Tabbed Out: Subverting the Android Custom Tab Security Model

Philipp Beer, Marco Squarcina, Lorenzo Veronese and Martina Lindorfer
TU Wien

***Abstract*—Mobile operating systems provide developers with various mobile-to-Web bridges to display Web pages inside native applications. A recently introduced component called Custom Tab (CT) provides an outstanding feature to overcome the usability limitations of traditional WebViews: it shares the state with the underlying browser. Similar to traditional WebViews, it can also keep the host application informed about ongoing Web navigations. In this paper, we perform the first systematic security evaluation of the CT component and show how the design of its security model did not consider cross-context state inference attacks when the feature was introduced. Additionally, we show how CTs can be exploited for fine-grained exfiltration of sensitive user browsing data, violation of Web session integrity by circumventing SameSite cookies, and how UI customization of the CT component can lead to phishing and information leakage. To assess the prevalence of CTs in the wild and the practicality of the mitigation strategies we propose, we carry out the first large-scale analysis of CT usage on over 50K Android applications. Our analysis reveals that their usage is widespread, with 83% of applications embedding CTs either directly or as part of a library.**

We have responsibly disclosed all our findings to Google, which has already taken steps to apply targeted mitigations, assigned three CVEs for the discovered vulnerabilities, and awarded us $10,000 in bounties. Our interaction with Google led to clarifications of the CT security model in the new Chrome Custom Tabs Security FAQ document.

1. Introduction

Mobile operating systems offer a variety of mobile-to-Web bridges that developers can use to integrate Web content into their native applications. Depending on a combination of factors, e.g., the operating system (OS, either Android or iOS), and the installed browser, these components have different sets of capabilities and privileges [1]. Developers can provide further customization to match the application's theme and minimize the visible disruption between the native application and the Web content, making it virtually indistinguishable from the host application. This feature is particularly attractive for social network and messaging applications that allow users to open arbitrary URLs directly within the host application. This can lead to security and privacy issues, e.g., when these *in-app browsing* mechanisms do not present users with adequate security indicators [2].

Yet, even when using standard components provided by the OS, bridging mobile native applications and Web content can have unforeseen consequences. Security risks previously unknown to mobile applications can become a threat when these components are used, as extensive research on the Android WebView component has demonstrated [3], [4], [5], [6], [7], [8]. Furthermore, new attack vectors are emerging as novel mechanisms and APIs are introduced to mobile platforms [9]. A widely used yet under-explored mechanism is the *Custom Tab* component, which we focus on in this paper. Custom Tabs (CTs) provide applications with a seamless way to implement in-app browsing but also come with two interesting features from a security and privacy perspective: they *share state with the underlying browser*, such as Chrome, other Chromium-based browsers including Edge and Brave, as well as Firefox, and provide *navigation awareness* to the host application through callbacks. These two features open the possibility for a new class of attacks that we call *Cross-Context Leaks*. Similarly to Cross-Site Leaks (XS-Leaks), which share information across different sites and represent a strict Web security vulnerability, Cross-Context Leaks can disclose sensitive information across the Web and the mobile context, i.e., between the website and the host application of the Custom Tab.

We perform the first systematic security evaluation of the Custom Tab component, assuming a potentially unwanted application (PUA) to be installed on the user's device, and show how Cross-Context Leaks, which were not considered in enough detail when the Custom Tab feature was launched, can be used to infer sensitive user data from websites, thus subverting the current security model of Custom Tabs. We present attacks that can be used to infer coarse-grained information about a user, such as whether the user is logged in on a specific website, but also more fine-grained information that allows to infer the user's location history. Furthermore, we demonstrate how HTTP request headers can be injected into requests initiated by a Custom Tab and how SameSite `Strict` cookies can be bypassed. To enable these and further attacks in a stealthy manner, we also demonstrate how an attacker can fully hide the Custom Tab browser activity and the Web content displayed within it. Additionally, we illustrate how the customization feature of the Custom Tab's interface enables phishing attacks.

We present possible mitigation strategies for the attacks and responsibly disclosed all vulnerabilities to Google through the Chrome Vulnerability Reward Program. The Chrome security team assigned three CVEs to the discovered vulnerabilities and awarded us $10,000 for our reports. We are further engaging with Google to discuss broader mitigations that address core issues of the Custom Tab security model. In particular, we propose to restrict state sharing with the underlying browser while preserving compatibility and offer websites the possibility to opt out of being loaded in a Custom Tab via standard Web security mechanisms.

To underscore the need for implementing these strategies and understand whether they break existing functionality, we analyze the usage of CTs in the most popular applications in the Google Play Store. We find that their use is widespread: the vast majority of applications (83%) integrate CTs either as part of the main application or through third-party libraries that provide security-critical functionality, such as authentication, but also frequently for advertisement purposes.

In summary, we make the following contributions:

- We present six new attacks using CTs that allow potentially unwanted applications to stealthily leak user browsing data, perform authenticated requests (with custom headers) on behalf of the user, bypass SameSite cookies, and carry out phishing attacks (Sec. 3).
- We propose mitigation strategies for the discovered attacks and discuss possible extensions to Web security mechanisms, e.g., CSP and Fetch Metadata, to allow specifying cross-context embedding restrictions (Sec. 4).
- We perform the first large-scale measurement of the usage of Custom Tabs on over 50K apps with more than 1M downloads, studying the most used libraries and characterizing their usage of the Custom Tab API. We discover that 83% of the analyzed apps use the component, and the majority of libraries that include CTs use them for authentication (e.g., OAuth 2.0) or advertising purposes (Sec. 5).
- We present five case studies of cross-context information leakage on high-profile websites to demonstrate the real-world impact of the discovered attacks (Sec. 6).

The artifacts of this work, including the proof of concepts for the attacks, the source code of our analysis pipeline, and the measurement results are available at purl.org/ct-paper.

2. Background

The traditional way of opening websites is by loading them in a standalone browser. However, on mobile platforms, this approach has usability and functionality limitations since users are forced to leave the application and switch to the browser. Developers can thus embed Web content in their native applications using WebView components, providing a seamless user experience. Starting from 2015, Android and iOS added support for Custom Tabs (CTs) [10] and `SFSafariViewController` [11], respectively, which are browser components that can be embedded in native applications to provide the same browsing experience as the full browser for in-app browsing. These components allow apps to launch a browser view for interacting with websites without leaving the application. In this section, we focus on Android CTs, being instrumental for the attacks presented in this paper. We also compare CTs to other components for embedding Web content on Android and iOS. Table 1 provides an overview of features supported by different browsers as of April 2023.

Feature	v.112	v.112	v.112	v.1.50	v.16.4
State sharing	●	●	●	●	○
Navigation callbacks	●	–	●	●	–
Scroll callbacks	●	–	●	●	–
Bottom bar	●	–	●	●	–
Adding approvelisted headers	●	–	●	●	–

TABLE 1: Support of selected CT features on Android (Chrome, Firefox, Edge, Brave) and SFSafariViewController (Safari) on iOS (● supported, ○ restricted, – unsupported).

2.1. State Sharing

Compared to other components for embedding Web content, such as `WebView` [12] on Android, or `WKWebView` [13] and `SFSafariViewController` on iOS, CTs offer a unique capability: they *share state with the underlying browser*, including cookies and permissions. Consequently, when users log into a website in the browser, they are also implicitly authenticated in the CT. However, unlike the `WebView` and `WKWebView` components, CTs do not allow for injecting JavaScript code into the website nor offer a built-in bridge to expose native functions of the app to the website. Nevertheless, the website and the native app can set up a `postMessage`-based channel that can be used for bidirectional communication, and apps can listen to website events via the `CustomTabsCallback` [14] class, as discussed next. Additionally, it is possible for an app to send browser vendor-specific commands to the CT via the `extraCommand` [15] function.

2.2. Navigation Awareness through Callbacks

Native applications can track interactions with websites by registering callbacks. Various types of callbacks are supported by CTs, such as *navigation callbacks* that are triggered during Web navigation events and *extra callbacks* that are specific to the browser implementation. Developers can receive and utilize these events in the host application by overriding the `CustomTabsCallback` class. Navigation callbacks are fired at various stages of the loading process or when a CT changes its visibility state. More specifically, the callback mechanism distinguishes between six types of navigation events [14]:

- `NAVIGATION_STARTED` when the page starts loading,
- `NAVIGATION_FAILED` when the page fails loading,
- `NAVIGATION_ABORTED` when a user cancels the loading,
- `NAVIGATION_FINISHED` when the page finishes loading,
- `TAB_SHOWN` when the browser becomes visible, and
- `TAB_HIDDEN` when the browser becomes hidden.

We discuss details about other types of callbacks, such as extra callbacks, in the following sections.

2.3. UI Customization and Other Features

CTs provide developers with the ability to tailor the style of the browser activity to suit the application's look. This includes the ability to modify the color of the URL bar, apply enter and exit animations, and add custom actions to specific buttons in the browser's navigation bar. Application developers can also create a *bottom bar*, a secondary toolbar at the bottom of the screen, and fully customize its appearance.

Chrome, Edge, and Brave CTs enable the inclusion of CORS-approvelisted HTTP request headers (including `accept`, `accept-language`, `content-language`, and `content-type`, see Table 6 for the complete list) when performing the first request to a website. In the presence of a trust relationship between the host application and the website, it is also possible to include HTTP request headers that are not CORS-approvelisted [16]. *Digital Asset Links* (DALs) are used to establish this mutual trust relationship. Websites need to place a *statement list* file at a specific location under the `/.well-known/` path, which includes the package names of the trusted applications for the website and the fingerprints of applications' signing certificates [17].

Compared to the `WebView` component, CTs also offer some performance optimizations. It is possible to pre-initialize the browser for a seamless integration with the application by calling the `warmup` [15] method. Developers can also provide the browser with a list of URLs that are likely to be visited using the `mayLaunchUrl` [18] method so that the browser can perform speculative work such as initializing the connection and pre-rendering Web pages.

2.4. Activities and Intents in Android

Activities are fundamental building blocks of Android applications and provide the visual component for user interaction. Activities are typically displayed full-screen and organized in a stack, where only the topmost activity is visible. An application can consist of multiple activities, where one activity can start another one by using *Intents*.

Intents are used for inter-component communication within the same application and with other applications and services. Intents can be implicit or explicit, depending on whether the application specifies a target activity that should handle it. For example, to open a URL in a browser, an application can issue an implicit Intent and can optionally specify the browser's package name, e.g., `org.mozilla.firefox`. Intents can also include arbitrary extra information as *Bundles*, i.e., lists of key/value pairs. To request the browser to open the website in a CT, the application must add the `android.support.customtabs.extra.SESSION` string to the Bundle of the Intent.

3. Subverting Custom Tabs

This section presents the first comprehensive security evaluation of the Android CT component. We first introduce the underlying threat model and discuss the core objectives of an attacker. We then describe our employed methodology. To make our attacks stealthy, we present two gadgets that enable an application to hide a CT activity or the Web content within a CT. Finally, we present six novel attacks that enable applications to violate the security and privacy of Web users. Table 2 summarizes the gadgets, attacks, and affected browsers.

3.1. Threat Model

We consider the classic *app attacker model* [6], i.e., attacks launched by a *potentially unwanted application* (PUA) installed on the user's Android device. The application only needs the permission to access the Internet. This permission is considered a *normal permission* on Android, meaning that it is already granted at install time and does not require run-time checks [19]. We also assume that the application is actively used by the user, i.e., an activity of the application needs to be in focus. The application can be a privacy-invasive app that either compromises the *confidentiality* or the *integrity* of user data and uses *hiding gadgets* to make the attack stealthy. When compromising confidentiality, a PUA wants to infer information about the user's browsing behavior, i.e., coarse-grained information, such as whether a user is logged in on a specific website, or more fine-grained information, such as the identity of a user. The attacker's objective in compromising integrity is to manipulate the user's browsing context, such as by swapping the user's existing sessions on websites or carrying out unauthorized actions. Additionally, the application can disguise itself as a legitimate and useful app to lure users into installing it. It is also plausible that an attacker operates a software development kit (SDK) that, when integrated as a library in a benign application, turns it rogue [20], [21].

3.2. Methodology

In order to study the security of CTs and uncover novel attack vectors, we employed the following methodology: First, we studied the available CT documentation to identify any potential design flaws in the component. Because this step only allowed us to cover documented functionality, we manually reviewed the Chromium source code to find yet undocumented APIs and also to detect discrepancies between the implementation and the documentation. This was then followed by manual API testing, e.g., mutating API parameters and triggering edge cases, which was instrumental in verifying our assumptions about the implementation. We then matched the observed behavior against Web standards to find potential violations and gray areas. When discovering a new attack, we implemented proof of concepts to validate our claims, performed additional testing to fully understand its capabilities and impact, reported the vulnerability, and engaged with the affected vendors to identify compatible mitigation strategies. We will detail these additional tests in the corresponding sections. Table 3 shows how each step of our methodology contributed to identifying the gadgets and attacks proposed in this paper.

3.3. Custom Tab Hiding Gadgets

When a CT is opened by an application via the `launchUrl` method, a CT activity is launched. The activity

	Gadget/Attack	v.112	v.112	v.112	v.1.50	Stealth	Affects	Reason
Gadget	CT Activity Hiding (⊗)	●	-	●	●	-	-	CT activity overlain with another activity
	Web Content Hiding (⊘)	●	-	●	●	-	-	Bug in height restriction of bottom bar
Attack	State Inference Attack	●	-	●	●	⊗	C	Reporting of navigation callbacks (by design)
	HTTP Header Injection	●<v.108	-	●<v.108	●<v.1.46	⊗	I	Improper sanitization of HTTP header values
	SameSite Cookie Bypass	●<v.109	-	●<v.110	●<v.1.48	⊗	I	Sending SameSite `Strict` cookies (by design)
	Scroll Inference Attack	●	-	●	-	⊘	C	Reporting of scroll callbacks (by design)
	Bottom Bar Info Leakage	●	-	●	●	⊘	C	URL in bottom bar Intent (by design)
	Bottom Bar Phishing	●	-	●	●	-	C	Existence of bottom bar (by design)

TABLE 2: CT gadgets, attacks, and vulnerable browsers on Android (● vulnerable, ⊗ can be combined with CT Activity Hiding gadget, ⊘ can be combined with Web Content Hiding gadget, C confidentiality, I integrity). Note that Firefox supports CTs but it is not vulnerable to any of our attacks.

Methodology Step	Identified Attacks/Gadgets
Review of documentation	CT Activity Hiding gadget Cross-Context State Inference Attack Bottom Bar Info Leakage Bottom Bar Phishing
Review of source code	Scroll Inference Attack
Manual API testing	Web Content Hiding gadget HTTP Header Injection
Cross-check of Web standards	SameSite Strict Cookie Bypass

TABLE 3: Identified attacks/gadgets by methodology step.

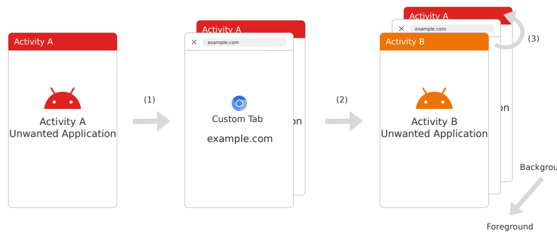


Figure 1: Using the CT Activity Hiding gadget to hide the CT by overlaying it with another activity.

is launched either full-screen or, more recently, as a bottom sheet in the lower part of the screen, but cannot be started in the background or closed without explicit user interaction. In this section, we therefore present two techniques that enable an application to hide a CT activity or the Web content within a CT. While these techniques may not pose severe threats to the user's security and privacy when viewed in isolation, they can be used as gadgets to perform other attacks stealthily, thus acting as enablers for more impactful attacks.

3.3.1. Custom Tab Activity Hiding. To fully hide a CT, an attacker can use CT callbacks. CTs offer the `CustomTabsCallback` class and the `onNavigationEvent` function to notify the launching application about Web navigation events. The `TAB_SHOWN` event is fired when the CT becomes visible. Attackers can stealthily launch CTs as shown in Fig. 1: (1) Activity A of the unwanted application launches the CT with the target website. (2) Activity A listens to the `TAB_SHOWN` event of the CT and launches activity B as soon as the event is received. (3) At this point, even if activity B is in the foreground, future callback events are still reported to activity A. One aspect that needs to be considered is the activity back stack. The back stack stores the activities according to the order in which they are opened. If the user presses the back button or performs the back gesture, the activity at the top of the stack is removed, and the CT is displayed [22]. To overcome this shortcoming, applications can overwrite the `onBackPressed` function in the overlaying activity. By doing so, the previous activity A can be relaunched instead of bringing the CT into the foreground.

Another way to hide the CT is to overlay it with the same activity that launched it. That is, activity A launches a CT that is then overlaid by activity A. This technique can, for instance, be employed during video playback. While the user is watching a video in activity A, a CT is launched but immediately hidden by activity A, providing the user with an uninterrupted video experience. Note that while the video is playing, UI controls are unresponsive for a short time during the CT launch.

Multiple CTs can be opened in a single run, i.e., in one activity transition. This can be achieved by launching CTs in sequence, each triggered by the `TAB_SHOWN` event of the preceding CT, and opening the overlay activity when the `TAB_SHOWN` event of the final CT is fired. Note that the overlay activity is visible only after all CTs are opened and, as with the previous method, the UI becomes unresponsive during the attack.

Performance. We evaluate the stealthiness of the CT Activity Hiding gadget, thus the impact of such unresponsiveness, by measuring the performance of the gadget. In particular, we launch multiple CTs in parallel and measure the time it takes from the beginning of the attack until the UI is responsive again. We test the technique on a Google Pixel 6a running Android 13 and Chrome 112, repeating the experiment 50 times. The results are provided in Fig. 2. Opening a single CT leaves, on average, the UI unresponsive for 0.2s, while launching 10 parallel CTs takes about 1.7s. Note that up

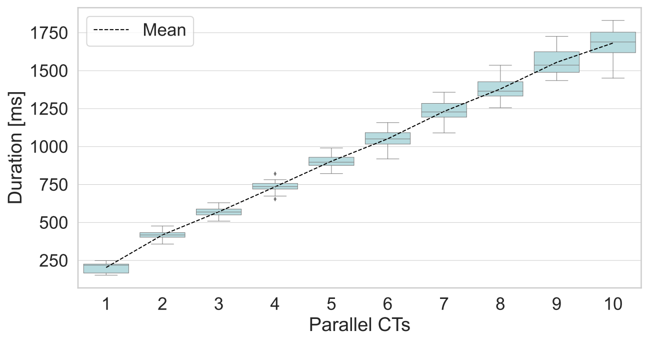


Figure 2: Execution duration of the Activity Hiding gadget ($n = 50$). The UI is unresponsive during this time period.

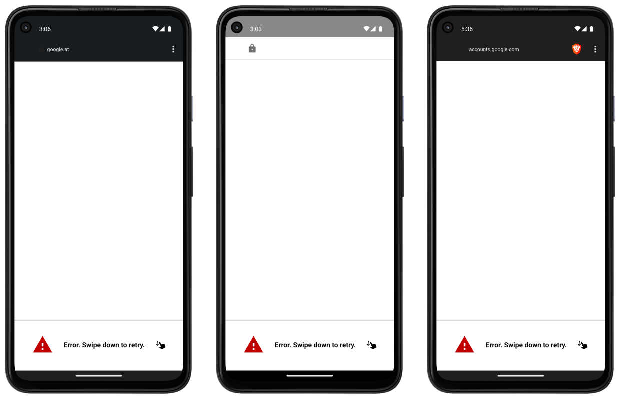
to 5 CTs can be opened within one second, thus allowing to probe 5 websites within a delay that would not raise suspicion among most users.

3.3.2. Web Content Hiding. To hide the Web content rendered in a CT, an application can take advantage of the bottom bar. Although the maximum height of the bottom bar is limited, we discovered that embedding a larger element into the bar can hide the website's content. This technique does not make it possible to include custom elements outside the bottom bar area, but it creates an overlay of the same color as the website's background over the Web content. Notice that this overlay does not prevent the website from being scrolled.

In addition to hiding the Web content, an application can alter the information displayed in the navigation bar of the CT. The share button can be removed by calling the `setShareState` method with the `SHARE_STATE_OFF` constant, and the close button can be hidden by setting a transparent icon via the `setCloseButtonIcon` method. Moreover, on Chrome and Brave, changing the navigation bar color via the `setToolbarColor` method so that it matches the color of the padlock icon results in effectively hiding the icon. Surprisingly, on Edge, explicitly assigning a color to the navigation bar causes the hostname to disappear, leaving only the lock icon visible. This technique is shown in Fig. 3, where we additionally spoofed the content of the bottom bar to look like part of the website.

3.4. Cross-Context State Inference Attack

CTs share state with the underlying browser, including cookies, Service Workers, caches, etc. As a result, if a user is authenticated on a website in the browser, they are also authenticated in a CT opened by the same browser. Additionally, CTs provide a callback mechanism that keeps the launching application updated on navigation events, such as when a navigation starts, fails, finishes, or is canceled. We refer the reader to Sec. 2.2 for a comprehensive list of events. This combination of shared state and callbacks can be exploited by an application to infer sensitive user information, thus compromising the confidentiality of user



google.at
Error. Swipe down to retry.
Error. Swipe down to retry.
accounts.google.com
Error. Swipe down to retry.

(a) Chrome 112 (b) Edge 112 (c) Brave 1.50

Figure 3: Web Content Hiding using the bottom bar. It contains custom UI elements to draw the user's attention.

data. By opening a target website in a CT and listening to these callbacks, an attacker can infer the user's state on the website based on the sequence, timing, and type of the fired navigation callbacks. This attack is comparable to an emerging class of attacks called cross-site leaks (XS-Leaks) [23], which uses a cross-site oracle to distinguish between different states on a website and to leak user information. In contrast to XS-Leaks, the callback mechanism serves as an oracle to leak information between the mobile and the Web context, making it a *cross-context leak*.

3.4.1. Attack Vectors. By experimentally testing combinations of HTTP status codes and headers[1], we have identified five attack vectors to infer user information on a target website. For all the techniques, an application launches the target website in a CT and analyzes the callback events. Although this section provides examples of information leakage, such as inferring a user's authentication status or whether they have previously visited a website, it is important to note that the attack is not restricted to these scenarios. Depending on the website, the attack can be used to infer any state that causes a different sequence of navigation events or timing, including the presence of a specific item in a shopping cart, whether the user is logged in with a specific user account, or reconstructing the social graph of a user.

Status Code. The status code-based attack is enabled by how CTs handle HTTP response status codes. Our experimental evaluation showed that an HTTP response with status code `4xx/5xx` and an empty response body triggers a `NAVIGATION_FAILED` event directly followed by the `NAVIGATION_FINISHED` event. On the other hand, `2xx/3xx` status codes, irrespectively of the response body, or `4xx/5xx` with a non-empty response body, only fire the `NAVIGATION_FINISHED` event. The conditional presence of the additional `NAVIGATION_FAILED` event can be used to

1. We tested combinations of status codes (200, 3xx, 4/5xx), response bodies (empty, non-empty, redirection), content-type (video/mp4, audio/mpeg, application/pdf), and content-disposition (inline, attachment).

infer a user's information on a target website. For instance, an attacker can request a resource that is only accessible by authorized users and check for the NAVIGATION_FAILED event. If the event is fired, the user is not authenticated. Since this attack technique only works when an empty response body is transmitted on status code 4xx and 5xx, the real-world impact of this method is significantly reduced.

Redirection. An attacker can also infer user information by checking whether requesting a specific resource triggers a redirection. This resource can be, for instance, the login page of the target website that automatically redirects the user to the home screen when authenticated. Another possibility can be a restricted resource that redirects the user to the login page when not authenticated. This allows an attacker to determine if the user is authenticated on the target website. Websites that redirect users to another page for giving consent to third-party cookies on the first visit also enable an attacker to probe if a user has accepted cookies and, hence, whether they have visited the website before. Our experiments on the CT component showed that the NAVIGATION_STARTED and NAVIGATION_FINISHED events are fired on redirection when HTML or JavaScript redirection is used, i.e., when the page is loaded with a <meta> tag with the http-equiv and content attributes [24], or when window.location is set. Redirections caused by HTTP response headers in the 3xx range only fire the initial NAVIGATION_STARTED and NAVIGATION_FINISHED events. Consequently, if the login page example.com/login redirects to the landing page example.com/home using HTML or JavaScript redirections, two NAVIGATION_STARTED and, respectively, also two NAVIGATION_FINISHED events are fired — one for each page. When HTTP redirection is used, only the initial NAVIGATION_STARTED and NAVIGATION_FINISHED events are fired. Websites employing HTTP redirection are thus not affected by the redirection-based attack vector.

Download. Another way to determine the user's state on a website is by a download-based approach. When a resource that triggers a download is accessed, the CT on Chrome, Edge, and Brave triggers the NAVIGATION_ABORTED, but not the NAVIGATION_FINISHED event. The CT then automatically downloads the resource and displays a silent notification, i.e., the notification does not trigger a pop-up but only results in showing a download icon in the device's status bar. When a resource is launched in the CT, the firing of the NAVIGATION_ABORTED event indicates that the resource was downloaded. If any other event is fired, the resource was not downloaded.

Content Type. The behavior of CT callbacks on Chrome, Edge, and Brave also depends on the Content Type of the resource that is fetched. We experimentally discovered that resources with media type video/mp4, audio/mpeg, and application/pdf trigger the NAVIGATION_ABORTED event, despite being successfully loaded in the CT. On the other hand, images and other Content Types such as text/html do not trigger this event. The presence of the NAVIGATION_ABORTED event can be exploited by an attacker to determine the state of a user on a website whenever a resource with a specific Content Type is only accessible by authenticated users.

Timing. In addition to the attack vectors mentioned above, CTs also leak information through timing-based attacks by measuring the time interval between the NAVIGATION_STARTED and NAVIGATION_FINISHED events. This technique relies on the assumption that the loading time of a specific resource in the CT is dependent on the user's website status. This is a valid assumption for two reasons. First, the website server may need to perform additional computation under an authenticated session, e.g., fetch resources from the database, which increases the loading time [25]. Second, browsers significantly reduce the loading time of pages containing cached assets, making it possible to determine whether a user has already visited a website [26]. Chrome, Edge, and Brave's partitioned cache [27] handles caches on a top-level frame basis. Therefore, cached resources like the image example.com/popular.png that is embedded in a.com/index.html are specific to that website and not reusable when embedded in b.com/index.html. This approach effectively eliminates false positives in which a website with an unvisited status includes resources from visited websites and gets classified as visited. The timing-based attack vector can also be used in the presence of HTTP redirections, depending on the user's state. Even though HTTP redirections cannot be identified by using the redirection-based attack, every redirection adds a round-trip, which affects the loading time.

3.4.2. Stealthiness and Performance.

All attack vectors discussed in this section can be combined with the CT Activity Hiding gadget described in Sec. 3.3.1. Thus, one website can be stealthily probed in 0.2s, while probing 10 websites takes about 1.7s.

Concerning the timing-based attack, an application may need to compare the measured time via CTs against a baseline value when the user is not authenticated or has never visited the target website. Notice that a fixed baseline value is not sufficient due to various factors affecting loading time, such as network speed. To do so, it is possible to load the target website in a hidden WebView in the application, measure the loading time using the onPageLoadStarted and onPageLoadFinished functions in the WebViewClient class and compare it to the loading time in the CT. Since WebViews do not share state with the browser, the user is surely not authenticated in them, nor has the user visited the target website before.

3.4.3. Advantages Over Traditional XS-Leak Attacks.

It is important to stress that opening a website in a CT represents a top-level navigation. Many modern Web security mitigations, such as SameSite cookies, X-Frame-Options [28], the frame-ancestors CSP directive [29], and Fetch Metadata [30] are designed to restrict cross-site and cross-origin interactions. The CT attack vectors we presented, however, are not cross-origin attacks but *cross-context* attacks and can infer user-dependent information by performing top-level navigations. In the following, we discuss distinct advantages of the CT attack compared to traditional XS-Leaks.

Framing Protection (XFO and CSP). Employing response headers that restrict whether a resource can be embedded in an attacker's website, such as `X-Frame-Options` (XFO) (set to `DENY` or `SAMEORIGIN`) and `Content-Security-Policy` (CSP) (`frame-ancestors` directive) headers do not prevent the CT attack. These headers only prevent embedding cross-origin resources in a malicious website, e.g., using the `<iframe>` tag.

Fetch Metadata. The Fetch Metadata HTTP request headers provide the context of the HTTP request to the server. The server can use the context to determine whether the request is legitimate or should be blocked. This allows site operators to deploy a Resource Isolation Policy [30] or a Navigation Isolation Policy [31] to protect critical endpoints from a range of attacks, including Cross-Site Request Forgery (CSRF) [32] and XS-Leaks. The `Sec-Fetch-Site` header informs the server about the relationship between the origin of the request and the target resource. The header supports the `same-origin`, `same-site`, `none`, and `cross-site` values. The `Sec-Fetch-Mode` header specifies the mode of the request, such as whether it is a top-level navigation request, a CORS request, etc. Until Chrome 109, HTTP requests from a CT were indistinguishable from top-level navigation requests with respect to the Fetch Metadata headers, except for the missing `Sec-Fetch-User=?1` header. Thus, Resource Isolation Policies and Navigation Isolation Policies employed on the server could not distinguish malicious requests initiated via CTs from legitimate ones. Starting from Chrome 110, as a result of an independent bug report related to a vulnerability we previously disclosed to Google (see Sec. 4.3), the `Sec-Fetch-Site` header on requests initiated by CTs is set to `cross-site` instead of `none`. We discuss additional protections based on Fetch Metadata in Sec. 4.1.1.

SameSite Cookies. SameSite cookies are a popular mitigation against XS-Leaks as they restrict authenticated requests to same-site navigations. The SameSite attribute can be set to `Strict` or `Lax` to prevent cookies from being attached to cross-site requests [33]. Popular browsers, including Chrome, Edge, and Brave, set the default value to `Lax` if not specified. Our CT attack vectors are not affected by SameSite `Lax` cookies, as they are attached to top-level navigations. We elaborate on additional capabilities of bypassing SameSite cookies, including `Strict` ones, in Sec. 3.6.

3.5. HTTP Header Injection

As mentioned in Sec. 2.3, Chrome, Edge, and Brave CTs allow to add CORS-approvelisted HTTP headers to CT requests in the presence of a DAL. We discovered that Chrome, Edge, and Brave do not properly sanitize the values of the HTTP headers that are added. Thus, applications could add arbitrary HTTP request headers (including non-CORS-approvelisted headers) to requests initiated by CTs, even when the relationship between the website and the unwanted application is not verified, compromising the integrity of the user's browsing session. Attacker-defined HTTP headers can be forced into a CORS-approvelisted HTTP header by setting the value of the permitted header to `"\n<Forbidden-Header>: <value>"`. For instance, a `Cookie` header can be injected by adding the permitted `sec-fetch-ua-full` header to the CT request and setting its value to `\nCookie: secret=cookie`. This vulnerability was fixed in Chrome 109, Edge 109, and Brave 1.47.

3.5.1. Web Security Implications.

Injecting non-CORS-approvelisted HTTP request headers on cross-context requests to third-party origins can have unforeseen consequences. This is primarily due to the fact that CTs share state with the underlying browser. The attack can also be combined with the CT Activity Hiding gadget in Sec. 3.3.1, making it fully stealthy. In the following, we discuss the security issues introduced by this attack.

Session Instantiation and Swapping. An application can set the `Cookie` header to authenticate the victim on a benign website. By setting the session cookie of the attacker in the CT request, the attacker can sign the victim into their account. This is similar to the Login CSRF attack proposed by Barth et al. [34]. User activities are then performed on behalf of the attacker, e.g., the user enters their credit card details on the legitimate website opened in the CT but is signed in with the attacker's account. In this way, the attacker gains knowledge of the user's credit card details. Similarly, the `Authorization` header [35] can be abused for session swapping, as it typically authorizes access to specific resources or authenticates a user on a server, e.g., by using a bearer token [36].

Origin Spoofing. An attacker can add the `Origin` header and spoof another site's origin. This can be used for CSRF attacks on websites that check the Origin of the request for CSRF mitigation, as further described below.

Cross-Site Request Forgery. The non-standard request headers `X-HTTP-Method`, `X-Method-Override`, and `X-HTTP-Method-Override` are used to declare that the request should be treated by the server as if it was issued with a different HTTP method than the original request [37]. For example, a `GET` request containing the `X-HTTP-Method: POST` header is treated as a `POST` request if the server/middleware supports one of these headers. Attackers can issue a `GET` request in a CT and set the method override header to indicate that the `GET` request should be treated as another request method, e.g., as a `POST` or `DELETE` request. An attacker can then make state-changing requests on the victim's behalf, possibly logging users in/out or performing transactions on e-commerce websites, assuming that protection against CSRF attacks is enforced by Origin checking, SameSite cookies, or custom headers.

3.5.2. Limitations.

Not all HTTP request headers can be injected. If the original request already contains certain headers, such as `Cookie`, injecting these headers does not affect their value in the request. The full list of headers that cannot be injected or can only be injected if not already present in the original request can be found in Sec. A. Furthermore, since the size of each extra CORS-approvelisted header is limited to 128 characters, injected headers can only contain at most 127 characters after the newline character `\n` used for the attack.

3.6. SameSite Cookie Bypass

The SameSite cookie attribute can be used to restrict whether a cookie should be attached to cross-site requests and plays a crucial role in preventing cross-site attacks such as CSRF and XS-Leaks. Despite being one of the most impactful security mechanisms to mitigate cross-site integrity and confidentiality abuse, the `Lax` attribute does not prevent cookies from being attached to top-level navigations, such as opening the target website via the JavaScript function `window.open` or via CTs as discussed earlier. Security-critical websites can use the `Strict` value to restrict cookies exclusively to same-site navigations and user-initiated top-level requests, at the expense of compatibility issues with SSO and other common cross-site use cases.

We discovered that the initial request that results from loading a website in a CT attaches SameSite cookies, even if they are marked as `SameSite=Strict`. This poses a significant security risk, even more so in combination with the CT Activity Hiding gadget, as it allows an attacker to silently perform cross-context authenticated requests to potentially critical endpoints. Note that according to a recent study by Khodayari et al. [38], over 10% of websites expose state-changing endpoints to `GET` requests.

Sending SameSite `Strict` cookies in CTs does not strictly violate the cookie standard [33], which does not specify a clear semantics for cross-context interactions:

> *If the "SameSite" attribute's value is "Strict", the cookie will only be sent along with "same-site" requests. [. . .] Same-site cookies in "Strict" enforcement mode will not be sent along with top-level navigations which are triggered from a cross-site document context.*

Although CTs represent a top-level navigation, they are triggered from a third-party context, making them similar to a cross-site document-initiated request and, therefore, untrusted. After disclosing the vulnerability, the issue was fixed in Chrome 110, Edge 111, and Brave 1.49 (see Sec. 4.3).

3.7. Scroll Inference Attack

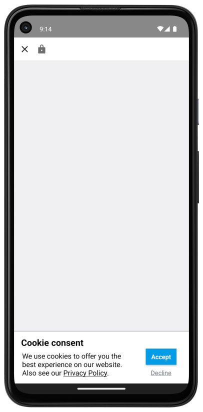
In this section, we describe a more fine-grained cross-context leak compared to the attack in Sec. 3.4. This attack allows us to infer the presence of arbitrary strings on the page of a website opened via CTs, thus compromising the confidentiality of user data. Similarly to the State Inference attack, this attack can be used to detect a specific state of the user on a website, e.g., if the user is authenticated by searching for the occurrence of the `Logout` string. However, it can also be used to determine whether a user searched for a specific term on a search engine or whether medical reports of the user contain certain keywords.

Chromium-based browsers support *URL Fragment Text Directives* [39] for specifying a string in the URL fragment that the browser emphasizes on the page. Adding `#:~:text=<string>` to the URL highlights matching strings on the page and scrolls to the first occurrence if it is located in a non-visible portion of the page. Furthermore, CTs offer the `extraCallback` callback. According to the documentation [14], this function is used to receive callbacks that are provided by the specific browser implementation. We experimentally found that Chrome, Edge, and Brave use this callback to fire the `onVerticalScroll` event whenever the user vertically scrolls within the CT. The event also includes the scroll direction. When the maximum scroll position on a website is increased by a user scroll, i.e., when the user scrolls further than they have scrolled before, the browser fires an `onGreatestScrollPercentageIncreased` event containing this maximum scroll position, where 0 refers to the beginning of the page and 1 to the end of the page. Querying this position can also be achieved by calling the `extraCommand` function. The maximum scroll position is not updated, and events are not fired when an automatic scroll is performed. Note that the scroll APIs required for this attack are only enabled if the *Help improve Chrome's features and performance* feature on Chrome and the *Optional diagnostic data* feature on Edge is enabled. These are, however, the default values.

The URL fragment text directive, the callbacks we discussed, and the Web Content Hiding gadget (Sec. 3.3.2) can be combined to stealthily infer whether a specific string is present on a website. To do so, an attacker opens a website in a CT and includes the target string in the URL's fragment text directive. Using the Web Content Hiding gadget, they hide the opened Web page from the user. If the string is found on the page, the browser automatically scrolls to the position where the text occurs. Then, the user can be lured into scrolling up within the CT by, e.g., showing a message in the bottom bar that instructs the user to perform a swipe-down, as shown in Fig. 3. When the user swipes down, the `onVerticalScroll` event is triggered, indicating that a scroll up was registered. If this event is followed by the `onGreatestScrollPercentageIncreased` event, the maximum scroll position was increased, even though no scroll down was performed. This indicates that the initial scroll position was not at the beginning of the page. Hence, the browser performed an automatic scroll and the fragment text appears on the page. If, however, the event is not fired, the initial scroll position is at the top of the page; thus, the text segment could not be found.

3.8. Bottom Bar Spoofing

In addition to being an enabler for the Web Content Hiding gadget, the bottom bar can be abused to introduce other security and privacy threats. More specifically, we identified two problems: leakage of user information and phishing. The root cause of both threats is the lack of visual feedback that allows users to understand whether the bottom bar is part of the webpage or belongs to the

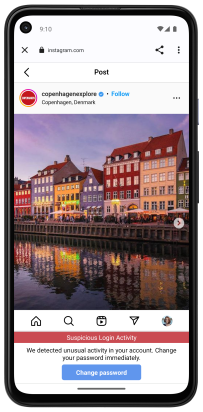


9:14

Cookie consent

We use cookies to offer you the best experience on our website. Also see our Privacy Policy.

(a) Information leakage using a cookie banner and the Web Content Hiding gadget (Edge 112).



Suspicious Login Activity

We detected unusual activity in your account. Change your password immediately.

(b) Phishing using a change password prompt (Chrome 112).

Figure 4: Attacks that make use of the Bottom Bar Spoofing.

native application: indeed, despite appearing as a component of the currently opened Web page, the bottom bar is under full control of the attacker. Although Android's `WebView` also allows mixing Web content with native UI components, the setting is different. The state in a `WebView` is not shared with a browser, thus, the risk of extracting sensitive user data or performing targeted phishing is lower compared to CTs.

3.8.1. Information Leakage. An application can declare that a CT should send an Intent if (some part of) the bottom bar is clicked. This Intent also includes the URL of the website that is currently open. While this feature offers a convenient way for the launching application to get informed about the activities of the user in the CT, it also acts as a side channel to infer user information. An attacker can customize the bottom bar to appear as a notification that is part of the website, such as a cookie banner, to trick the user into clicking on it. An example of such a bottom bar that masquerades itself as a cookie banner is shown in Fig. 4a. As soon as the cookie bar is clicked, the full URL of the website in the CT, including query parameters and the URL fragment, is transferred to the hosting application. This attack can be used, e.g., to deanonymize a user on a specific website, access Personally Identifiable Information (PII), or extract session identifiers and authentication tokens from the URL.

3.8.2. Phishing. The bottom bar can also be used for phishing purposes. When an application loads a CT with a website where the user is authenticated, the personal information displayed on the page, e.g., the user's username or profile picture, serves as a trust anchor for the user. By showing a prompt in the bottom bar that fits the current context of the website, the application can lure the victim into following the actions listed in the prompt, such as clicking on the bar, causing a redirection to a website where the actual exfiltration of user's data takes place. Similar to the former attack, the bottom bar is not visibly separated from the rest of the Web content and appears to be part of the website. Fig. 4b shows an example of such a phishing attack, which we detail as a case study in Sec. 6.

4. Mitigations

In this section, we propose and discuss strategies to mitigate the attacks we introduced in Sec. 3. As some of the mitigations are not limited to a specific attack, we first discuss cross-attack mitigations. We then present proposals targeting specific attacks that we have identified. A summary of these mitigation strategies can be found in Table 4.

4.1. Cross-Attack Mitigations

The mitigations discussed in this section aim to address core issues that we identified in the security model of CTs and thus serve as mitigations for all proposed attacks. One problem involves the lack of a mechanism for websites to prevent embedding from CTs. The other issue is the unrestricted state sharing between the CT and the browser.

4.1.1. Custom Tab Embedding Policies. Website operators are provided with standard mechanisms to specify embedding rules for the websites they own. Popular approaches include the Content Security Policy (CSP) [40] that obsoletes the `X-Frame-Options` header [28] and the Fetch Metadata headers [30]. However, websites have no way to opt-out from being displayed in a CT. By opting-out of being loaded in a CT, websites can prevent cross-context leaks targeted against them. Also, the proposed phishing attack is mitigated, since by disallowing the website to be loaded in a CT, the website cannot be used as a lure and a trust anchor, thus making the attack less useful. We propose two concrete implementations for this opting-out mechanism, namely by extending the CSP and the Fetch Metadata headers. Even though we frame these mitigations for CTs, the approach discussed in this section can be easily generalized to WebViews and other Web embedding components for mobile applications. Following our responsible disclosure process, we are currently discussing some of these proposals with the Google Chrome Security team and plan to engage with standardization bodies as a next step.

Extending Fetch Metadata Headers. Fetch Metadata defines a set of HTTP request headers that browsers are required to attach to outgoing HTTP requests to provide additional context. Servers can take advantage of these headers to make informed security decisions on whether a request must be blocked or not by implementing standard policies. Among the available headers, `Sec-Fetch-Dest` [41] specifies the destination of the request, including information on the embedding context of the request, e.g., the header is set to `iframe` when the request originates from an HTML `<iframe>` tag. We propose to extend the list of directives supported by this header with the `webview` keyword for HTTP requests originating from a CT. Provided with additional context,

	CT opt-out	Restrict state sharing	Restrict navigation CBs	Restrict CBs in background	Restrict scroll CBs & command	Restrict bottom bar height	Sanitize bottom bar Intent URL	Omit SameSite Strict cookies	Header sanitization
State Inference	◐	●	●	○	-	-	-	-	-
Header Injection	◐	●	-	-	-	-	-	-	●
SameSite Cookie Bypass	◐	●	-	-	-	-	-	●	-
Scroll Inference	◐	●	-	-	●	○	-	-	-
Bottom Bar Leak	◐	●	-	-	-	-	●	-	-
Bottom Bar Phishing	◐	●	-	-	-	-	-	-	-

TABLE 4: Comparison of mitigation strategies (● prevents attack, ◐ prevents attack only if opted-out, ○ only prevents stealthy attack).

Web servers can then determine how to handle the request. For example, the server may choose to apply additional security measures, such as requiring authentication or authorization, before returning the requested resource to the client. If the server identifies a security risk associated with rendering content within the CT and determines that redirecting to a stand-alone browser window is a safer alternative, it is also possible to break out of the CT and force-open a stand-alone browser window by leveraging an HTTP redirect. This can be achieved in Android CTs by setting an Intent in the `Location` header, such as `intent://<redir_website>#Intent;scheme=https;end`, which opens the given URL in the default browser.

Extending CSP. The Content Security Policy (CSP) is a broad Web security mechanism that can be used to allow-list the resources to be loaded in a Web page. Additionally, the `frame-ancestors` [29] directive defines the embedding policy for a page, i.e., whether the CSP-protected page can be embedded by another website via the `<iframe>` HTML tag or other methods. Similarly, we propose a new CSP directive called `webview-embed` to prevent unwanted embedding of a page in a CT. The directive may take the following values:

- `deny`: the page is not allowed to load in a CT;
- `trusted`: the page is only allowed to load in a CT if a relationship with the website and the host application can be established, e.g., using Digital Asset Links (DALs);
- `allow`: the page is allowed to load in any CT (default).

By providing developers with more granular control over the behavior of their Web pages in CTs, the `webview-embed` directive can help mitigate the security risks associated with the component. In particular, the `trusted` value of the `webview` directive can be used to ensure that sensitive data or functionality is only accessible within a trusted environment, reducing the risk of data leakage.

4.1.2. Restrict State Sharing. Compared to WebViews, CTs have the distinctive feature of sharing the state with the underlying browser. This allows users to benefit from browser features, such as the password manager, and seamlessly authenticate on websites where an active session is in place. However, this feature also opens the door to attacks that exploit the shared state, such as the Cross-Context State Inference attack, or allow the website to serve as a trust anchor for phishing attacks. To mitigate the issues that we have identified, we propose to open websites in a private browsing context by default. To reduce the impact on user experience, the CT APIs could be extended to support a permission dialog that explicitly requires users' approval to enable state sharing with the browser. Furthermore, the dialog could be bypassed, i.e., no user interaction is required, in presence of a DAL asserting a trust relationship between the native app and the website. We noticed that the `SFSafariViewController` component on iOS, which can be considered the iOS counterpart of CTs, is already following a similar behavior to the one we propose by requiring user mediation to enable state sharing. However, no way to share state automatically (e.g., via DALs) is currently supported.

4.2. Attack-Specific Mitigations

Unlike cross-attack mitigations, which protect against a range of security and privacy issues, this section discusses fixes on the individual attack vectors that we identified.

Restricting Navigation Callbacks. As navigation callbacks represent an inherent attack vector for cross-context information leakage, we propose restricting them. While completely eliminating these callbacks would disrupt existing usage in apps, as we assess in our measurement of real-world CT usage in Sec. 5, we can mitigate the impact on benign applications by reducing the granularity of shared information with the host application. This can be realized by grouping all the finished navigation events (including errors) into a single one. Alternatively, navigation callbacks could only be allowed in the presence of a Digital Asset Link (DAL). Additionally, a random delay before the callbacks are triggered would reduce the usefulness of the timing-based side channel.

Restricting Background Callbacks. Another possible mitigation strategy for the Cross-Context State Inference attack involves restricting callbacks to CTs that are in the foreground. Although this strategy would not entirely prevent attacks, it would hinder apps from launching them stealthily. Since CTs are designed to display content for active user interaction, we do not foresee legitimate use cases where a hidden CT would need to receive callbacks. Therefore, this mitigation does not introduce significant compatibility problems.

Restricting Scroll Callbacks and Extra Command. Allowing third-party applications to receive scroll events within the Web context is, as we have shown, not secure. To prevent this type of attack, one solution is to disable the scroll callbacks and additional commands that enable querying the scroll behavior. Similar to restricting navigation callbacks, browsers could also only enable this functionality on websites and applications for which a DAL is in place. Google took a more fine-grained approach to mitigation by only disabling scroll callbacks when a text fragment is part of the URL. Furthermore, they implemented enabling scroll callbacks only after a scroll down is registered, thus making it harder to determine the initial scroll position on the page.

Restricting Bottom Bar Height. The Web Content Hiding gadget stems from the fact that the height of the bottom bar can cover the entire viewport, thus hiding the content of the page. Preventing the bottom bar from overlapping the page content would thus make the Scroll Inference attack not stealthy anymore.

URL Sanitization in Bottom Bar Intents. URLs can contain PII and other sensitive data. Removing the URL from the Intent that is sent to the application when the bottom bar is clicked prevents the Bottom Bar Spoofing attack (Sec. 3.8). To ensure compatibility with existing applications making use of this feature, a compatible solution would be to sanitize the URL before sending it to the application to strip sensitive data, e.g., by keeping only the origin. Notice that this approach would still allow the application to infer the origin of the page visited by the user, which could still raise privacy concerns.

Omitting SameSite Strict Cookies. Loading a website in a CT causes a cross-context request between potentially untrusted parties. This is similar to loading a website in a browser pop-up from a cross-site position. Being a top-level navigation, SameSite `Lax` cookies should be transmitted, but `Strict` cookies must not be attached to the request.

Sanitizing HTTP Header Values. The Header Injection attack can be fixed by rejecting malformed CORS-approvelisted HTTP headers that contain the newline character `\n`. Unlike other mitigations discussed in this section, this fix does not impact compatibility as the vulnerability is caused by an implementation bug instead of a design flaw.

4.3. Ethical Disclosure and Adopted Mitigations

We disclosed the Cross-Context State Inference attack to Google in August 2021. Initially, the Chrome Security team categorized it as intended behavior but reopened the bug report after further interactions. As of July 2023, the vulnerability has not yet been fixed. We also reported the SameSite Cookie Bypass in June 2022, which unfortunately did not receive immediate attention. Although our report resulted in a bounty of $5,000, it was another party's later discovery of a related vulnerability (based on the same root cause) that led to the fix of SameSite `Strict` cookies and the change of the Fetch Metadata headers in CTs. Google issued CVE-2022-4926 for both our original report and the later identified issue. In addition, we reported the HTTP Header Injection, which was fixed in Chrome 109 by appropriately sanitizing HTTP header values. Our report resulted in a $3,000 bounty and has been assigned CVE-2022-4188.

In April 2023, we reported additional attack vectors for cross-context abuse, i.e., the Scroll Inference attack and the Bottom Bar Spoofing vulnerability. While the former has been fixed, awarded $2,000, and assigned CVE-2023-3736, we are actively engaging with the Chrome team to identify mitigations to address the latter. Google acknowledged the data leakage through the URL in the bottom bar Intent but has not yet decided to address the security risks due to phishing, which are considered acceptable.

Overall, our extensive analysis of the CT APIs and their security implications has raised awareness of the security risks associated with the mechanism. Recent interactions with the Chrome Security team have led to the acknowledgment that the original security model of CTs did not consider the possibility of cross-context attacks. As a result of our disclosure, Google released the Chrome Custom Tabs Security FAQ [42], which acknowledges the concerns raised in this paper.

5. Prevalence of Custom Tabs

To fully understand the impact of possible mitigation strategies on legitimate CT usage of applications, it is necessary to understand how the component is used in the wild. To this end, we assess the prevalence of CTs in a set of over 50K widely used applications available on the Google Play Store. For scalability reasons, we opted for lightweight static code analysis, which potentially introduces false negatives (i.e., we miss the usage of CTs) and false positives (i.e., we detect the usage of CTs in code that is never executed). We deem these limitations acceptable as our aim is to gain an overview of how CTs are used, and they do not undermine the validity of our key findings.

5.1. Dataset

As there is no comprehensive listing of applications available on the Google Play Store, we first retrieved the list of applications available from AndroZoo [43] as of December 2022. We then filtered the list for package names that are available on the Google Play Store and further selected only applications with over 1M installations according to their Google Play Store metadata that we collected with `google-play-scrape` [44]. This process left us with 54,988 candidate apps that we downloaded from a European Google Play Store using `gplaycrawler` [45] in the first two weeks of March 2023 on a Pixel 4 device running Android 13. We were able to successfully download 50,831 out of 54,988 applications (92%). We merged applications that are distributed as multiple ("split" [46]) APK files into a single APK using APKEditor [47]. If the merging process failed, we fell back to analyzing only the base APK.

5.2. Evaluation Methodology

Our analysis consists of three main stages. First, we perform an automated static analysis of the APK files (the distribution format of Android applications) to detect how many apps use CTs. Then, we analyze the class names for which we detect CT usage and distinguish between their use in the main app and in libraries. After compiling a list of the most commonly used libraries, we manually analyze them to gain insights into how CTs are used.

5.2.1. Detection of CT Usage. To automatically analyze applications for CT usage, we use Androguard [48], a static analysis framework for Android applications. We first search for the presence of the `android.support.customtabs.extra.SESSION` string, a prerequisite for launching CTs, to determine whether an app uses CTs [49]. We then search for the usage of the `launchUrl` function of the `CustomTabsIntent` class to collect all fully-qualified class names from which CTs are launched. Similarly, we check whether the `CustomTabsCallback` class is overridden in code other than the support library to collect all classes that use the CT navigation callbacks. Due to code obfuscation, e.g., classes or methods being renamed in the build process, this approach is not always able to identify the correct usage of the support library functions or classes. For this reason, we additionally collect the fully-qualified class name for classes that contain the `CustomTab` string in their signature.

5.2.2. Characterization of Libraries. The output of the static analysis phase includes, for every application, whether it uses CTs and a list of possible code locations where CT APIs are (likely) invoked. We then group these code locations by package name and aggregate them across all applications, collecting the most frequently used package names to build a list of CT-related libraries. We then manually analyze the libraries' documentation, open-source code, and sample applications to characterize their usage of CTs.

5.2.3. Limitations. We believe that our lightweight analysis is sufficient to understand and approximate the usage of CTs in the wild and reflect on the impact of mitigation strategies. However, we inherit fundamental limitations of static code analysis that can impact our results in the following ways.

Obfuscation. Developers can employ a wide range of obfuscation mechanisms to complicate reverse engineering, such as changing the signature of classes and methods and obfuscating strings. Our approach can only identify usages of CT APIs if they are not obfuscated. Similarly, we can only detect the CT session string, and thus determine whether an app uses CTs, if strings are not obfuscated. This may introduce *false negatives*, i.e., the application uses CTs, but we do not identify it. Similarly, when the package names of libraries are obfuscated, we miss them during our aggregation. This is a common limitation in Android application analysis, and the robust detection of third-party libraries is an open and orthogonal research area [50].

Dead Code. Applications may include code snippets that are never used and called in practice. If an application includes code fragments containing CT APIs, our analysis will flag the application as using CTs, leading to *false positives*. Note, however, that the Android build toolchain removes unused code by default when the application is minified [51], thus reducing the impact of this limitation.

5.3. Experimental Results

Out of the 50,831 applications in our dataset, the static analysis failed for 63 applications due to malformed APKs. 510 apps using split APKs could not be merged, leaving us with the base APKs.

Library (package name)	`launchUrl`	Callback
GMS Ads (`com/google/android/gms/ads`)	18,732	0
Facebook (`com/facebook`)	9,532	0
Inmobi (`com/inmobi`)	5,326	5,348
Firebase Auth (`com/google/firebase/auth`)	2,403	0
Firebase Msg (`com/google/firebase/inappmessaging`)	1,010	0
UniWebView (`com/onevcat/uniwebview`)	270	268
AWS (`com/amazonaws/mobile/client`)	125	151
BIGO (`sg/bigo/ads`)	127	127
Taboola (`com/taboola/android`)	143	87

TABLE 5: Top 5 sources of the `launchUrl` call and callback override and the amount of apps that use them. Note that one app can include more than one library.

5.3.1. Custom Tab Usage. Overall, we detected 42,372 APKs (83%) using CTs. Due to obfuscation, we were able to identify a call to the `launchUrl` function of the CT support library in 23,337 applications (55% of those that use CTs), and we identified callback usage in 6,147 APKs, i.e., 26% of those that also call `launchUrl`.

Table 5 reports the list of the top libraries included by the applications that call `launchUrl` or override `CustomTabsCallback`.

5.3.2. Custom Tab Usage Patterns in Libraries. We characterize the usage patterns of the CT APIs for all libraries mentioned in Table 5 as well as two of the most frequently used libraries (AppAuth, 229 applications, and OneSignal, 1,435 applications) for which, due to obfuscation, we could not detect the `launchUrl` call but contain `CustomTab` in their class signature.

Advertisement. Multiple applications use CTs to track interactions with advertisements. The InMobi Android SDK [52], BIGO Ads [53], and Taboola Android SDK [54] provide ways to monetize applications by showing advertisements. If clicking the advertisements redirects users to an external website, CTs can be used to open them. InMobi only uses the `TAB_SHOWN` and `TAB_HIDDEN` events. The events are then used to monitor if and how long a user viewed the content opened in a CT. BIGO Ads listens to and uses all callback navigation events. It records the navigation events and the timestamp at which they are fired and calculates the timespan between them. Taboola listens to the `NAVIGATION_FINISHED` event to record if the website was loaded successfully. To determine whether a website has failed to load, it does not use the `NAVIGATION_FAILED` event but sets a timeout. If the timeout is reached and no `NAVIGATION_FINISHED` event is fired, Taboola infers that the website has failed to load.

Authentication. The Facebook SDK for Android [55], Firebase Authentication [56], Amazon Web Services SDK for Android [57], AppAuth for Android SDK [58], and UniWebView [59] use CTs to authenticate users in the application and open the authorization request screen in them. The Amazon Web Services SDK and UniWebView use the `TAB_HIDDEN` navigation event. In AWS, it is used to capture when a user has interrupted the login flow. In UniWebView, it is used to inform the host that the user dismissed the browser.

In-app Messaging. In-app messages can be used to show users messages that are comparable to pop-ups. Developers can customize a message by adding buttons and associating each button with an event, such as opening a URL. Firebase In-App Messaging [60] and the OneSignal Android SDK [61] can use CTs to open this URL. Both libraries do not use callbacks.

Miscellaneous. In addition to authentication, the Facebook SDK for Android also uses CTs to show specific dialogs when sharing content on Facebook or joining games. CTs are also used in WebView libraries, such as UniWebView, which allows developers to open websites within an application, play videos, and authenticate users via OAuth 2.0. Developers can choose to open websites in a so-called "Safe Browsing Mode" that launches the website in a CT. As previously mentioned, UniWebView only uses the `TAB_HIDDEN` navigation callback to inform the host that the user dismissed the browser.

5.4. Discussion

Our evaluation shows that CTs are primarily used for two purposes: to show Web content that is out of the context of the application and for authentication with external services. In the first case, CTs provide a smoother user experience while transitioning to Web content than opening a separate browser. In the second case, CTs are used to present the user with the authorization request in authentication flows. The native application-based flows section of the OAuth 2.0 standard explicitly recommends CTs to initiate the authorization request [62]. We point out that the standard does not mention CT callbacks. Our empirical analysis confirms that callbacks are not widely adopted in OAuth 2.0 flows. We discovered a single usage of the `TAB_HIDDEN` callback in the AWS SDK to detect when the user has interrupted the login flow.

Our analysis reported that 6,147 applications (12% of our dataset) employ CT callbacks. The main use case of callbacks is to detect whether a user has closed the browser and to measure the time spent on a page. On the other hand, navigation callbacks, such as `NAVIGATION_STARTED` and `NAVIGATION_FAILED` are used less frequently. Based on our results, removing callbacks to mitigate the Cross-Context State Inference attack would primarily affect existing applications that rely on this feature for tracking and measuring user engagement. A viable approach to ensure compatibility and lower the impact of the attacks discussed in this work is to restrict callbacks to CTs in the foreground and enable more powerful callbacks only in the presence of a trust relationship between applications and websites, as discussed in Sec. 4.

6. Case Studies

To demonstrate the practicality of the proposed attacks, we report on case studies impacting prominent websites.

Sexual Preference Detection. The Cross-Context State Inference attack can be mounted using a range of attack vectors (Sec. 3.4.1). One such vector is timing, which impact we demonstrate against FetLife. FetLife is a social network targeted at users who consider themselves part of the "BSDM, Fetish & Kinky Community" [63]. The platform claims to have a user base of over 10 million users. Using the timing-based technique, a PUA can determine whether a user has an active session with the website and, thus, whether they consider themselves part of this community. The technique involves simultaneously opening `https://fetlife.com/users/sign_in` in a CT and a hidden WebView and measuring the loading times of the website in both Web components, as described in Sec. 3.4.

To illustrate the feasibility of this approach on FetLife, we carried out the attack 50 times with an active session with the website and replicated it 50 times without one. The experiments were performed on Chrome 115 on a Pixel 6a running Android 13. The timing results are presented in Fig. 5. The figure shows that a loading time in the CT exceeding the WebView's loading time (the reference value) by more than 400ms serves as a strong indication that the user is logged in. Conversely, a difference below 400ms suggests that the user is not logged in.

ProtonMail Login Detection. Besides timing, a Cross-Context State Inference attack can be mounted using the redirection-based attack vector. Here, we show the feasibility of the redirection-based technique against the privacy-focused email service ProtonMail (`protonmail.com`). When the user has an active session with the website, visiting the URL `https://account.proton.me/login` triggers a JavaScript redirect to the inbox page at `https://mail.protonmail.com/inbox`. In contrast, unauthenticated users visiting the same URL are prompted with a login form without incurring further redirections. By opening the `/login` URL in a CT, the attacker can probe if the user is logged in by counting the number of `NAVIGATION_FAILED` events received, i.e., two if the user is logged in and one otherwise. The attacker can also use the CT Activity Hiding gadget to make the attack stealthy.

Location History Leakage. The Google Maps Timeline service (`timeline.google.com`) allows users to view their location history. Devices on which the user is logged in with their Google account report their location to Google if the feature is enabled. By encoding the target date in the URL and specifying the target location in a URL fragment text directive as in `https://timeline.google.com/maps/timeline?pb=!1m2!1m1!1s<yyyy-mm-dd>&#:~:text=<location>`, the Web interface offered by the service can be abused to probe if the user visited a specific place on a specific date. Since the CT browser scrolls to the occurrence of the location string, the Scroll Inference attack can be used to leak this information. The granularity of the location that can be probed is the precise postal address, including the street number, e.g., `160 Broadway, San Francisco`. Because only one website can be opened in a CT at a time, one location-date-pair can be probed in one attack run.

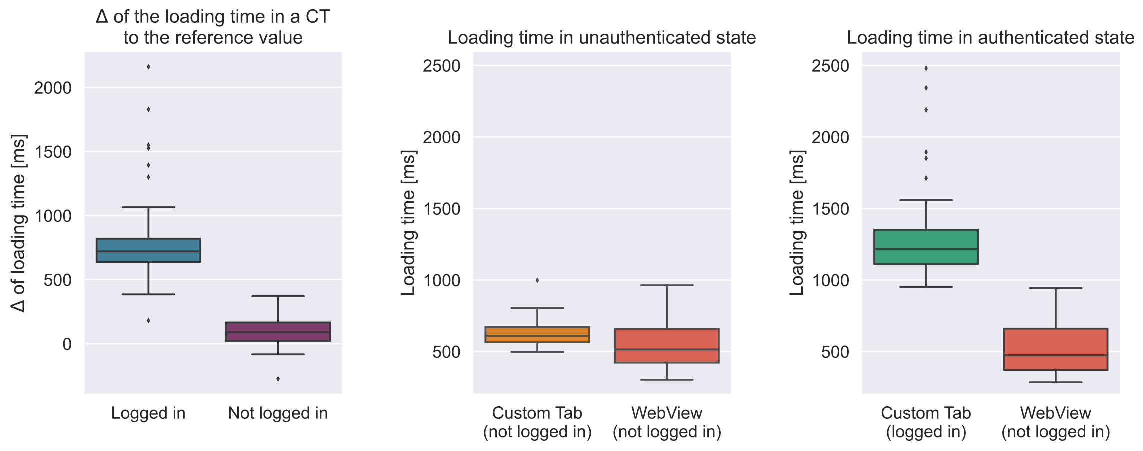


Figure 5: Cross-Context State Inference timing-based attack on FetLife ($n = 50$). The figure compares the delta between the loading time of the CT and the hidden WebView when authenticated and unauthenticated (left). It also shows the loading times of the website in each state (middle, right).

Social Handle De-Anonymization. Social networks often provide an authenticated endpoint that performs an automatic redirection to a URL that contains the user ID. For example, `https://m.facebook.com/profile` performs an HTTP redirection to `/<username>`. Other major platforms have similar endpoints, such as YouTube (`https://m.youtube.com/profile`) and the Russian social network VK (`https://vk.com/id0`). In combination with the CT bottom bar, these endpoints can be abused to perform a full de-anonymization of the user on social networks. The attack flow is as follows: (i) the user of a potentially unwanted application opens a legitimate website (`example.com`) in a CT, e.g., by tapping on a link; (ii) the application, instead, loads the Facebook `/profile` endpoint in the CT and hides the entire viewport using the Web Content Hiding gadget, showing a fake cookie banner in the bottom bar (Fig. 4a); (iii) the user accepts or rejects the cookies by tapping on the fake cookie banner, thus sending the current URL to the application, which can then obtain the user's Facebook username (Sec. 3.8.1); (iv) the application opens `example.com` in another CT window, thus simulating that the cookie banner simply disappeared. Notice that since we can hide the displayed URL in Edge CTs, the attack is fully stealthy in this browser.

Phishing for Credentials. We demonstrate a phishing attack on Instagram using the CT bottom bar, as shown in Fig. 4b. A PUA displays a post on Instagram using a CT. Assuming that the user has an active session on Instagram in the underlying browser, the page loaded in the CT shows the user's profile picture and other personal information. This helps build trust with the user, making them more confident in the authenticity of the page. The CT bottom bar includes a fake message that warns the user about unusual activity in the account, suggesting an immediate password change. After tapping on the bottom bar, the user is then prompted to enter both the current password and a new password, thus leaking the credentials to the application.

7. Related Work

Mobile Web Bridges. CTs are not the only way on Android to display Web content in mobile applications. Another such component is Android WebView [12]. Luo et al. [6] were among the first to analyze vulnerabilities in this component. They discussed how malicious websites loaded in WebViews can attack vulnerable applications and how malicious applications can attack websites loaded in WebViews. Tuncay et al. [64] and Zhang et al. [65] presented mechanisms for fine-grained access control when interacting with Web content on Android. Zhang et al. [66] performed a systematic evaluation of so-called app-in-app ecosystems that embed applications inside other applications using WebViews and found vulnerabilities in all 47 analyzed ecosystems. Furthermore, Zhang et al. [2] focused on the usability of in-app browsing interfaces, including CTs, and discovered that several in-app browsers fail to provide users with enough information about the website that is opened, including information about unsafe operations during browsing, e.g., when the website is served over an unencrypted connection. Similarly, Luo et al. [67] focused on the history of UI vulnerabilities in mobile browsers and found that 98.6% of the tested browsers are vulnerable to at least one attack. Most similar to our work, Palfinger et al. [68] mention the possibility of using the CT timing side channel to detect user logins and rely on Android animations to divert users' attention while the attack is being executed. Unlike our work, they only consider the timing side channel without extending the security analysis to other CT functionalities.

XS-Leaks. Our proposed Cross-Context State Inference attack based on CT callbacks to infer user information is comparable to the emerging class of cross-site leak (XS-Leak) attacks. XS-Leaks date back to 2000, when a timing-based attack to infer a user's browsing history was proposed by Felten and Schneider [69]. Since then, several other attacks have been introduced, including approaches to infer the user's authentication status by detecting differences in the response when loading resources that are only available to authenticated users [70], [71]. Bortz et al. [25] showed that timing side channels make it possible to detect fine-grained data about a user, e.g., the number of items in the shopping cart. Similarly, Gelernter et al. [72] were able to infer other sensitive user information, such as whether a user has sent another user an email containing a specific keyword. The attack is based on the assumption that services such as Gmail and Bing take a different amount of time to process search queries, depending on the state of the user. Acar et al. [73] showed how a Web-based attacker can discover the presence of certain IoT devices on a user's LAN using the differences in the resulting error messages when loading specific device endpoints in the HTML `audio` tag, thus allowing user profiling and tracking. Furthermore, Burnett and Feamster [74] demonstrated how XS-Leaks can be used to detect and measure censorship based on resource availability. Sudhodanan et al. [75] conducted a systematization of known XS-Leak attacks, categorized them in attack classes, and proposed a tool called Basta-COSI to automatically find cross-site leak attacks on target websites. Building on this work, Knittel et al. [76] developed a formal model of XS-Leaks and detected 14 new attack classes through a systematic search for new leak techniques. To evaluate the impact of XS-Leaks on different browsers, they created XSinator that automatically scans a Web browser and detects if it is vulnerable to XS-Leaks. Karami et al. [77] showed how service workers can enable additional XS-Leaks. In their work, they present techniques to conduct history sniffing attacks, as well as more fine-grained information leaks, such as user interactions on WhatsApp. Most recently, Rautenstrauch et al. [78] developed a framework to automatically discover XS-Leak observation channels in browsers, revealing 280 channels in Chrome, Firefox, and Safari. Moreover, testing the Tranco Top 10K websites for XS-Leaks, they found that it was possible to detect a user's previous website visits on 15%, and determine cookie acceptance on 34% of the tested websites.

8. Conclusion

We presented six new attacks that subvert the security of Android's Custom Tab component. These attacks can be used for Cross-Context State Inference, to inject HTTP headers into requests initiated by the CT, and to circumvent SameSite `Strict` cookies. Furthermore, we showed how an attacker can infer fine-grained information about a user by exploiting the reporting of user scroll actions and how the CT bottom bar can be exploited for information leakage and phishing. Since there is a multitude of different Chromium-based browsers available on Android (both in application markets and pre-installed and potentially customized with device firmware [79]), we restrict our study to Chrome, Edge, and Brave. In addition, we considered Firefox, which is the most-used non-Chromium-based browser available for Android [79]. While Firefox supports CTs, it is the only browser that is not vulnerable to any of our attacks. In order to mitigate these attacks on all browsers, we discussed how to address the core issues of the CT security model and responsibly disclosed the vulnerabilities to Google. Finally, we conducted a large-scale analysis of the usage of CTs in the wild and found that 42K out of over 50K applications use the component.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback. This material is based on research supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22060; and Grant ID: 10.47379/ICT19056]. This work has also been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), and SBA Research (SBA-K1), a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

References

[1] T. Steiner, "What is in a Web View? An Analysis of Progressive Web App Features When the Means of Web Access is not a Web Browser," in *The Web Conference*. ACM, 2018.

[2] Z. Zhang, D. Wu, L. Li, and D. Gao, "On the Usability (In)Security of In-App Browsing Interfaces in Mobile Apps," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 2021.

[3] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A View to a Kill: WebView Exploitation," in *Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.

[4] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. ACM, 2017.

[5] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Mobile Security Technologies Workshop (MoST)*. IEEE, 2015.

[6] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android System," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[7] B. Anantapur Bache, "Cross-site Scripting Attacks on Android WebView," *International Journal of Computer Science and Network*, vol. 2, 2013.

[8] E. Chin and D. A. Wagner, "Bifocals: Analyzing WebView Vulnerabilities in Android Applications," in *International Workshop on Information Security Applications (WISA)*, 2013.

[9] M. Squarcina, S. Calzavara, and M. Maffei, "The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches," in *Workshop on Offensive Technologies (WOOT)*. IEEE, 2021.

[10] Chrome Developers, "Android Custom Tabs Overview," Feb. 2020, (Accessed on 04/28/2023, https://archive.is/lLR6E). [Online]. Available: https://developer.chrome.com/docs/android/custom-tabs/

[11] Apple Developer Documentation, "SFSafariViewController," (Accessed on 11/15/2023, https://archive.is/TIphA). [Online]. Available: https://developer.apple.com/documentation/safariservices/sfsafariviewcontroller

[12] Android Developers, "WebView," Oct. 2023, (Accessed on 11/15/2023, https://archive.is/ggV3c). [Online]. Available: https://developer.android.com/reference/android/webkit/WebView

[13] Apple Developer Documentation, "WKWebView," (Accessed on 11/17/2023, https://archive.is/xjO2v). [Online]. Available: https://developer.apple.com/documentation/webkit/wkwebview

[14] Android Developers, "CustomTabsCallback," Aug. 2023, (Accessed on 11/15/2023, https://archive.is/Ggws6). [Online]. Available: https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsCallback

[15] ——, "CustomTabsClient," Aug. 2023, (Accessed on 11/15/2023, https://archive.is/oP0lF). [Online]. Available: https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsClient

[16] P. Drotar, "How to add extra HTTP Request Headers to Custom Tab Intents," Aug. 2020, (Accessed on 11/15/2023, https://archive.is/qYvLo). [Online]. Available: https://developer.chrome.com/docs/android/custom-tabs/howto-custom-tab-request-headers/

[17] Google Developers, "Getting Started - Google Digital Asset Links," Nov. 2022, (Accessed on 11/15/2023, https://archive.is/RB0SO). [Online]. Available: https://developers.google.com/digital-asset-links/v1/getting-started

[18] Android Developers, "CustomTabsSession," Nov. 2023, (Accessed on 11/15/2023, https://archive.is/Cf7G3). [Online]. Available: https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsSession

[19] ——, "Connect to the network," Nov. 2023, (Accessed on 11/15/2023, https://archive.is/alswu). [Online]. Available: https://developer.android.com/training/basics/network-ops/connecting

[20] Z. Zhang, W. Diao, C. Hu, S. Guo, C. Zuo, and L. Li, "An Empirical Study of Potentially Malicious Third-Party Libraries in Android Apps," in *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2020.

[21] J. Wang, Y. Xiao, X. Wang, Y. Nan, L. Xing, X. Liao, J. Dong, N. Serrano, H. Lu, X. Wang, and Y. Zhang, "Understanding Malicious Cross-library Data Harvesting on Android," in *Security Symposium (USENIX)*, 2021.

[22] Android Developers, "Tasks and the back stack," May 2023, (Accessed on 11/15/2023, https://archive.is/o0fCo). [Online]. Available: https://developer.android.com/guide/components/activities/tasks-and-back-stack

[23] XS-Leaks Wiki, "Introduction," Dec. 2020, (Accessed on 11/15/2023, https://archive.is/lV84m). [Online]. Available: https://xsleaks.dev/

[24] MDN web doc, "Redirections in HTTP - HTTP," Oct. 2023, (Accessed on 11/15/2023, https://archive.is/WfDgF). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections

[25] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *International Conference on World Wide Web (WWW)*. ACM, 2007.

[26] Y. Jia, X. Dong, Z. Liang, and P. Saxena, "I know where you've been: Geo-inference attacks via the browser cache," *IEEE Internet Computing*, vol. 19, no. 1, pp. 44–53, 2015.

[27] E. Kitamura, "Gaining security and privacy by partitioning the cache," Oct. 2020, (Accessed on 11/15/2023, https://archive.is/R7tww). [Online]. Available: https://developer.chrome.com/blog/http-cache-partitioning/

[28] MDN web doc, "X-Frame-Options - HTTP," Jul. 2023, (Accessed on 11/15/2023, https://archive.is/wip/RDMlU). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options

[29] ——, "CSP: frame-ancestors - HTTP," May 2023, (Accessed on 11/15/2023, https://archive.is/hSyTY). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors

[30] L. Weichselbaum, "Protect your resources from web attacks with Fetch Metadata," Jun. 2020, (Accessed on 11/15/2023, https://archive.is/Y1HP3). [Online]. Available: https://web.dev/fetch-metadata/

[31] XS-Leaks Wiki, "Navigation Isolation Policy," Dec. 2020, (Accessed on 11/15/2023, https://archive.is/UIzZE). [Online]. Available: https://xsleaks.dev/docs/defenses/isolation-policies/navigation-isolation/

[32] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the Web: A Journey into Web Session Security," *ACM Computing Surveys (CSUR)*, 2017.

[33] L. Chen, S. Englehardt, M. West, and J. Wilander, "Cookies: HTTP State Management Mechanism (IETF Draft)," Internet Requests for Comments, Internet Engineering Task Force, RFC 6265bis, 11 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-11

[34] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Conference on Computer and Communications Security (CCS)*. ACM, 2008.

[35] MDN web doc, "Authorization - HTTP," Aug. 2023, (Accessed on 11/15/2023, https://archive.is/5ooeK). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization

[36] M. Jones and D. Hardt, "RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage," Internet Requests for Comments, Internet Engineering Task Force, RFC 6750, Oct. 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6750

[37] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your cache has fallen: Cache-poisoned denial-of-service attack," in *Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[38] S. Khodayari and G. Pellegrino, "The state of the samesite: Studying the usage, effectiveness, and adequacy of samesite cookies," in *Symposium on Security and Privacy (S&P)*. IEEE, 2022.

[39] W3C, "URL Fragment Text Directives," Oct. 2023, (Accessed on 11/15/2023, https://archive.is/Ng7zj). [Online]. Available: https://wicg.github.io/scroll-to-text-fragment/

[40] MDN web docs, "Content Security Policy (CSP) - HTTP," Jul. 2023, (Accessed on 11/15/2023, https://archive.is/2RRln). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[41] MDN web doc, "Sec-Fetch-Dest - HTTP," Oct. 2023, (Accessed on 11/15/2023, https://archive.is/evnJJ). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Dest

[42] Chromium Docs, "Chrome Custom Tabs Security FAQ," (Accessed on 11/30/2023, https://archive.is/ZOJ8I). [Online]. Available: https://chromium.googlesource.com/chromium/src/+/main/docs/security/custom-tabs-faq.md

[43] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *International Conference on Mining Software Repositories (MSR)*. ACM, 2016.

[44] GitHub, "facundoolano/google-play-scraper," (Accessed on 03/28/2023). [Online]. Available: https://github.com/facundoolano/google-play-scraper

[45] GitLab, "marzzzello / gplaycrawler," (Accessed on 03/28/2023). [Online]. Available: https://gitlab.com/marzzzello/gplaycrawler

[46] Android Developers, "Build Multiple APKs," Apr. 2023, (Accessed on 11/15/2023, https://archive.is/J8rrw). [Online]. Available: https://developer.android.com/build/configure-apk-splits

[47] GitHub, "REAndroid/APKEditor," (Accessed on 04/11/2023). [Online]. Available: https://github.com/REAndroid/APKEditor

[48] ——, "androguard/androguard," (Accessed on 03/27/2023). [Online]. Available: https://github.com/androguard/androguard

[49] Chrome Developers, "Custom Tabs Low level API," Feb. 2020, (Accessed on 04/28/2023, https://archive.is/4CjAt). [Online]. Available: https://developer.chrome.com/docs/android/custom-tabs/low-level-api/

[50] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2020.

[51] Android Developers, "Shrink, obfuscate, and optimize your app," Nov. 2023, (Accessed on 11/15/2023, https://archive.is/Xu1QI). [Online]. Available: https://developer.android.com/build/shrink-code

[52] InMobi, "InMobi Android And IOS SDK," (Accessed on 11/15/2023, https://archive.is/ltTyO). [Online]. Available: https://www.inmobi.com/sdk

[53] BIGO Ads, "Bigo Ads Developer Management Platform," (Accessed on 11/15/2023, https://archive.is/Sz6jK). [Online]. Available: https://www.bigossp.com/guide/sdk/android

[54] Taboola, "Getting Started with the Android SDK," (Accessed on 11/15/2023, https://archive.is/7vQ0U). [Online]. Available: https://developers.taboola.com/taboolasdk/v2/docs/taboola-android-sdk-install

[55] GitHub, "facebook/facebook-android-sdk," (Accessed on 04/11/2023). [Online]. Available: https://github.com/facebook/facebook-android-sdk

[56] Firebase, "Firebase Authentication," Nov. 2023, (Accessed on 11/15/2023, https://archive.is/iuokm). [Online]. Available: https://firebase.google.com/docs/auth

[57] GitHub, "aws-amplify/aws-sdk-android," (Accessed on 04/11/2023). [Online]. Available: https://github.com/aws-amplify/aws-sdk-android

[58] ——, "openid/AppAuth-Android," (Accessed on 04/12/2023). [Online]. Available: https://github.com/openid/AppAuth-Android

[59] UniWebView, "UniWebView," Oct. 2023, (Accessed on 11/15/2023, https://archive.is/hk0Y8). [Online]. Available: https://docs.uniwebview.com/api/

[60] Firebase, "Firebase In-App Messaging," Nov. 2023, (Accessed on 11/15/2023, https://archive.is/uHBok). [Online]. Available: https://firebase.google.com/docs/in-app-messaging

[61] GitHub, "OneSignal/OneSignal-Android-SDK," (Accessed on 04/12/2023). [Online]. Available: https://github.com/OneSignal/OneSignal-Android-SDK

[62] W. Denniss and J. Bradley, "RFC 8252: OAuth 2.0 for Native Apps," Internet Requests for Comments, Internet Engineering Task Force, RFC 8252, Oct. 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8252

[63] FetLife, "The Social Network for the BDSM, Fetish & Kinky Community," (Accessed on 08/02/2023, https://archive.is/smiIe). [Online]. Available: https://fetlife.com/

[64] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on android," in *Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[65] X. Zhang and Y. Zhang, "React: A resource-centric access control system for web-app interactions on android," in *The Web Conference (WWW)*. ACM, 2021.

[66] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity confusion in WebView-based mobile app-in-app ecosystems," in *Security Symposium (USENIX)*, 2022.

[67] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis, "Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers," in *Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[68] G. Palfinger, B. Prünster, and D. J. Ziegler, "Androtime: Identifying timing side channels in the android api," in *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020.

[69] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Conference on Computer and Communications Security (CCS)*. ACM, 2000.

[70] J. Grossman and R. Hansen, "Detecting States of Authentication With Protected Images," Nov. 2006, (Accessed on 04/04/2023). [Online]. Available: https://web.archive.org/web/20150417095319/http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/

[71] M. Cardwell, "Abusing HTTP Status Codes to Expose Private Information," Jan. 2011, (Accessed on 11/15/2023, https://archive.is/TDNX0). [Online]. Available: https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information

[72] N. Gelernter and A. Herzberg, "Cross-Site Search Attacks," in *Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[73] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster, "Web-based attacks to discover and control local iot devices," in *Workshop on IoT Security and Privacy (IoT S&P)*. ACM, 2018.

[74] S. Burnett and N. Feamster, "Encore: Lightweight measurement of web censorship with cross-origin requests," in *Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[75] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[76] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk, "Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers," in *Conference on Computer and Communications Security (CCS)*. ACM, 2021.

[77] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[78] J. Rautenstrauch, G. Pellegrino, and B. Stock, "The leaky web: Automated discovery of cross-site information leaks in browsers and the web," in *Symposium on Security and Privacy (S&P)*. IEEE, 2023.

[79] A. Pradeep, A. Feal, J. Gamba, A. Rao, M. Lindorfer, N. Vallina-Rodriguez, and D. Choffnes, "Not your average app: A large-scale privacy analysis of android browsers," in *Privacy Enhancing Technologies Symposium (PETS)*, 2023.

[80] MDN web doc, "HTTP headers - HTTP," Jul. 2023, (Accessed on 07/31/2023, https://archive.is/KLu40). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

Appendix A. HTTP Header Injection

By default, only CORS-approvelisted headers can be added to the initial request in a Custom Tab. These approvelisted headers (as of Chrome 107) are summarized in Table 6. The Header Injection attack (Sec. 3.5) additionally allows the injection of non-CORS-approvelisted headers.

Given Chrome's implementation, certain headers either cannot be injected or can only be injected if absent in the original request. To identify these headers, we enumerated the approvelisted headers present in Chrome source code and the non-approvelisted headers listed on MDN [80], excluding caching- and encoding-related request headers. For each header, we executed the Header Injection attack on a Pixel 6a running Chrome 107 and Android 13 and observed whether it was actually sent. Some headers, such as `Accept`, were always sent in the initial request in a CT. For the others, we manually configured Chrome to send them (e.g., in the case of the `Cookie` header, we first set a cookie) and recorded whether the headers were sent accordingly. The results are provided in Table 6 and Table 7.

Feature	Sent by CT	Injection possible
Accept	●	–
Accept-Language	●	●
Content-Language	–	●
Intervention	–	●
Content-Type	–	●
Save-Data	–	●
Device-Memory	○	●
DPR	○	●
Width	–	●
Viewport-Width	○	●
Sec-CH-Viewport-Height	○	●
Sec-CH-UA	●	●
Sec-CH-UA-Platform	●	●
Sec-CH-UA-Arch	○	●
Sec-CH-UA-Model	○	●
Sec-CH-UA-Mobile	●	●
Sec-CH-UA-Full-Version	○	●
Sec-CH-UA-Platform-Version	○	●
Sec-CH-UA-Bitness	○	●
Sec-CH-UA-Reduced	○	●
Sec-CH-Prefers-Color-Scheme	○	●
Sec-CH-Device-Memory	○	●
Sec-CH-DPR	○	●
Sec-CH-Width	–	●
Sec-CH-Viewport-Width	–	●
Range	–	●
Sec-CH-UA-Full-Version-List	○	●
Sec-CH-UA-Full	○	●
Sec-CH-UA-WoW64	○	●

TABLE 6: Injection of CORS-approvelisted headers in Chrome 107 (● yes, ○ conditionally, – no).

Feature	Sent by CT	Injection possible
Content-Length	–	–
Connection	●	–
Keep-Alive	–	–
Proxy-Authorization	?	–
Upgrade	?	–
Authorization	–	●
Sec-CH-Prefers-Reduced-Motion	–	●
Downlink	○	●
ECT	○	●
RTT	○	●
Accept-Encoding	●	●
Except	–	●
Max-Forwards	–	–
Cookie	○	○
Access-Control-Request-Headers	–	●
Access-Control-Request-Method	–	●
Origin	–	●
Content-Disposition	–	●
Forwarded	–	●
X-Forwarded-For	–	–
X-Forwarded-Host	–	●
X-Forwarded-Proto	–	–
Via	–	●
From	–	●
Host	●	–
Referer	○	–
User-Agent	●	●
Upgrade-Insecure-Requests	●	–
Sec-Fetch-Site	●	–
Sec-Fetch-Mode	●	●
Sec-Fetch-User	–	–
Sec-Fetch-Dest	●	–
Sec-Purpose	–	●
Service-Worker-Navigation-Preload	–	●
Date	?	?
Early-Data	–	●

TABLE 7: Injection of (selected) non-CORS-approvelisted headers on Chrome 107 (● yes, ○ conditionally/only if not present, – no, ? unknown).

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper presents a series of attacks regarding Android's Custom Tab (CT) component. The authors attribute root causes to these attacks and discuss their mitigation strategies. Multiple CVEs are reported as a result.

B.2. Scientific Contributions

- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) This paper identifies multiple impactful vulnerabilities. The authors systematically study CT and discover multiple cross-context attacks to infer user-private states of CT or to phish the user. Concrete case studies that can help the community verify the impactful vulnerabilities discovered are provided. The root causes of each are well explained in the paper. The discovered vulnerabilities have been ethically disclosed. The vulnerabilities discovered have been confirmed by Google's Chrome team and led to mitigations. In-depth mitigation strategies that have been verified and adopted are presented.
2) This paper provides a valuable step forward in an established field. Integration of web content in apps has been primarily focused on WebView. The authors add to the existing landscape by exploring the security model of CT, hence moving the field forward.

B.4. Noteworthy Concerns

1) The paper lacks a systematic methodology that could generalize the discovery of the attacks. The attacks rely on manual investigation, testing, and review. Many of the discovered attacks depend on bugs that are not systematic. It may be hard for the community to learn a principle approach from this paper or replicate the discovery from the proposed attacks.
2) The empirical evaluation of CT prevalence may be inaccurate. The static analysis approach used to derive the results is prone to false positives and false negatives.