Open Access Alert: Studying the Privacy Risks in Android WebView's Web Permission Enforcement

Trung Tin Nguyen
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
tin.nguyen@cispa.de

Ben Stock
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
stock@cispa.de

Abstract

Besides rendering pages in common browsers like Chrome, it is customary for apps to rely on WebViews to display web pages. While browsers handle permissions through user prompts for each visited site, WebViews require developers to manage web permission requests individually, leaving significant room for error. However, to date, the community lacks insight into the current developers' practices of WebView's permission enforcement.

To address this research gap, we present the first large-scale study on the implementation of WebView regarding web permission enforcement in the wild, focusing on Android apps. Particularly, we develop an automated pipeline to detect apps that utilize WebView to display websites to users but lack proper web permission enforcement, which we refer to as *privacy-harmful apps* (PHAs). Our pipeline flagged 12,109 potential PHAs that compromise user-sensitive data due to a failure to implement web permission enforcement. Among these potential PHAs, we further demonstrate how malicious apps without sensitive permissions can exploit 2,219 PHAs through a confused deputy attack to load targeted malicious websites that access sensitive data like location, camera, and microphone simply by starting these PHAs. Our results highlight a notable privacy risk – including apps with over 500 million installations – as any website can secretly collect user data while browsing online, and malicious apps can abuse such PHAs to collect user sensitive data at scale.

To help developers, we notify affected developers and gather insights from their feedback. Our findings reveal widespread and often misunderstood issues, emphasizing the necessity of collaborative efforts among stakeholders to address these privacy concerns. Based on our insights, we further derive concrete recommendations for developers to mitigate privacy risks associated with WebView.

CCS Concepts

• **Security and privacy → Mobile platform security**; **Usability in security and privacy**.

Keywords

Android WebView, Web Permission, Usable Security and Privacy

ASIA CCS '25, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/25/08
https://doi.org/10.1145/3708821.3710821

ACM Reference Format:
Trung Tin Nguyen and Ben Stock. 2025. Open Access Alert: Studying the Privacy Risks in Android WebView's Web Permission Enforcement. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3708821.3710821

1 Introduction

Web browsers serve as the primary gateways between users and the vast digital content of the Web, e.g., allowing users to interact with websites, multimedia, and other online resources. To enhance user experience, web browsers have the capacity to access devices' sensitive information, i.e., have significant privacy and security implications [64]. For example, to offer location-based recommendation services, they require access to the device's GPS. However, granting web browsers access to devices' sensitive information without any restrictions may become a major problem for individuals' rights (e.g., users are secretly tracked and profiled online [25, 45, 48, 49]). As such, to protect user privacy, web browsers widely implement the web permission mechanism (see Section 2.1), which provides users the ability to explicitly grant or deny websites access to the devices' sensitive information for each website [64].

Historically, desktop browsers have been the primary portals for viewing web pages. However, in today's digital era, people are not just accessing web pages via desktops but also extensively through smartphones, i.e., over 55% of website traffic comes from mobile devices [21]. In mobile, the number of fully-developed browser apps is limited due to the complexity of web browsers as software, which demands extensive engineering to build from scratch [54]. Consequently, many browser apps (or apps that allow users to visit websites) are built on top of WebView in Android [54] (i.e., a component that allows developers to display web-based content within their apps [7]).

While the Android system employs an OS permission-based mechanism to govern access to sensitive data, it is important to note that the permissions granted to the app should not automatically extend to the content loaded within the WebView. This means that when a user grants permissions to the host app, those permissions apply only to the app itself and its own functionalities, not to the content displayed within WebView. This distinction is important for developers to understand to ensure the security and privacy of Android apps. In practice, Android's WebView provides developers with complete control and customizable configurations for handling web permission requests, thereby leaving significant room for error. On the contrary, web browser vendors have strictly implemented permission mechanisms allowing users to explicitly grant or deny access to sensitive data for each website.

Prior research mainly focuses on Android OS permissions, including security and privacy measurements [13, 23, 50, 56], user perception studies [10, 11, 57, 65], concerns vulnerabilities related to component hijacking (e.g., permission leakage, unauthorized data access, and intent spoofing) [38, 44, 63], the impact of timing and rationales on users' decisions [19], and the development of new permission models [27, 37, 55, 73]. While the problem of component hijacking and permission delegation has been well studied in the literature, the primary focus has been on the misuse or lack of enforcement of Android OS permissions, particularly those related to permission-protected APIs [38, 44, 50, 63]. However, despite the distinction between Android OS permissions and web permissions, and the importance of enforcing web permissions in Android apps, there is a notable lack of understanding within the community regarding how developers currently enforce WebView permissions. Research concerning Android's WebView primarily examines vulnerabilities arising from the security risks posed by developers explicitly exposing Java code to WebView [14, 42, 47]. Recently, Beer et al. [9] first attempted to manually study the potentially vulnerable due to the lack of web permission enforcement of two popular third-party libraries from a small set of Android apps (i.e., 250 apps). However, manual methods alone are insufficient for fully understanding real-world web permission implementation in WebView and developers' current practices.

To fill this research gap and gain an understanding of how web permissions are currently enforced in Android's WebView and to see if app developers adhere to established standards for user privacy protection [64], we conduct the first large-scale study on the implementation of WebView regarding web permission enforcement in the wild, focusing on Android apps[1]. Particularly, we develop a static and dynamic analysis pipeline that automatically detects apps that use WebView to display web pages to users but lack proper web permission enforcement, namely privacy-harmful apps (PHAs). Applying to a set of 276,760 Android apps in Google Play, we identify 54,605 apps that instantiate WebView and enable JavaScript execution within their WebView settings. By searching for bytecode-level signatures of misconfigured implementations in web permission enforcement within WebView, our pipeline detects 12,109 potential PHAs (out of 54,605 apps) that compromise user-sensitive data due to a failure to implement web permission enforcement. Such a vulnerability poses severe privacy threats, allowing websites to silently harvest user data during browsing sessions and enabling malicious actors to exploit these potential PHAs to silently aggregate data on a large scale. For example, PHAs may have more permissions than malicious apps, allowing the malicious apps to exploit this by conducting a confused deputy attack and load targeted sites that collect user sensitive data.

Further, we to demonstrate how malicious entities can exploit those PHAs by simply launching those potential PHAs, i.e., without performing a comprehensive context analysis to execute the app. Particularly, our dynamic testing pipeline identifies 2,219 PHAs that enable any website to access devices' location, camera, and microphone without informing users. Notably, improper enforcement of web permissions in Androids' WebView is a privacy issue that affects not only non-browser apps but also highly popular browser apps with hundreds of millions of installations (e.g., Phoenix Browser), posing a significant risk to user privacy. Our findings reveal evidence of open permission access with potential impacts on billions of users. To support developers in fixing these privacy problems, we perform a notification campaign, reaching out to the developers involved with emails to inform them. Our research findings indicate that this issue is not only prevalent and widespread but also frequently misinterpreted, highlighting the need for a collective effort from all stakeholders to mitigate these privacy risks effectively. In summary, our paper makes the following contributions:

- We perform the first large-scale study on the implementation of web permission enforcement in Androids' WebView.
- We develop a pipeline (including dynamic and static analysis) that automatically analyzes Android apps to identify PHAs. Our pipeline demonstrates how straightforward it is to abuse these PHAs to collect users' sensitive data on a large scale.
- We notify affected developers, gather insights, and provide recommendations to all stakeholders, urgently addressing these problems.

Organization. The paper is structured as follows. Section 2 describes the background of web permission standards, web permission enforcement in Androids' WebView, our privacy threat models, and the related work. Section 3 presents our approach to identifying the lack of web permission enforcement in Android apps. Section 4 presents our large-scale analysis of Android apps and demonstrates our approaches to detecting PHAs and potential PHAs. Section 5 describes our email notifications and presents the feedback from developers. Section 6 discusses our findings. Section 7 draws conclusions.

2 Background and Threat Models

This section describes the background of web permissions, Android OS permissions, and web permissions in Androids' WebView. We also delve into privacy threat models, app developers' responsibilities, legal implications, and related work.

2.1 Web Permissions

Based on the web standards [64], accessing browsers' sensitive information, e.g., GPS data, requires explicit granting from users. Users' consent to allow a website to access a sensitive data is generally given and controlled through the browser UI. Accordingly, most browsers (both on desktop and mobile) strictly implement these requirements. Generally, websites can use privacy-sensitive JavaScript APIs to access browsers' sensitive information, i.e., Table 1 presents a list of privacy-sensitive JavaScript APIs. For example, Listing 1 shows the JavaScript code for using Geolocation API to collect the current position of users. Particularly, when the *navigator.geolocation.getCurrentPosition* is called (at line 2), if the location permission has not been granted, the browsers will prompt to ask for user decisions (either allow or deny access). Then, if the users allow access, the web page can access the users' GPS data (lines 3 and 4). Depending on each browser app, the permission

[1]We focus on Android apps since iOS WebViews enforce more stringent web permissions by default. In iOS, websites must directly request user permission through the native iOS system without overburdening developers.

Name	API
Background Synchronization API	background-sync
Clipboard API	clipboard-read, clipboard-write
Geolocation API	geolocation
Local Font Access API	
Media Capture and Streams API	microphone, camera
Notifications API	notifications
Payment Handler API	payment-handler
Push API	push
Sensor APIs	accelerometer, gyroscope, magnetometer, ambient-light-sensor
Storage Access API	storage-access
Storage API	persistent-storage
Web Audio Output Devices API	speaker-selection
Web MIDI API	midi

Table 1: List of privacy-sensitive JavaScript APIs [18].

```
1  function getCurrentLocation() {
2    navigator.geolocation.getCurrentPosition((pos) => {
3      console.log(pos.coords.latitude);
4      console.log(pos.coords.longitude);
5    });
6  }
```

Listing 1: Getting the devices' GPS data via Geolocation API.

```
1  public class WebClient extends WebChromeClient
2  {
3    @Override
4    public void onGeolocationPermissionsShowPrompt(String origin,
     ↪  GeolocationPermissions.Callback callback) {
5      callback.invoke(origin, true, false);
6    }
7
8    @Override
9    public void onPermissionRequest(PermissionRequest request) {
10     request.grant(request.getResources());
11   }
12 }
```

Listing 2: Granting permissions to Androids' WebView.

request prompt can be shown differently. For example, for a particular website, a prompt is shown every visit or the user's decisions will be persisted for the next visit.

2.2 Web Permissions in Android Apps

Mobile platforms like Android use an OS permission system to control app access to devices' hardware and other sensitive resources [28], such as GPS data. For Android prior to 6.0, users have to accept or reject all permissions an app requests at installation, without the option to grant permissions individually, leading to over-privileged apps with access to more sensitive data than necessary [22, 30, 33, 50]. Starting with Android 6.0, the runtime permission model was introduced, improving user security and privacy protections [10]. It aims to connect permission requests to specific functionalities at runtime, and developers can clarify the need for access by providing additional explanations.

However, in Android apps using WebView for displaying websites, users granting permissions (Android OS permissions) to the host app does not by default mean they acknowledge every loaded website within the WebView having access to their sensitive data. More importantly, it is impossible for users to know when and where certain data access takes place unless the app developers explicitly clarify it. For example, to access the device's GPS data, an app must first request permission from the Android OS (e.g., it is reasonable for users to grant location permission since it is a weather forecast app, not an alarm app). Once this permission is granted, the app can then share the location data with any website within a WebView without additional permission requests. This highlights a key difference in the timing and responsibility of the permission prompt — the Android OS manages the initial permission, but once granted, the app has the privilege to use this permission in its interactions with websites in WebView without further oversight from the OS or users. Therefore, developers have to ensure that users' sensitive data are not made accessible to any websites without the users' intervention or consent by default. It is different from traditional browser apps, which strictly adhere to web standards regarding web permissions. In particular, traditional browser apps manage web permissions and privileges granted to each website or web application. Users can grant or deny these permissions, often through prompts or dialog boxes that appear when a website requests access to specific resources or functionality.

While browsers like Chrome or Firefox by default prompt for user consent for each website requesting sensitive data, Android's WebView grants developers full control over web permission handling. In particular, app developers have to write custom code to manage web permission requests. Generally, when the host app receives a permission request from a web page (see Listing 1), it is up to the app first to request those permissions from the device (Android OS permissions), which can be done using *requestPermissions* of the Android framework [6]. Afterward, it is up to the app to present the UI, which asks for the user to grant the web page the ability to access a sensitive data, e.g., GPS location.

In Android's WebView, the *WebChromeClient* class manages permission requests from web pages (see Listing 2). The WebView will notify the host app that web content from the specified origin is attempting to use the sensitive information (see Listing 1), but no permission state is currently set for that origin (consists of the host, scheme, and port of a URI). The host app will invoke the specified callback with the desired permission state. Specifically, if the web content from a specified origin is attempting to use the Geolocation API, but no permission state is currently set for that origin, the *onGeolocationPermissionsShowPrompt()* callback will be invoked (from lines 3 to 6 in Listing 2). For other privacy-sensitive JavaScript APIs (such as Camera, Microphone), the *onPermissionRequest()* will be invoked (from lines 8 to 11 in Listing 2). The host app must then invoke *PermissionRequest#grant()* or *PermissionRequest#deny()* methods in order to deny or grant access. If these two methods are not overridden, all web permission requests are denied by default.

2.3 Privacy Threat Models

Androids' WebView is desired to display trusted first-party web pages [7]. Google suggests using Chrome for web page display, but many app developers still prefer WebView for a seamless user experience, allowing web page embedding within apps without

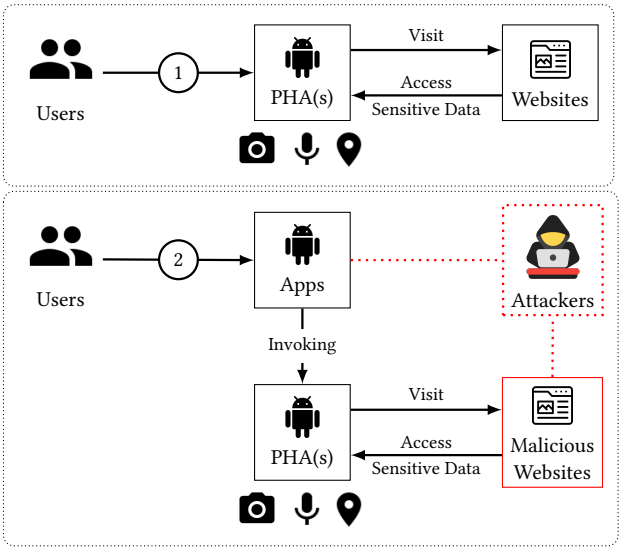

Figure 1: Two privacy threat models regarding the lack of web permission enforcement in Android's WebView.

the need to navigate away. Further, many Android browsers have been built on top of WebView [54]. Doing so raises many security and privacy problems, especially when visiting third-party web pages [14, 31, 42, 47, 71, 72].

While many studies have examined security concerns in Android's WebView, the research community lacks insight into the real-world behavior of web permission implementation within Android's WebView. In this work, we identify apps as privacy harmful (dubbed *Privacy-Harmful Apps* or *PHAs* for short) if they use WebView without implementing web permission enforcement (see Section 2.1). This poses a significant privacy risk, as users' sensitive information can be covertly collected when using such PHAs. Besides, malicious apps can exploit these PHAs to gather user data without their knowledge. In practice, users may grant permission (Android OS permission) to host apps without realizing that every website they visit gains access to their sensitive data.

In the following, we present our privacy threat models of Androids' WebView regarding web permissions.

2.3.1 Web Browser and In-App Browser Apps. Browser apps are purpose-built for Internet browsing, such as accessing and exploring websites. While launching an external browser app is a heavy context switch for users, many apps also allow users to visit websites within the apps (which allows users to remain within the app while browsing online). In practice, most of these browser and in-app browser apps are built on top of Androids' WebView because the browser engine is complex and requires significant engineering effort to develop from scratch. However, app developers do not correctly implement the privacy protections regarding web permissions. For example, in Listing 2, access to Geolocation and other sensitive resources is granted without any prompts to inform users. As such, when users are using these PHAs to browse the websites, all their personal data is at risk. Specifically, all the websites that they have visited can collect users' information without their awareness (case 1 in Figure 1). This creates a privacy threat since the attackers can easily trick the victim into opening malicious websites [42].

2.3.2 Web Browsable Apps. The apps are not purpose-built for Internet browsing nor provide in-app browsing functionality. However, as a part of these apps, they need to load websites using Android's WebView. However, the app developers unintentionally expose the apps' functionality to other apps (or do not prevent the loading of third-party websites), which leads to the app being tricked into performing an undesirable action. For example, in case 2 in Figure 1, the malicious apps without sensitive permissions can invoke these PHAs to visit the malicious websites (which collect users' sensitive data) via an Intent, i.e., communication between different apps. We note that attackers can also apply the Intent communication to the browser and in-app browser apps since they are purpose-built for Internet browsing.

2.4 Legal Implications

It is known that app developers are nowadays in a disadvantaged position, where third parties make it cumbersome for them to comply with data protection regulations around the globe, e.g., the General Data Protection Regulation (GDPR) in the European Union and the European Economic Area, the California Consumer Privacy Act (CCPA) applies to California residents in the US [48, 49]. However, as first-party data controllers (i.e., decide why and how personal data is processed), developers are legally responsible for ensuring that the users' data is protected.

For example, GDPR Art. 25 states: *"The controller shall implement appropriate technical and organizational measures for ensuring that, by default, only personal data which are necessary for each specific purpose of the processing are processed. That obligation applies to the amount of personal data collected, the extent of their processing, the period of their storage, and their accessibility. In particular, such measures shall ensure that* ***by default personal data are not made accessible*** *without the individual's intervention to an indefinite number of natural persons."*

As such, developers have to implement appropriate technical measures to protect the rights of data subjects and be transparent about how data is collected and used.

2.5 Related Work

Researchers have conducted many studies on the privacy and security of browser apps. Leith et al. [35] compared privacy features in major browsers, Luo et al. [40, 41] uncovered UI vulnerabilities in mobile browsers and evaluated for security features. Lin et al. [36] identified privacy threats in autofill features, while Wu et al. [67] discovered vulnerabilities compromising user data. Kondracki et al. [34] analyzed security trade-offs in data-saving features. Nomoto et al. recently proposed a framework for analyzing web browser behavior, focusing on web permission implementations of major browsers on different platforms and highlighting that the inconsistencies can lead to user tracking and unknowingly granting permissions to malicious sites [51]. Luo et al. [39] conduct the first large-scale study of certificate validation in mobile browsers, revealing inconsistencies that may expose users to man-in-the-middle and spoofing attacks. Debnath et al. [16] shows many proxy-based mobile browsers (i.e., use proxies for traffic compression and censorship circumvention)

Figure 2: Overview of our methodology to identify privacy-harmful apps (PHAs).

downgrade the quality of TLS sessions, exposing users to potential security and privacy threats. On the other hand, Pradeep et al. conducted an extensive analysis of Android web browsers' data protection, showing both protective and harmful privacy behaviors across different browsers [54].

While many mobile apps design in-app browsing interfaces for a seamless user experience, reducing the need to switch between the app and standard browser apps, Zhang et al. [74] show that poorly designed or customized in-app browsing can introduce usability and security risks. However, despite the importance of permission mechanisms in protecting user privacy, the community primarily focuses on the standard browser apps and lacks insight into Androids' WebView permission behavior, while Androids' WebView component is widely used to display websites within mobile apps.

Regarding Androids' WebView, many studies have identified possible attacks on WebView [14, 31, 42, 47, 71, 72]. Chin et al. [14] introduced a tool to detect security flaws in apps employing WebView, suggesting enhanced security policies. Luo et al. [42] discussed the security implications of WebView usage in mobile apps, highlighting how its features could weaken web security infrastructure. Mutchler et al. [46] evaluated the prevalence of vulnerabilities in mobile web apps, emphasizing the need for API improvements. Neugschwandtner et al. [47] presented case studies on security threats associated with WebView, collectively contributing to our understanding of the security and privacy challenges in this domain. Such attack models outlined in prior work primarily originate from app developers explicitly exposing sensitive APIs through interfaces registered to WebView. This allows JavaScript code in embedded web pages to invoke these interfaces to access sensitive data. Alternatively, attacks may occur due to network compromise by an attacker or the compromise of a server used to deploy malicious JavaScript. Orthogonal to these attack models, our threat models originate from the lack of implementation of web permission enforcement by app developers, rather than intentionally exposing sensitive data to WebView. Further, our study focuses on understanding the current practices of Android's WebView permission enforcement among developers in the wild and examining its vulnerabilities that risk users' privacy.

3 Methodology

To gain an understanding of how web permissions are currently enforced in WebView and to see if app developers adhere to established standards for user privacy protection, we perform the first large-scale study on the implementation of WebView concerning permission enforcement in real-world scenarios. Specifically, we propose an automated and scalable pipeline to identify misconfiguration of WebView regarding the web permission enforcement in Android apps, i.e., the lack of permission enforcement when webpages invoke privacy-sensitive JavaScript APIs to access sensitive data (see Figure 2). In particular, we first perform static analysis to detect *potential PHAs*. We referred to these apps as potential PHAs because their behaviors had not been confirmed through dynamic testing, which might result in over-approximation. Specifically, our approach begins by employing static analysis to identify apps utilizing WebView, followed by the static analysis of instances where web permission enforcement is lacking. From those identified potential PHAs, we then conduct a dynamic testing that automatically invokes the apps' functionality to open our test pages containing JavaScript code to access sensitive data through privacy-sensitive JavaScript APIs (see Section 2.1). The underlying assumption is that we do not interact with the app's potential permission dialogues regarding websites' permission requests (if there are any), implying that if our test pages can gain access to users' sensitive data, the access is not properly safeguarded by prompts, other protection mechanisms implemented by the app developers, or there are no enforcements at all. Our dynamic analysis aims to demonstrate how easily these PHAs can be invoked to open targeted malicious websites and collect users' sensitive information on a large scale, without requiring a complex execution context.

Based on the collected results from our test pages, we can identify the lack of web permission enforcement (or improper protection) of the Androids' WebView that leads to user-sensitive data being exposed to any websites. Specifically, we consider it improper protection if our test pages can be successfully loaded and can collect sensitive data without interactions with the app user interface. This work focuses on Location, Camera, and Microphone because Androids' WebView mandates explicit implementation by the host apps (i.e., developers have to explicitly write code to grant these permissions to WebView). Data that is either granted by default or not supported by Android's WebView is excluded from our analysis, e.g., sensor information. More importantly, data related to Location, Camera, and Microphone permissions are considered personal data under GDPR, which strictly regulates how to collect, process, and protect these data. For example, to be legally compliant, an online service is required to obtain users' explicit consent before collecting personal data if such a service uses the data for its own purposes.

We note that Android recently tried to improve privacy protections by displaying indicators when an app uses a camera and a microphone, i.e., employed in Android 12 (released in 2021), which displays an icon in the status bar when the camera or microphone is being used [29]. However, the indicator does not prevent the attack from happening, and users may not even notice it. Even if they do, it will already be too late since the malicious websites will have already taken photos.

```
[Method: onPermissionRequest]

r0 := @this: ([A-Za-z0-9]+(\.[A-Za-z0-9]+)+)\$[A-Za-z0-9]+;
$r1 := @parameter0: android.webkit.PermissionRequest;
$r2 = virtualinvoke $r1.<android.webkit.PermissionRequest:
↪ java.lang.String[] getResources()>();
virtualinvoke $r1.<android.webkit.PermissionRequest: void
↪ grant(java.lang.String[])>($r2)
return;
```

Listing 3: A bytecode-level signature of *onPermissionRequest* that grants all requested sensitive permissions to WebView.

```
[Method: onGeolocationPermissionsShowPrompt]

r0 := @this: ([A-Za-z0-9]+(\.[A-Za-z0-9]+)+)\$[A-Za-z0-9]+;
$r1 := @parameter0: java.lang.String;
$r2 := @parameter1: android.webkit.GeolocationPermissions$Callback
interfaceinvoke $r2.<android.webkit.GeolocationPermissions$Callback: void
↪ invoke(java.lang.String,boolean,boolean)>($r1, 1, 0);
return;
```

Listing 4: A bytecode-level signature of *onGeolocationPermissionsShowPrompt* that grants location permission to WebView.

3.1 Identify Potential PHAs by Static Analysis

In the following sections, we provide a detailed explanation of our methodology for detecting potential PHAs by using static analysis.

3.1.1 Apps Embed Androids' WebView. To identify apps that embed WebView, we conduct static analysis using the following criteria:

- We decompile the app to analyze its resources and select apps that have declared *WebView* in their layout files, i.e., located in "*res/layout*" directory.
- We then statically analyze the app code to identify the initialization of *android.webkit.WebView* instance.
- Finally, we only consider apps that enable JavaScript execution, i.e., *setJavaScriptEnabled(true)* should be called.

3.1.2 Web Permission Enforcement in WebView. To improve privacy protections, apps have to ensure that users are provided with appropriate prompts when a loaded website within an app executes privacy-sensitive JavaScript API(s) (see Section 2.1), such as GPS data. If an app's WebView does not require access to sensitive data, it should deactivate this capability. In Androids' WebView, developers must explicitly implement such web permission enforcement. For example, like standard browser implementations, upon a user's first visit to a website that requests location access, WebView should present a prompt to the user, seeking their consent to grant or deny the permission, or simply deny based on the apps' functionality.

To achieve this, developers have to override and implement the special callbacks that handle the permission requests from websites (see Section 2.2). For instance, the *onGeolocationPermissionRequestPrompt* method handles the location permission, and *onPermissionRequest* handles the access to the devices' camera and microphone. For developers, it's important to create prompts that interact with the app's UI thread, informing users about requested permissions (see Listing 5 in Appendix). If websites don't need access to sensitive data, overriding special callbacks isn't necessary, as Android's WebView disables execution by default. Also, if apps only load trusted first-party websites, prompts may not be needed. In such cases, developers must ensure WebView doesn't load arbitrary URLs. We define an app as a *potential* PHA if it overrides special callbacks without implementing user prompts. Additionally, if WebView gains access to sensitive resources without prompts and lacks validation to prevent opening third-party websites, it is classified as a potential PHA.

In particular, we begin by employing Soot [59], a static analysis framework designed for analyzing Java and Android apps, to decompile and statically analyze the app code to determine if the app has overridden these special callbacks. Specifically, we scan through the app's bytecode to search for the signature of *onGeolocationPermissionRequestPrompt* and *onPermissionRequest* methods. We then use Soot to construct the control flow graph (CFG) that includes all the reachable API calls, in which the entry points are these identified callbacks. Upon analyzing the CFG of these callbacks, we determine whether the app permits WebView to access sensitive information, and this process does not include the activation of UI components. We classify this scenario as a potential PHA.

Specifically, we follow a similar approach used in prior work [75]. We first generate bytecode-level signatures for two implementations of misconfigurations related to web permission enforcement in WebView (granting the requested permissions without any enforcement), i.e., Listing 3 allows all requested permissions for WebView and its loaded content, while Listing 4 grants location permissions (see Listing 2 for the original Java code of these two methods). We then use these two bytecode-level signatures to search for potential PHAs. This classification is grounded on the premise that the absence of UI component engagement could mean a lack of transparency or user consent. This analysis enables us to understand how permissions are handled within the app and whether they adhere to the web standards.

We further analyzed the implementation of the *shouldOverrideUrlLoading()* method within *WebViewClient*. In Android, this method allows developers to define custom behavior for handling URL loading within a WebView. This method is invoked when a new URL is clicked, enabling developers to decide whether to load the URL in the same WebView, open it in a different app, or perform some other action. As such, it is possible for those potential PHAs to have other privacy protections, e.g., preventing the apps from opening any third-party links and navigating the third-party links to standard browser apps, only loading first-party trusted links. Particularly, if a *shouldOverrideUrlLoading* is overridden, returning true causes the current WebView to abort loading the URL, while returning false causes the WebView to continue loading the URL as usual (which is the default behavior of WebView if the WebViewClient is provided).

3.2 Identify PHAs by Dynamic Testing

Recall that our dynamic analysis aims to demonstrate how easily these PHAs can be triggered to open targeted malicious websites and collect users' sensitive information on a large scale, without

requiring a complex execution context. As such, without performing a comprehensive context analysis for dynamic execution, we propose a straightforward approach that includes two steps. First, for a given potential PHA, we decompile the app to analyze the app manifest file to identify the potential browsable activities (apps' functionality that can be invoked by other apps). We then automatically invoke these browsable activities to visit our test pages. In practice, these two steps can be easily done by any installed app on the same device. A malicious app could simply launch these potential PHAs to visit targeted malicious websites without conducting a comprehensive context analysis of the those potential PHAs.

In the following, we outline how we conduct each of the steps in more detail.

3.2.1 Identify Browsable Activity. The Android apps' manifest file (*AndroidManifest.xml*) describes essential information about Android apps, including all activities, services, broadcast receivers, and content providers [4]. To identify the potential browsable activities (i.e., the activity can be launched by components of other apps), we first use *Androguard* to decompile the given app and extract its manifest file [3]. We then analyze the apps' manifest to select activities that have the attribute "*android:exported=true*", which determines whether an activity, service, or receiver is accessible to external apps. For instance, when one shares a file, the presence of available apps allows you to open the corresponding activity, meaning that it's declared in the manifest with "*android:exported=true*".

Further, we also want to determine which apps are browser apps, i.e., an app capable of opening arbitrary websites. More specifically, these apps must declare this capability in the apps' manifest file via Intent filters that handle *HTTP/HTTPS URL* schemes. For the Android OS to recognize the app as a browser app, it also has to specify "*android.intent.category.APP_BROWSER*" in the Intent filters. The Android OS uses this Intent-based filtering to display available browser options when users click a URL.

3.2.2 Visit Test Page. Our goal is to detect PHAs that expose sensitive data to any websites without web permission enforcement. To do that, we first install the apps and automatically grant the necessary permissions specified in the app manifest. The idea is that users may grant permission to host apps via Android OS permission. However, it does not mean that every website they visit gains access to their sensitive data.

Subsequently, from the list of browsable activities (identified from the previous step), we use Android Debug Bridge to invoke these activities to visit our test pages[2]. The test pages include JavaScript code that invokes a set of privacy-sensitive JavaScript APIs, i.e., Geolocation API and Media Capture and Streams API [15]. The collected data will be transmitted to our testing server and then saved in a persistent database. When the app opens the test pages, a visit record is created in the database, allowing us to monitor for any privacy issues and identify any access to sensitive data such as location, camera, and audio stream data. If we can successfully collect sensitive data through these JavaScript APIs during dynamic testing, the app is considered harmful to users' privacy.

[2]adb shell am start -n {package_name}/{activity_name} -a android.intent.action.VIEW -d "{testing_url}"

3.3 Limitations

While our analysis provides valuable insights into potential PHAs, we naturally face certain limitations, which potentially lead to incomplete results.

Like another static analysis approach, our pipeline might not capture some app behaviors that occur dynamically or more complex scenarios at runtime. For instance, we consider apps declaring WebView in layout files, but this may overlook those using only dynamically generated WebViews. Further, by combining CFG and bytecode-level signatures to detect improper web permission enforcement in *onPermissionRequest* and *onGeolocationPermissionsShowPrompt*, our approach can account for some variations of the signatures in Listing 3 and Listing 4, such as when developers add extra statements (e.g., logging). In these cases, the semantics of granting permission by default would remain unchanged. However, more complex patterns might be overlooked by our approach. For instance, our focus is primarily on apps that completely lack web permission enforcement in straightforward ways, i.e., granting permissions without invoking UI thread. However, more complicated scenarios exist. For example, an app might first request Android permissions (involving a UI thread) and perform certain operations when a website requests access to sensitive data. Our approach may overlook the need for prompts for such cases, since an Android permission dialog may appear on the first visit to a web page but not on subsequent visits, a scenario our static analysis does not account for. Generally speaking, all these are potential causes for false negatives.

Future research could reveal additional potential privacy harms, although this does not contradict our current findings. We acknowledge that our *purely* static analysis-based report on potential PHAs might lead to an overestimation, which is why we added a manual sampling and confirmation step. Nonetheless, it illuminates critical aspects of web permission enforcement in Android's WebView, uncovering potential privacy concerns for users. Future research can build upon our findings to achieve more precise results and explore these privacy problems in greater depth. Further, our analysis does not fully analyze the prevention of third-party website loading, which might not be solely managed by the *shouldOverrideUrlLoading* but can also be handled through other functions or network restrictions. However, this will not affect our results regarding the app having an implementation that simply returns false.

4 Large-Scale Study

In this section, we present findings from our analysis of Android apps on Google Play, focusing on WebView's web permission enforcement. Particularly, we detail the app dataset construction process, discuss potential PHAs that may pose privacy risks, and demonstrate how malicious entities can easily exploit those potential PHAs to access sensitive data without user awareness.

4.1 App Dataset Construction

Our analysis aims to assess the state of privacy harm of WebView in Android apps in the wild. Therefore, we crawled all free Android apps from July 2023 to September 2023 on the Google Play store based on the list of apps from AndroZoo, which has 5.8M Android apps' names [2]. However, because of country-specific restrictions,

the presence of outdated apps (removed from Google Play) and paid apps in AndroZoo's list, time limitations, and the access rate limit set by Google Play, we were unable to crawl all the apps. Specifically, we managed to crawl 821,468 free Android apps, which constitutes approximately 33.82% of the total number of free apps currently available on Google Play by November 2023 [68].

Focusing on these 821,468, we acknowledge that our findings cannot be generalized to all apps on Google Play, e.g., paid apps. However, this approach aligns with previous large-scale Android security research [24, 48–50, 53, 62]. We note that this limitation will not impact the findings of our study, including the privacy issues we discuss in this paper. We further applied a filtering step to get the most relevant apps to our study (e.g., focusing on apps that have the capacity to access devices' sensitive information, such as GPS data or the device's camera). Specifically, we consider only apps which meet the following criteria:

- Apps request INTERNET permission, which is required to visit websites on the Internet.
- Apps request at least one of the following permissions: LOCATION, CAMERA, MICROPHONE. These apps can access and disclose this sensitive information, classified as personal data under data protection laws (e.g., GDPR, CCPA).

As a result, we obtained 276,760 Android apps. In the next steps, we present how to identify potential PHAs by static analysis.

4.2 Identify Potential PHAs by Static Analysis

From 276,760 Android apps, we identified 102,436 apps that contain WebView within their code. Further examination showed that out of these, there are 54,605 apps that have enabled JavaScript execution within their WebView settings, i.e., *setJavaScriptEnabled(true)*, and have configured their implementation of *WebViewClient* through method *setWebChromeClient*, allowing them to handle JavaScript dialogs and other features, such as granting sensitive permissions to WebView. In the following, we focus on these 54,605 apps that are statically relevant to our study.

As a result of this step, we successfully constructed the control flow graphs (CFGs) for 28,185 apps out of a set of 54,605 apps. These particular apps had overridden and implemented at least one callback handling website permission requests. To be specific, 22,532 apps had overridden the *onPermissionRequest* callback, while 26,306 apps had overridden the *onGeolocationPermissionsShowPrompt*.

Notably, within the subset concerning *onPermissionRequest*, 18,873 apps — or 83.76% of the 22,532 that implemented this callback, were identified as originating from third-party libraries (see Figure 3). Similarly, for the *onGeolocationPermissionsShowPrompt*, 21,706 apps, accounting for 82.51% of the 26,306 utilized this callback, were classified as third-party libraries (see Figure 4). This finding suggests that a majority of WebView components are derived from third-party services or utilities commonly used across various apps.

This widespread integration of third-party solutions raises notable privacy concerns, particularly when acknowledging that over 80% of the overridden methods are contributed by these third-party services. As such, it can lead to significant privacy implications, as these services may have data handling and privacy practices different from those of the primary app developers. As noted in prior

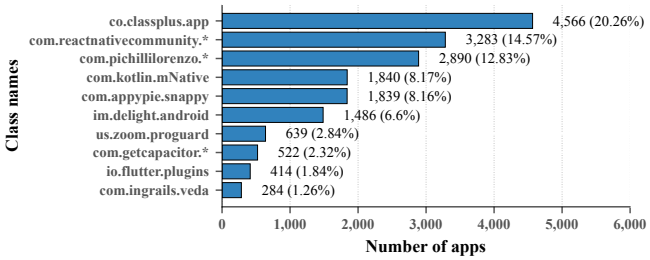

Figure 3: Top 10 third-party libraries overriding *onPermissionRequest*.

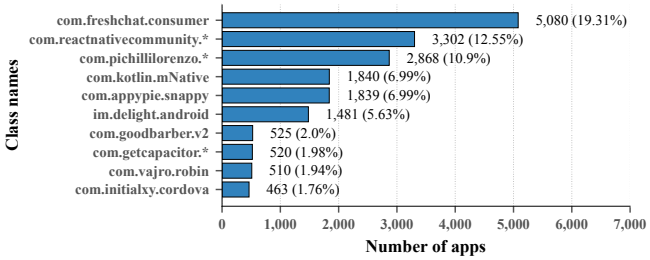

Figure 4: Top 10 third-party libraries overriding *onGeolocationPermissionsShowPrompt*.

research [8, 17, 20, 48, 58] , it is often observed that third-party components integrated into Android apps significantly influence their default behaviors, specifically exposing developers and users to severe threats of security and privacy vulnerabilities. For instance, these risks include the unauthorized disclosure of sensitive user information, the exploitation of the privileges of the host app, and the tracking of user activities, which potentially violate data protection laws. Our findings contribute to the existing body of evidence concerning the privacy concerns associated with the integration of third-party services in Android apps. Notably, this integration not only has the potential to expose user-sensitive data to a single third-party service but could also lead to the leakage of such data to a multitude of services on a larger scale.

We have identified 12,109 distinct potential PHAs, which constitute 15.35% of the 54,605 apps analyzed. Specifically, we discovered 11,430 apps that grant location access to every website loaded via the WebView component (matching the bytecode-level signature in Listing 4). Furthermore, 992 apps grant permissions, such as camera and microphone access to WebView without any form of verification (matching the bytecode-level signature in Listing 3). These apps are classified as potential PHAs since they do not implement any form of user notification or consent prompts. The lack of such protective measures means that any website accessed via WebView can retrieve user sensitive data, potentially without the user's knowledge or explicit permission.

Figure 5 and Figure 6 illustrate the top ten third-party libraries that have overridden the *onGeolocationPermissionsShowPrompt* and *onPermissionRequest* methods to grant permissions without enforcement. Specifically, regarding *onGeolocationPermissionsShowPrompt*, we observed that *com.freshchat.consumer* (Freshchat) accounted for 46.23% of the 11,430 identified potential PHAs. In the case of

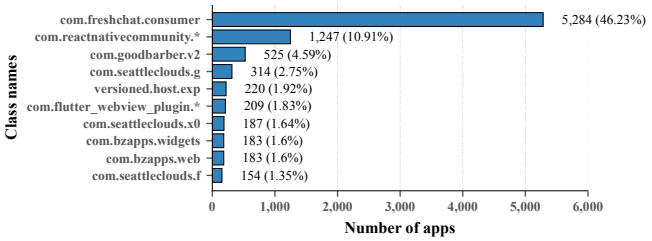

Figure 5: Top 10 third-party libraries overriding *onGeolocationPermissionsShowPrompt* to grant permissions without enforcement.

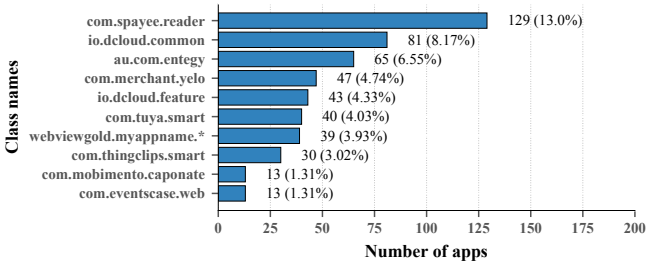

Figure 6: Top 10 third-party libraries overriding *onPermissionRequest* to grant permissions without enforcement.

onPermissionRequest, *com.spayee.reader* was responsible for 13% of 992 potential PHAs.

In our analysis, we discovered 5,691 apps, which represent 47% of the 12,109 potential PHAs that had implemented the *shouldOverrideUrlLoading* method. However, these 5,691 simply returned false. This implies that these 5,691 potential PHAs allow the WebView to load any links without restriction. Consequently, if these potential PHAs permit users to input arbitrary URLs, it creates an opportunity for malicious actors to deceive targets into opening malicious websites. This situation could lead to the secret collection of sensitive user data since there are no permission prompts to notify users, and there are no restrictions on the URLs that can be visited. In our study, we did not identify or differentiate whether the apps have the functionality to enable users to input arbitrary URLs. However, even if the app does not provide this feature, many apps utilize WebView to display their own web pages, e.g., the app's privacy policy usually links to third-party services where users can navigate to other sites with social media links or Google search. As such, without URL restrictions in place, users can easily navigate to other websites, thereby exposing their data freely to external sites.

Manually Analyzing Potential PHAs: To gain insight into the actual behaviors of those 12,109 potential PHAs, we conducted an additional manual analysis. Specifically, we first used Frida [26], i.e., a dynamic binary instrumentation toolkit, to intercept and modify the *WebView#loadURL* method for loading our test pages. We then installed the app on the physical device and manually browsed the app functionality within five minutes. This approach aimed to test whether potential PHAs would open our test pages and whether we can collect sensitive data successfully, indicating a lack of web permission enforcement or other protective mechanisms against the loading of third-party websites. We used this method to investigate the issue while interacting with the app's functionality, as manually inspecting the app's binary code is infeasible at scale. Out of the 12,109 potential PHAs, we successfully intercepted and modified 3,046 of them (without exception error from Frida). Other apps failed to use Frida for different reasons, such as installation issues on our physical devices due to unsupported libraries, or failure to hook or override the targeted method. We then randomly sampled 100 apps out of 3,046 apps since it is impractical to analyze all of them. Among 100 potential PHAs, we successfully installed and browsed the functionality of 59 apps. We could not analyze the remaining 41 apps due to internal login requirements (e.g., used internally by a company) or runtime errors while using them (e.g., crashed upon opening). Out of 59 apps, our test pages successfully collected sensitive data from 42 apps.

4.3 Identify PHAs by Dynamic Testing

In this section, we present our results from dynamic analysis, which aims to demonstrate how easily these PHAs can be triggered to open targeted malicious websites and collect users' sensitive information.

4.3.1 Experiment Setup and Overview. Our experimental tests were executed on three x86_64 Debian servers, each equipped with 192 CPUs and 1.5 TB of RAM. We utilized the Android Virtual Device for managing Android emulators and the Android Debug Bridge, a command-line tool, for communication with the devices, thereby automating our entire process (see Section 3). Throughout our experimentation, we used 180 Android emulators, especially using a Pixel 3 model operating on Android API 30. On average, the analysis for a single app, including installation time, the wait for the app to fully initialize, and loading the test page, took approximately one minute. This setup allows us to identify the apps that can access our test pages on a large scale. To confirm PHAs that expose sensitive information without permission enforcement, we retested the apps capable of visiting our test pages on physical devices (Google Pixel 6) to prevent scenarios where the app detects an emulator and behaves differently. Further, we randomly selected 1,000 apps that did not access our test pages through emulators and retested on physical devices. As a result, these apps were unable to load the test pages, even when tested on physical devices.

Out of 12,109 potential PHAs, we identified 8,421 distinct browsable activities derived from 7,887 apps. Among these browsable activities, a notable subset of 1,405 activities explicitly declared support for *HTTP/HTTPS URL* schemes, originating from 1,374 apps. Moreover, our dataset contained a total of 5 browser apps (based on the declaration on the apps' manifest file, i.e., *APP_BROWSER*).

We ran our pipeline for all of the 8,421 browsable activities. As a result, our pipeline successfully invoked 2,794 activities from 2,711 apps to open and visit our test pages. Notably, among these 2,794 activities, we only found 686 (25% of 2,794) activities that are from the set of 1,405 activities declared support for *HTTP/HTTPS URL* schemes. It indicates that in the majority of activities used to present web-based content within the app (75% of 2,794), developers do not intentionally expose the apps' functionality to other apps for presenting third-party web-based content. However, due to the lack of validation, other apps can invoke these activities to perform undesire functionality, e.g., open any websites.

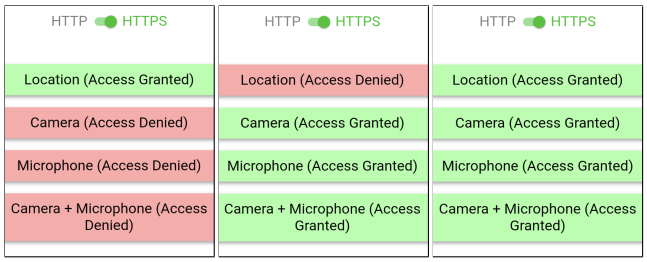

Figure 7: The taken screenshots while testing. The green button indicates that sensitive data was successfully collected.

We acknowledge that the success rate for opening the test page was relatively low (only 2,794 activities), which is a well-known limitation of dynamic analysis. The low coverage can be attributed to the diverse app types and their complex behaviors (e.g., a direct messaging app that requires selecting friends through several screens before using a WebView to open an in-app browser when sharing a link), which make comprehensive analysis challenging. For example, among other reasons, that could be due to the complex app-specific parameter handling, mandatory login requirements, and some apps not installing correctly in our test environment. However, these 2,794 activities allowed us to confirm the behavior and functionality of these apps regarding their implementation of web permission enforcement.

4.3.2 Privacy Harmful Apps in The Wild. Our experiment revealed 2,219 PHAs allowed our test pages to successfully collect sensitive data without any web permission enforcement. Through dynamic analysis, we proved that malicious entities can successfully exploit these 2,219 PHAs to load any website that accesses sensitive data without user knowledge. Figure 7 presents some sample screenshots that were taken from our analysis pipeline.

Table 2 presents the summary of our findings, where each column presents the type of sensitive data and the number of PHAs accordingly. Notably, sensitive data is not only exposed in isolation but also is often combined with others. For example, we found 285 PHAs exposed location and camera access, while another 51 PHAs exposed camera and microphone access. More seriously, we detected 165 PHAs permitted access to all of these three sensitive data (all, location, camera, and microphone), which raises a significant privacy risk to users.

Generally, our analysis indicates the PHAs occur across different app categories, as the top 3 categories that have more PHAs than others are Health & Fitness (206 PHAs), Education (170 PHAs), and Shopping (100 PHAs). Table 3 shows the top 10 PHAs in our dataset, ranking by the number of installations. Among these, Phoenix and Pi Browser are publicized as standard browser apps, which respectively have more than 500M installs and 10M installs. Notably, Phoenix Browser allows any website access to users' sensitive information, including location, camera, and microphone, without the user's explicit consent. This means a significant threat to user privacy and security, as it implies that personal data can be silently harvested while users navigate the web. Additionally, we also observed a lack of web permission enforcement by dating apps with

L	C	A	L+C	C+A	L+C+A
1,271	199	22	285	51	165

Table 2: Number of PHAs that are categorized by type of sensitive data. L = Location, C = Camera, A = Audio.

Package Name	Installs	Permission		
		L	C	A
com.transsion.phoenix	500M+	✓	✓	✓
com.ebay.kr.auction	10M+	✓	-	-
com.fordeal.android	10M+	✓	✓	-
com.lwi.android.flapps	10M+	✓	✓	-
com.saramart.android	10M+	✓	✓	-
pi.browser	10M+	-	✓	-
com.taobao.htao.android	5M+	✓	-	-
digifit.virtuagym.client.android	5M+	✓	✓	-
ch.local.android	1M+	✓	-	-
com.almart.android	1M+	✓	✓	-

Table 3: Top 10 PHAs ranked by installation numbers. The symbol (✓) indicates permission that can be accessed. L = Location, C = Camera, A = Audio.

millions of installations. For instance, in the C-Date app, we were able to silently invoke the app to take a picture, record the audio, and then transmit it to our server without any enforcements[3].

Further, we also want to understand the source of these problematic activities (differentiating between those originating from first-party versus third-party libraries). As such, we consider an activity to be a third-party activity (from third-party libraries) if there are more than ten apps that embed these activities. We note that the goal is not to identify particular third-party libraries but to determine whether these issues originate from shared libraries that could be utilized across multiple apps, including internal libraries employed by the same developer for various apps. Therefore, we choose not to use dedicated third-party library detection frameworks that could add significant runtime overhead, such as LibScout [8]. Further, these frameworks do not specifically target a particular activity identification, which is required for our purposes. Similar to any other static analysis, our approach naturally suffers from certain limitations, such as the app's code being obfuscated, and the results naturally only serve as a lower bound. However, it will provide an initial insight into the results of third-party libraries (see the second row in Table 2).

More specifically, upon manual examination of the top ten third-party libraries, we were able to find that eight out of ten originated from online app generators (which contributed to 231 out of 2,219 PHAs)[4]. These generators simplify implementation through drag-and-drop module assembly. However, it's crucial to note that this convenience introduces security risks that require careful mitigation to protect both end-users, as highlighted in a prior study by Oltrogge et al. [52]. Our findings also provide evidence of new privacy issues, relating to the absence of permission enforcement in WebView.

[3]The developers (Phoenix Browser and C-Date) have confirmed the issues were rectified in the latest version after our email notification in Section 5.
[4]For example, online app generators including *https://modolabs.com/*, *https://imweb.me/*, *https://appsgeyser.com/*, *https://yapp.li/en/*.

To sum up, our key takeaways and lessons learned are:

(1) Privacy issues with Android's WebView often arise in apps not primarily for web browsing. However, even purpose-built browsing apps, where privacy protection is crucial, face these problems.
(2) Most privacy issues arise from online app generators and third-party libraries, posing significant risks to users.
(3) At the same time, even first-party code frequently contains such errors, making it paramount to inform their developers about the issues.

5 Notifying App Developers

In our efforts to assist developers in addressing PHAs, we reached out to app developers to inform them about the privacy problems of their apps, particularly those that lack web permission enforcement in Androids' WebView, to prevent potential malicious actors from abusing these PHAs. Further, we also aim to understand the root causes of these issues. As such, we briefly inquire with developers about their awareness of the WebViews' privacy problems and their understanding of web permission enforcement in Androids' WebView. We aim to keep our inquiry brief to encourage more feedback instead of the complete surveys. In our email, we inform developers about our study's aims, and our approach and provide our contact details for any queries, noting that our institution does not require ethics approval for such research. We collect developers' contact by using email addresses from the Google Play Store. Like prior research, we used a web interface rather than an email to detail our findings and proof of concepts [12, 43, 48, 49, 60].

5.1 Email Notifications

On November 1, 2023, we sent a total of 5,099 emails to app developers, who are the point of contact for 2,219 PHAs and 12,109 potential PHAs. We note that the count of developers who received notifications is lower than the total number of PHAs and potential PHAs. This is because, in cases where a single developer manages multiple apps, we consolidated our notifications to ensure that each developer received a single email containing links to all relevant reports. Further, we also adhered to the best practices recommended by prior research, providing developers the option to opt out of our study. By November 27, 2023, we observed that 473 developers accessed our reports, who are responsible for 756 distinct apps. In addition, 58 unique developers answered our questions regarding the Androids' WebView. Further, one developer wanted to stop receiving communication from us on the subject. The response rate to our questionnaires was low, as might be expected from sending unsolicited emails to prospective participants [1, 48, 49, 66].

5.2 Developer Feedback

Among the pool of 58 respondents, each individual acknowledged the receipt of our email and conveyed their intention to consider its contents carefully. Notably, four respondents explicitly stated their decision to remove their respective apps from the app store in response to our communication. Further, it is important to note that not all developers responded to all the questions outlined in our email notifications.

In particular, among the 15 responses we received to our first question about WebView usage in mobile apps (*Q1. Did you intentionally allow your app's WebView to open any websites?*), 11 developers stated that they did not intend to allow WebView to open external websites. In contrast, 4 developers specifically designed their apps to incorporate WebView for the purpose of accessing external websites.

In response to our second question (*Q2. Did you intentionally allow JavaScript to access $permission$ features? Why did you not implement the prompts to request users' permissions when JavaScript access to sensitive data like standard browser apps (Google Chrome or Firefox)?*), 13 participants provided their insights regarding WebView's access to sensitive information. Surprisingly, 10 of them expressed no intention to allow WebView access to these capabilities. In contrast, only 3 respondents indicated that they intended to enable such access, but exclusively for a select set of trusted websites, including third-party domains. Notably, despite these intentions, our static analysis did not identify any enforcement measures to ensure the implementation of these intentions.

We received responses from 12 participants for our third question (*Q3. Are you aware that granting these permissions to the WebView without any privacy protection implementations (prompt to inform users) leads to user privacy at risk?*). Interestingly, there was a notable division in their level of awareness regarding the privacy implications of the issue at hand. Specifically, 5 respondents indicated that they possessed knowledge about the privacy concerns associated with the topic, demonstrating an understanding of the potential privacy risks and implications linked to their mobile applications. In contrast, the majority, comprising 7 respondents, expressed a lack of awareness regarding the privacy aspects of the issue. For non-responders, there might be a social desirability bias influencing them, possibly because they wish to avoid admitting their lack of knowledge by saying they do not know about the subject. This distinction highlights the diverse levels of awareness among developers when it comes to privacy considerations in their app development endeavors.

Regarding the final question (*Q4. What type of support (e.g., documentation or automated tools) would benefit you to address such issues?*), we received responses from 6 participants. Remarkably, every single one of these respondents expressed the need for improved documentation and requested the availability of automated tools, similar to our pipeline, to aid in the detection of these issues.

The findings from our collected data reveal that the problem is not only widespread but also commonly misinterpreted by app developers, e.g., "*We were not aware that manual privacy protection implementations were required on Android with the current version*". Further, security and privacy problems frequently stem from online app generators [52], e.g., "*I create apps using an online app development website that allows users to create apps without programming*".

6 Dicussion

Our findings show the widespread absence of web permission enforcement in WebView, despite existing web standards for privacy protection. We now explore the related challenges.

6.1 Privacy and Legal Implications

By default, Android's WebView permits app developers to handle web permission requests on their own. Our findings suggest that in real-world settings, apps that utilize Android's WebView and fail to comply with the privacy protection guidelines outlined in Web Permissions [64] are vulnerable to potential exploitation by malicious actors. These PHAs can load websites that access sensitive data like location, camera, and microphone without the user's knowledge, leading to significant privacy concerns. Such websites can secretly gather user data during browsing, and malicious apps could misuse this to collect information without user consent. Often, users grant broad permissions to host apps, unaware that this action allows every website they visit through the app to access their sensitive data. More importantly, this practice could also result in GDPR noncompliance, particularly with Article 25, which requires sufficient technical and organizational measures for information security. Recently, many cases involving fines due to insufficient technical and organizational measures to ensure information security [61, 69, 70].

For displaying third-party websites, Google suggests using Custom Tabs (CTs) unless developers have specific requirements [5], e.g., interpreting the loaded content. CTs in Android, as opposed to WebView, provide an enhanced user experience, strengthened security, and improved privacy. This is because Custom Tabs are powered directly by the user's preferred browser, which inherits the browser's standard features. As such, developers do not need to write custom code to handle permission requests or permission grants. Surprisingly, the adoption of CTs is relatively low, within our dataset, we identified only 10,307 apps that utilized CTs.

6.2 App Generators and Discussion Platforms

Feedback from developers has also highlighted concerns with online app generators (i.e., 7 responses indicating app development via such platforms, for example "*I create apps using an online app development website that allows users to create apps without programming*"). These tools, designed to ease app development, deployment, and maintenance, offer cost-effective solutions by optimizing different phases of the app lifecycle. They enable simplified app creation through drag-and-drop modules and offer limited customization options. Oltrogge et al. [52] have shown that this convenience comes with security vulnerabilities that need to be addressed to safeguard both end-users and online services. Our study highlights another inherent privacy issue in these platforms.

Further, platforms such as Stack Overflow, well-known for online programming discussions, are valuable to software developers, offering a rich source of dynamic discussions and ready-to-use code snippets. Previous research has highlighted Stack Overflow as an essential tool for developers, who often copy and paste code snippets from these platforms into their projects. This includes instances where insecure code snippets are transferred into Android apps [24, 32]. To understand the impact of this copy-paste behavior on web permission enforcement in Android's WebView, we conducted a targeted search for relevant questions and their answers on these platforms. Specifically, we searched for questions tagged with *[android] [webview]* that contained the word "*permission*" and had at least one answer. This led us to 217 relevant questions. Notably, among these, we identified 19.3% answers that suggest potentially privacy-harmful behaviors, such as granting permissions without proper enforcement.

6.3 Recommendations

Based on our findings, we provide recommendations for enhancing privacy and ensuring compliance when using Android's WebView. As such, developers can protect the rights of data subjects and be transparent about how data is collected and used.

- **Implement Explicit Permission Requests**: Ensure that any access to sensitive data like location, camera, or microphone through WebView prompts the user for explicit consent, clearly explaining the purpose of the request (e.g., see Listing 5 in Appendix).
- **Restrict Permissions to Trusted Content**: Limit WebView to load content only from trusted sources. Implement validation checks to prevent loading of unverified or potentially malicious websites that could exploit granted permissions.
- **Transition to Custom Tabs (CTs)**. For use cases that don't require a highly customized web experience or only involve displaying external web content, we recommend developers transition to CTs. CTs are powered by the user's preferred browser, adhering to web standards for permissions [64]. This eliminates the need for custom code to handle permissions, simplifies implementation, and enhances user privacy.

7 Conclusion

In this study, we filled a critical research gap by conducting an extensive analysis of how Android apps implement WebView for web permission enforcement. We introduced an automated pipeline to identify privacy-harmful apps (PHAs). In our analysis of 276,760 Android apps in Google Play, we identify 54,605 apps that instantiate WebView and enable JavaScript execution within their WebView settings. By searching for bytecode-level signatures of misconfigured implementations in web permission enforcement within WebView, our pipeline detects 12,109 potential PHAs (out of 54,605 apps) that compromise user-sensitive data due to a failure to implement web permission enforcement. Further, we demonstrated how 2,219 PHAs could be exploited to access sensitive user data without user awareness. This evidence reveals substantial privacy risks, enabling widespread data collection by websites and malicious apps. Proactively reaching out to affected developers, we uncovered common issues and misconceptions. Collaboration between developers and platform providers is important for enhancing user protection and privacy in app development.

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the usability of cryptographic apis. In *SP*.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *MSR*.

[3] androguard. 2023. Androguard. https://github.com/androguard/androguardurl Accessed: 2023-11-20.

[4] Android. 2023. App manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro Accessed: 2023-09-18.

[5] Android. 2023. Custom Tabs. https://developer.chrome.com/docs/android/custom-tabs/ Accessed: 2023-11-29.

[6] Android. 2023. requestPermissions. https://developer.android.com/reference/androidx/core/app/ActivityCompat#requestPermissions(android.app.Activity, java.lang.String[],int) Accessed: 2023-11-20.

[7] Android. 2023. WebView. https://developer.android.com/reference/android/webkit/WebView Accessed: 2023-09-18.

[8] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *CCS*.

[9] Philipp Beer, Lorenzo Veronese, Marco Squarcina, and Martina Lindorfer. 2022. The Bridge between Web Applications and Mobile Platforms is Still Broken. In *SecWeb*.

[10] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. 2017. Exploring decision making with Android's runtime permission dialogs using in-context surveys. In *SOUPS*.

[11] Weicheng Cao, Chunqiu Xia, Sai Teja Peddinti, David Lie, Nina Taft, and Lisa M Austin. 2021. A large scale study of user behavior, expectations and engagement with Android permissions. In *USENIX Security*.

[12] Orcun Cetin, Carlos Ganan, Maciej Korczynski, and Michel van Eeten. 2017. Make notifications great again: learning how to notify in the age of large-scale vulnerability scanning. In *WEIS*.

[13] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. 2012. Is this app safe? A large scale study on application permissions and risk signals. In *WWW*.

[14] Erika Chin and David Wagner. 2013. Bifocals: Analyzing webview vulnerabilities in android applications. In *WISA*.

[15] chromium. 2023. permission.site. https://github.com/chromium/permission.site Accessed: 2023-11-20.

[16] Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 2020. When tls meets proxy on mobile. In *ACNS*.

[17] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on android. In *CCS*.

[18] developer.mozilla.org. 2023. Permissions API. https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API Accessed: 2023-09-18.

[19] Yusra Elbitar, Michael Schilling, Trung Tin Nguyen, Michael Backes, and Sven Bugiel. 2021. Explanation beats context: The effect of timing & rationales on users' runtime permission decisions. In *USENIX Security*.

[20] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. 2011. A study of android application security.. In *USENIX Security*.

[21] explodingtopics.com. 2023. Internet Traffic from Mobile Devices (Oct 2023). https://explodingtopics.com/blog/mobile-internet-traffic Accessed: 2023-10-05.

[22] AP Felt, E Ha, S Egelman, A Haney, E Chin, and D Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *SOUPS*.

[23] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *CCS*.

[24] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *SP*.

[25] forbrukerradet.no. 2023. OUT OF CONTROL. https://fil.forbrukerradet.no/wp-content/uploads/2020/01/2020-01-14-out-of-control-final-version.pdf Accessed: 2023-10-12.

[26] Frida. 2024. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/ Accessed: 2024-12-16.

[27] Xing Gao, Dachuan Liu, Haining Wang, and Kun Sun. 2015. Pmdroid: Permission supervision for android advertising. In *SRDS*.

[28] Google. 2023. Permissions on Android. https://developer.android.com/guide/topics/permissions/overview Accessed: 2023-10-12.

[29] Google. 2023. What's new in Android Privacy. https://android-developers.googleblog.com/2021/05/android-security-and-privacy-recap.html Accessed: 2023-09-18.

[30] A Gorla, I Tavecchia, F Gross, and A Zeller. 2014. Checking App Behavior Against App Descriptions. In *ICSE*.

[31] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A tale of two cities: How webview induces bugs to android applications. In *ASE*.

[32] Alfusainey Jallow, Michael Schilling, Michael Backes, and Sven Bugiel. 2024. Measuring the Effects of Stack Overflow Code Snippet Evolution on Open-Source Software Security. In *SP*.

[33] PG Kelley, S Consolvo, LF Cranor, J Jung, N Sadeh, and D Wetherall. 2012. A Conundrum of Permissions: Installing Applications on an Android Smartphone. In *FC*.

[34] Brian Kondracki, Assel Aliyeva, Manuel Egele, Jason Polakis, and Nick Nikiforakis. 2020. Meddling middlemen: Empirical analysis of the risks of data-saving mobile browsers. In *SP*.

[35] Douglas J Leith. 2021. Web Browser Privacy: What Do Browsers Say When They Phone Home? *IEEE Access* (2021).

[36] Xu Lin, Panagiotis Ilia, and Jason Polakis. 2020. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *CCS*.

[37] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun Aerin Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. 2016. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *SOUPS*.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *CCS*.

[39] Meng Luo, Bo Feng, Long Lu, Engin Kirda, and Kui Ren. 2023. On the complexity of the Web's PKI: Evaluating certificate validation of mobile browsers. *TDSC* (2023).

[40] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. 2019. Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers. In *NDSS*.

[41] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. 2017. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *CCS*.

[42] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *ACSAC*.

[43] Max Maass, Marc-Pascal Clement, and Matthias Hollick. 2021. Snail mail beats email any day: on effective operator security notifications in the internet. In *ARES*.

[44] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. 2011. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems.* Technical Report. ETH Zurich.

[45] C Matte, N Bielova, and C Santos. 2020. Do Cookie Banners Respect my Choice?: Measuring Legal Compliance of Banners from IAB Europe's Transparency and Consent Framework. In *SP*.

[46] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A large-scale study of mobile web app security. In *MoST*.

[47] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. 2013. A View to a Kill:WebView Exploitation. In *LEET*.

[48] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. 2021. Share First, Ask Later (or Never?) Studying Violations of GDPR's Explicit Consent in Android Apps. In *USENIX Security*.

[49] Trung Tin Nguyen, Michael Backes, and Ben Stock. 2022. Freely given consent? Studying consent notice of third-party tracking and its violations of GDPR in android apps. In *CCS*.

[50] Trung Tin Nguyen, Duc Cuong Nguyen, Michael Schilling, Gang Wang, and Michael Backes. 2020. Measuring User Perception for Detecting Unexpected Access to Sensitive Resource in Mobile Apps. In *ASIA CCS*.

[51] Kazuki Nomoto, Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. 2023. Browser Permission Mechanisms Demystified.. In *NDSS*.

[52] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. 2018. The rise of the citizen developer: Assessing the security impact of online app generators. In *SP*.

[53] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. 2021. Why Eve and Mallory Still Love Android: Revisiting TLS (In) Security in Android Applications. In *USENIX Security*.

[54] Amogh Pradeep, Álvaro Feal, Julien Gamba, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, David Choffnes, et al. 2023. Not Your Average App: A Large-scale Privacy Analysis of Android Browsers. In *PETS*.

[55] Yiting Qu, Suguo Du, Shaofeng Li, Yan Meng, Le Zhang, and Haojin Zhu. 2020. Automatic permission optimization framework for privacy enhancement of mobile applications. *IEEE IoT Journal* (2020).

[56] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *CCS*.

[57] Lena Reinfelder, Andrea Schankin, Sophie Russ, and Zinaida Benenson. 2018. An inquiry into perception and usage of smartphone permission models. In *TrustBus*.

[58] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android.. In *NDSS*.

[59] Soot. 2023. Soot. https://github.com/soot-oss/soot Accessed: 2023-09-18.

[60] B Stock, G Pellegrino, C Rossow, M Johns, and M Backes. 2016. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *USENIX Security*.

[61] techcrunch.com. 2023. Irish Supervisory Authority announces decision in Facebook Data Scraping inquiry. https://techcrunch.com/2022/11/28/facebook-gdpr-penalty/ Accessed: 2023-09-18.

[62] Vasant Tendulkar and William Enck. 2014. An application package configuration approach to mitigating android ssl vulnerabilities. *MoST* (2014).

[63] Güliz Seray Tuncay, Jingyu Qian, and Carl A Gunter. 2020. See no evil: phishing for permissions with false transparency. In *USENIX Security*.
[64] w3c.github.io. 2023. Web-based content. https://w3c.github.io/permissions/ Accessed: 2023-09-18.
[65] Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, and Tatsuya Mori. 2015. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *SOUPS*.
[66] C Weir, B Hermann, and S Fahl. 2020. From Needs to Actions to Secure Apps? The Effect of Requirements and Developer Practices on App Security. In *USENIX Security*.
[67] Daoyuan Wu and Rocky KC Chang. 2014. Analyzing android browser apps for file://vulnerabilities. In *ISC*.
[68] www.appbrain.com. 2023. Android and Google Play statistics. https://www.appbrain.com/stats Accessed: 2023-11-20.
[69] www.dpa.gr. 2023. Cosmote Mobile Telecommunications S.A. https://www.dpa.gr/sites/default/files/2022-01/4_2022%20anonym%20%282%29_0.pdf Accessed: 2023-09-18.
[70] www.uodo.gov.pl. 2023. Fortum Marketing and Sales Polska S.A. https://www.uodo.gov.pl/pl/138/2389
[71] Guangliang Yang, Jeff Huang, and Guofei Gu. 2019. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *USENIX Security*.
[72] Lei Zhang, Zhibo Zhang, Ancong Liu, Yinzhi Cao, Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang. 2022. Identity confusion in WebView-based mobile app-in-app ecosystems. In *USENIX Security*.
[73] Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen. 2016. Rethinking permission enforcement mechanism on mobile systems. *IEEE Trans. Inf. Forensics Secur.* (2016).
[74] Zicheng Zhang. 2021. On the usability (in) security of in-app browsing interfaces in mobile apps. In *RAID*.
[75] Zhang Zicheng, Ma Haoyu, Wu Daoyuan, Gao Debin, Yi Xiao, Chen Yufan, Wu Yan, and Jiang Lingxiao. 2024. MtdScout: Complementing the Identification of Insecure Methods in Android Apps via Source-to-Bytecode Signature Generation and Tree-based Layered Search. In *EuroS&P*.

```
@Override
public void onPermissionRequest(final PermissionRequest request) {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
            == PackageManager.PERMISSION_GRANTED) {
        String url = request.getOrigin().toString();
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setTitle("Confirmation");
        builder.setMessage("Allow" + url + "to use your camera and
        ↪ microphone?");

        // Add Yes button
        builder.setPositiveButton("Yes", new
        ↪ DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                request.grant(request.getResources());
                dialog.dismiss(); // Close the dialog
            }
        });

        // Add No button
        builder.setNegativeButton("No", new
        ↪ DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                // Action for No button
                request.deny();
                dialog.dismiss(); // Close the dialog
            }
        });

        // Create and show the dialog
        AlertDialog alertDialog = builder.create();
        alertDialog.show();

        request.grant(request.getResources());
    } else {
        request.deny();
    }
}
```

Listing 5: The WebView notifies users of camera access requests and lets users decide via a custom prompt.