modifying CSP and 41K+ users by altering the XFO headers alone. This shows that the underlying behavior is prevalent across ecosystems and may jeopardize the client-side security of the Web in all browsers. We only considered Chrome & Firefox extensions here since these extensions could be ported to compatible formats for other browsing ecosystems, such as Safari [31], or directly installed from the Chrome Web Store, as in the case of Opera & MS Edge [8, 54].

With the increasing transition to the new MV3 standards in Chrome, extensions may use the new `declarativeNetRequest` API to define static header-modifying rules, which were not included in our study. We separately analyzed those extensions that follow the new specifications to modify headers in 2022 and found that 13,521 extensions adhere to MV3, and 82 of them also specify static rules to alter 14 different security-related headers (as of 23.06.2022). For instance, *Greenhub Free VPN*, a popular VPN extension with 10K+ users, chooses to drop the CSP entirely, XFO, COOP and COEP headers on m.facebook.com and mobile.twitter.com domain and their subdomains by specifying corresponding static rules. Interestingly, *Pincase* with 2K+ users, also perform identical alterations and to the same set of headers. Other such extension is *Flyp Crosslister* with 10K+ users that nullifies CSP on facebook.com.

While intuitively, the analysis may become easier by analyzing the static manifest-defined rules, the documentation states that header-modifying rules could also be added dynamically [17], which again necessitates dynamic analysis. We observe such cases among 118 other extensions that only inject dynamic rules to potentially modify headers along with other operations traditionally allowed by `webRequest` API (e.g., redirection). Further, from the ongoing discussions by Chrome Developers in public space [22], it appears that header *addition & replacement* would still be allowed in MV3. We believe this would again put an onus on the extension developers to correctly add/replace headers such that the changes do not degrade the client-side security, as in the case of CSP, originally desired by the server. Notably, Mozilla will still allow header modifications based on MV2 standards in the future [44]. We also contacted other browser vendors to understand their roadmap for transition to MV3. Brave, Opera, and Safari would still allow MV2 extensions, while MS Edge will allow them until June 2023 [35].

Limitations. We observed certain limitations of our framework during our analysis. We do not inject our hooks into any external script included in the background script. To measure its impact, we parsed the manifest and background pages to locate remote scripts. While 61 extensions, which could intercept headers, included remote code in 2020, this was reduced to only 35 in 2022, indicating that the impact of the remote script is very limited. However, we note that our API instrumentation still captures any alterations made by extensions with obfuscated code as we overwrite the native definition of the APIs before any other script executes. Beyond that, our analysis indicated that some extensions only install event handlers after user consent or when an option is enabled. We do not create honeypages or simulate user interactions, as in other related studies [33, 51] to capture or fulfill any runtime checks; yet, our analysis reveals only those extensions that alter security headers without any user intervention. Thus, we do not capture any alterations for 223 extensions where the target APIs were only registered but not triggered, of which 88 overlap across the three datasets. All the above limitations lead to our results being a lower bound of the gravity of the issue of extensions that undermine the security. Moreover, we stress that we assume extensions to be altering security headers *inadvertently* and for *benign purposes*. Notably, our dynamic analysis is limited when the manipulation is hidden behind a trigger condition. Hence, this limitation also implies that it is unlikely we caught malicious extensions on conditional triggers [33].

Call to Action. Generally speaking, we believe that the header modifications occur in a benign fashion, i.e., the extensions do not drop the headers to undermine security *on purpose*. However, for the user, this does not make any difference; their security is undermined. Given that our results showed that even popular extensions with more than 1M installs had such issues, users should be adequately informed about the potential risk. Notably, the undermined security does not have adverse effects on the developer but may well be detrimental to the user base. For instance, a user that installs an extension that drops CSP on all domains may be at severe risk while using Internet banking or other critical Web sites. Hence, users must be explicitly informed about these risks. This can be achieved by incorporating our pipeline into the extension vetting process and automatically adding a warning to the overview page in the extension store if specific alterations are found. Naturally, our pipeline is prone to malicious developers, e.g., if the extension only starts interfering with headers days after installation. However, assuming that the vast majority of extensions undermine security inadvertently, such an additional vetting may also trigger the developers to find alternatives to the respective roadblocks they tried to bypass by dropping headers. For example, tools that remove XFO to allow framing of arbitrary pages could load those in popup windows instead or only alter headers on specific pages where it is necessary. These recommendations put an onus on the user and require some technical understanding from them. However, this would incentivize the developers to make minimal changes, explore alternatives, and prevent users from "warning fatigue".

We envision Google & Mozilla can integrate our pipeline into the vetting process. The *sequential* runtime of our pipeline scales linearly with the number of URLs to test, but apart from the rewriting step, all other steps can be parallelized. Hence, our analysis could be conducted in just a few seconds on every update on an extension. This process must be repeated every time, as an extension with corresponding privileges could otherwise (inadvertently) start altering security-critical headers in later updates or change their targets. We have seen such cases in our temporal analysis, e.g., *Pure Motion*, which intends to remove ads from videos with more than 40K+ users, updated its host permissions by removing 33 URL patterns and adding another 42 of them.

8 CONCLUSION

In this work, we studied the inadvertent detrimental effect that browser extensions can have on the Web's security by modifying the security-related headers. Such modifications occur when extensions are blocked from operating seamlessly through mechanisms such as CSP, XFO, or Cookie Security attributes. To study this problem space at scale, we present an automated framework that identifies extensions with header modification capabilities and instruments and dynamically analyzes them to detect those that alter any security-related headers. From the three sets of Chrome extensions downloaded over three calendar years, we find 1,129 unique extensions across the three datasets that alter at least one security header sent along with the client-server exchange. Similarly, we extend our analysis to the Firefox ecosystem and report 84 extensions to exhibit similar behavior.

While the number of reported extensions is small compared to the entire ecosystem, our findings show the issues to be more prevalent than, e.g., extensions that turn malicious [47] or purposely alter headers [33]. Considering the most-modified headers, CSP and XFO, extensions interfering with those affected millions of unknowing users. To enable stores to issue such warnings automatically, we make our pipeline publicly accessible and call on different store operators to incorporate it into their vetting process.

ACKNOWLEDGMENTS

We would like to thank our reviewers for their valuable feedback. Special thanks to Ben Stock for insightful discussions throughout this work. We would also like to acknowledge the suggestions by Aurore Fass and Sebastian Roth to help better our work. This work

was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

REFERENCES

[1] 2022. Black Canary Code. https://github.com/shubh401/black_canary.git
[2] Shubham Agarwal and Ben Stock. 2021. Critical errors in our recent MADweb paper. https://swag.cispa.saarland/default/2021/07/19/madweb-headers.html
[3] Shubham Agarwal and Ben Stock. 2021. First, Do No Harm: Studying the manipulation of security headers in browser extensions. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2021*.
[4] Anupama Aggarwal, Bimal Viswanath, Liang Zhang, Saravana Kumar, Ayush Shah, and Ponnurangam Kumaraguru. 2018. I spy with my little eye: Analysis and detection of spying browser extensions. In *IEEE Euro S&P*.
[5] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. 2010. VEX: Vetting Browser Extensions for Security Vulnerabilities.. In *USENIX Security*.
[6] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting browsers from extension vulnerabilities. In *NDSS*.
[7] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan Tian. 2014. Analyzing the dangers posed by Chrome extensions. In *IEEE Conference on Communications and Network Security*.
[8] Opera Blogs. 2021. Using Chrome Extensions in Opera. https://blogs.opera.com/tips-and-tricks/2021/10/using-addons-from-chrome-in-opera/
[9] Mallory Bowes-Brown. 2021. Chrome Malicious Extension Listing. https://github.com/mallorybowes/chrome-mal-ids
[10] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. 2020. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In *USENIX Security*.
[11] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An Evaluation of the Google Chrome Extension Security Architecture. In *USENIX Security*.
[12] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *ACM CCS*.
[13] Chrome Developers. 2012. *Declare permissions*. https://developer.chrome.com/docs/extensions/mv3/declare_permissions/#host-permissions
[14] Chrome Developers. 2017. *Chrome DevTools Protocol*. https://developer.chrome.com/docs/extensions/mv3/match_patterns/
[15] Chrome Developers. 2020. *chrome.activeTab*. https://developer.chrome.com/extensions/activeTab
[16] Chrome Developers. 2020. *Match Patterns*. https://developer.chrome.com/extensions/match_patterns
[17] Chrome Developers. 2020. *Methods*. https://3-72-0-dot-chrome-apps-doc.appspot.com/extensions/declarativeNetRequest#method-updateDynamicRules
[18] Chrome Developers. 2020. *Puppeteer*. https://developers.google.com/web/tools/puppeteer
[19] Chrome Developers. 2020. *webRequest*. https://developer.chrome.com/extensions/webRequest
[20] Chrome Developers. 2022. Sitemap - Chrome Extensions. https://chrome.google.com/webstore/sitemap
[21] Louis F. DeKoven, Stefan Savage, Geoffrey M. Voelker, and Nektarios Leontiadis. 2017. Malicious Browser Extensions at Scale: Bridging the Observability Gap between Web Site and Browser. In *USENIX Security Workshop on Cyber Security Experimentation and Test*.
[22] Chrome Developers. 2021. Manifest v3 : Web Request Changes. https://groups.google.com/a/chromium.org/g/chromium-extensions/c/veJy9uAwS00/m/9iKaX5giAQAJ
[23] Christian Dresen, Fabian Ising, Damian Poddebniak, Tobias Kappert, Thorsten Holz, and Sebastian Schinzel. 2020. CORSICA: Cross-Origin Web Service Identification. In *ACM ASIA CCS*.
[24] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *ACM CCS*.
[25] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. 2017. Measuring HTTPS adoption on the web. In *USENIX Security*.
[26] Google. 2018. *Chromium Blog*. https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html
[27] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. 2015. May I?-Content Security Policy endorsement for browser extensions. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.
[28] IETF. 2012. *HTTP Strict Transport Security (HSTS)*. https://tools.ietf.org/html/rfc6797
[29] IETF. 2013. *HTTP Header Field X-Frame-Options*. https://tools.ietf.org/rfc/rfc7034
[30] IETF. 2016. *Initial Assignment for the Content Security Policy Directives Registry*. https://tools.ietf.org/html/rfc7762
[31] Apple Inc. 2022. Converting a Web Extension for Safari. https://developer.apple.com/documentation/safariservices/safari_web_extensions/converting_a_web_extension_for_safari
[32] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. 2015. Trends and lessons from three years fighting malicious extensions. In *USENIX Security*.
[33] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security*.
[34] Ravie Lakshmanan. 2021. Over a Dozen Chrome Extensions Caught Hijacking Google Search Results for Millions. https://thehackernews.com/2021/02/over-dozen-chrome-extensions-caught.html. Accessed on 2021-04-27.
[35] Microsoft. 2022. Overview and timelines for migrating to Manifest V3. https://docs.microsoft.com/en-us/microsoft-edge/extensions-chromium/developer-guide/manifest-v3
[36] Mozilla. 2021. *mozilla/web-ext*. https://github.com/mozilla/web-ext
[37] Mozilla Add-ons Community Blog. 2019. *Add-on Policy and Process Updates*. https://blog.mozilla.org/addons/2019/05/02/add-on-policy-and-process-updates/
[38] Mozilla Developer Network. 2012. *XMLHttpRequest*. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[39] Mozilla Developer Network. 2021. *Cross-Origin-Embedder-Policy*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy
[40] Mozilla Developer Network. 2021. *Cross-Origin-Opener-Policy*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy
[41] Mozilla Developer Network. 2021. *Cross-Origin-Resource-Policy*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP)
[42] Mozilla Developer Network. 2021. Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[43] Mozilla Developer Network. 2021. Referrer-Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy
[44] Mozilla Developer Network. 2022. Manifest v3 in Firefox: Recap & Next Steps. https://blog.mozilla.org/addons/2022/05/18/manifest-v3-in-firefox-recap-next-steps/
[45] Mozilla Developer Network. 2022. Sitemap - Firefox Extensions. https://addons.mozilla.org/api/v5/addons/search/?app=firefox&type=extension
[46] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *IEEE S&P*.
[47] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. You've Changed: Detecting Malicious Browser Extensions through Their Update Deltas. In *ACM CCS*.
[48] Raffaello Perrotta and Feng Hao. 2018. Botnet in the browser: Understanding threats caused by malicious browser extensions. In *IEEE S&P*.
[49] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies.. In *NDSS*.
[50] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. 2022. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In *USENIX Security*.
[51] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. 2022. The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions. In *USENIX Security*.
[52] Dolière Francis Somé. 2019. Empoweb: empowering web applications with browser extensions. In *IEEE S&P*.
[53] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. 2020. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*.
[54] Microsoft Edge Support. 2021. Add, turn off, or remove extensions in Microsoft Edge. https://support.microsoft.com/en-us/microsoft-edge/add-turn-off-or-remove-extensions-in-microsoft-edge-9c0ec68c-2fbc-2f2c-9ff0-bdc76f46b026
[55] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. 2015. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *IEEE S&P*.
[56] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. 2019. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX Security*.
[57] W3C. 2017. *Referrer Policy*. https://www.w3.org/TR/referrer-policy/
[58] W3C. 2020. *Permissions Policy*. https://www.w3.org/TR/permissions-policy-1/
[59] W3C. 2021. *Fetch Metadata Request Headers*. https://www.w3.org/TR/fetch-metadata/
[60] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. 2015. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *WWW*.

A EXTENSION INSTRUMENTION & CONTENT-SECURITY-POLICY

We check for the existence of a `content_security_policy` definition within the manifest. This is necessary as the framework relies on XMLHttpRequests [38] to forward the recorded set of headers to our logging server for post-processing and analysis. The extension's CSP potentially governs such interactions. If not specified, this defaults to `script-src 'self'; object-src 'self'`. Since XHRs are governed by `connect-src` (or the fallback `default-src`), we can omit any rewriting in case of the default policy. For any other policy, we first determine if either `connect-src` or `default-src` is specified by parsing the CSP as JSON. When `connect-src` is specified, we append the source expression `http://localhost` server, where we collect all the header data. In case only, `default-src` is specified. we define `connect-src` directive in the policy with our server address. This is because `default-src` serves as a fallback directive for other directives, and simply adding the server address to its values means we modify the policy beyond our needs.

B EXAMPLE OF UNSAFE CSP POLICY INJECTIONS

```
default-src https:; script-src https: 'unsafe-inline'
↪ 'unsafe-eval' blob: 'unsafe-inline'; style-src
↪ https: 'unsafe-inline'; ... base-uri
↪ https://*.gracenote.com
```

Listing 3: Trivially bypassable CSP injected by 3 extensions on *theguardian.com*.

C SECURITY IMPACT OF ALTERATIONS

We continue the discussion on the impact of alterations among individual security headers from Section 6 here.

Origin & Referer. We found 263 extensions that injected either of the two headers. 26 extensions modified the `Origin` while 57 modified the `referer`. One of them also modified the origin to chrome-extension://*, as if the extension initiated the request. Further, 29 extensions entirely dropped the two headers. Any alteration to these header values may harm the applications' security if the server-side processes the request differently based on their availability and set values, e.g., filtering cross-origin requests.

Fetch Metadata. Four extensions injected the `Sec-Fetch-Dest` header set to *document*, indicating that it is a user-initiated top-level navigation request. One other extension injected this, set to *empty*, which is rather harmless. 11 extensions modified the existing non-document header value (e.g. iframe, script and style) to *document*. One extension dropped the *Sec-Fetch-User* header, and the other two injected this, indicating that the request is user-initiated. Altering these headers may cause security issues if the server-side processes the request differently based on this header, e.g., blocking non-user-initiated requests or certain unexpected requests. For `Sec-Fetch-Mode`, nine extensions injected this header with value either set to *cors* or *navigate*. Only one of them dropped this header. One extension modified every existing header value with *navigate*, indicating user-initiated navigation to the server, which could also cause security concerns when handled differently. Lastly, for *Sec-Fetch-Site*, we found five extensions injecting this header, set to *none*, whereas ten other extensions injected the header set to *same-origin*. Only one of them dropped this header. Notably, ten extensions modified the existing header value by setting it to *none*, which may be potentially unsafe as it indicates a user-initiated request. Further, two of them set the current header value to *same-site* and the other two set this to *same-origin*. Any of the modifications mentioned above can have a negative security impact if the server-side refers to these headers before processing the requests, e.g., filtering out cross-site requests or limiting cross-origin requests.

HSTS. 13 extensions dropped the HSTS header on 156 domains, the security impact of which depends on the presence of the affected domains in the preload list. 85/156 domains also sent `includeSubdomains` while 57 of them included `preload` as well. Six other extensions only made syntactic changes to HSTS, causing no security impact, while two of them added `includeSubdomains` to HSTS sent by tmall.com, potentially enforcing security. Notably, one of them stripped `preload` from HSTS on two domains. This will not have any impact if the domains are already on the preload list.

COOP, COEP, CORP. We observe that one extension injects the `COOP` set to *same-origin*, enforcing cross-origin isolation across multiple windows. On the contrary, four others dropped this header while one of them also modified the value from *same-origin-allow-popups* to *unsafe-none*, thus, disabling the desired isolation on the client-side. For `COEP`, one extension injected this with *require-corp* value, intended to restrict cross-origin resources. However, this might break certain functionalities if the cross-origin resources do not include corresponding policies. Two other extensions dropped this header, disabling the cross-origin restrictions on embedded resources. Lastly, for the `CORP` header, two extensions injected this with the value set to *cross-origin*, meaning there is no restriction enforced on cross-origin resources. Six of them also dropped this header. While dropping this may not have a security impact, it may still block cross-origin resources if the embedding page sets the COEP header or the target resource is not CORS-configured.

Set-Cookie. We found nine extensions that injected the `Set-Cookie` header on 372 domains. Four of these always set cookies with `samesite=none` attribute while five of them set `samesite` to *strict* or *lax* only in 15 altered instances. This could potentially allow trackers to identify users online. Nine other extensions dropped this header, which is harmless from a security viewpoint. 22 extensions also modified the existing header values. These modifications range from altering security attributes (e.g. *samesite*, *secure*, etc.) to modifying cookie values. Altering the existing `Set-cookie` header may lead to potential issues on the client-side as it may serve as a potential tracking vector if not configured properly or even leak privacy-sensitive information.

X-Content-Type-Options. Two extensions injected this header, enforcing protection against MIME-type sniffing vulnerabilities. While ten of them dropped this header, and one of them modified the existing header value by setting it to a non-standard value, thus disabling the security restrictions.