

The Bridge between Web Applications and Mobile Platforms is Still Broken

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

philipp.beer@student.tuwien.ac.at, {lorenzo.veronese, marco.squarcina, martina.lindorfer}@tuwien.ac.at

Abstract—The traditional way for users to access web content on mobile devices is by loading websites in a standalone browser like Google Chrome or Firefox. Websites and recently also Progressive Web Applications (PWAs) can, however, not only be rendered in such standalone browsers, but also in so-called mobile Web Views embedded in native mobile applications. PWAs are a new paradigm in web development that brings native app-like features, such as push notifications and offline usage, to the Web. We investigate the security of those Web Views at the intersection of application security and web security and present two new attacks: (1) an attack in which Android’s Custom Tab browser feature serves as a cross-site oracle to infer information about a user on target websites and (2) a vulnerability in Web View plugins of two third-party development frameworks that allows an attacker to use a vulnerable application to access the victim’s microphone and camera stealthily. We perform a preliminary real-world evaluation on the top 250 free Android applications and found that 5% of those that request microphone or camera permissions are potentially vulnerable to the Web View attack.

Index Terms—Mobile Security, Web Security, Web Views, Information Leakage, XS-Leaks

I. INTRODUCTION

Nowadays, web content is no longer only consumed on desktop devices, but also heavily viewed on smartphones and tablets. The worldwide market share of mobile and tablet devices amounts to 58.08% compared to a market share of 41.96% on desktop devices as of June 2021 [1]. While desktop devices offer the browser as the primary means to view websites, mobile devices are much more diverse. Mobile operating systems, such as the two big players Android and iOS, also allow developers to embed websites into native applications using a mobile Web View component. Developers can customize the Web View to match the application’s theme and minimize the visible disruption between the native application and the web content. Both Android and iOS offer multiple Web View components providing different functionalities. These components are widely used in mobile applications. For instance, the `WebView`¹ class, a Web View component in Android, is used in 85% of all free Android applications listed in the Google Play Store as of June 2014 [2]. Examples of applications that use such in-app browsers are popular social network applications like Facebook and Twitter.

Note. This is a work-in-progress paper presented at SecWeb 2022, co-located with the 43rd IEEE Symposium on Security and Privacy (S&P 2022).

¹To prevent confusion, we use the notation `WebView` to refer to the Android class and `Web View` for the generic embedding mechanism.

Mobile Web Views offer APIs that allow the host application to communicate with the web content loaded in the Web View and vice versa. Bridging the mobile native application and the web content can, however, have unforeseen consequences. Since Web Views are at the intersection between native mobile applications and web applications, classical security analysis of mobile applications, on the one hand, and web security, on the other hand, cannot be isolated but need to be considered as a whole. Applying traditional knowledge of both security disciplines separately does not lead to secure mobile applications when embedding Web Views. Security risks previously unknown to mobile applications can become a threat when these components are used [2]. Even though there was extensive research on the topic [3]–[6], new attack vectors, as we present in this paper, emerge. Research on these components is thus still relevant today.

Moreover, mobile Web Views can not only display static websites that serve simple HTML and CSS files over HTTP, but also dynamic websites such as Progressive Web Applications (PWAs) that make use of modern Web APIs. PWAs are a new paradigm in web development that brings features previously known only on native applications to the Web, such as push notifications, offline usage, and background synchronization. The availability of such APIs blurs the distinction between websites and native applications and makes them first-class citizens in the mobile ecosystem by, e.g., allowing users to add PWAs to the device home screen [7] or install them via the Google Play Store as Trusted Web Activities [8].

In this paper, we focus on the security of mobile Web Views. First, we state necessary background knowledge on mobile Web Views. We then define common threat models of mobile Web View attacks used in our work. Afterward, we propose two new attacks using Android’s Web View components. The first attack uses a malicious application to launch the Android Custom Tabs component to leak user information on target websites, whereas the second attack aims at the `WebView` component of vulnerable applications using third-party plugins of popular Android frameworks to access the user’s camera and microphone. We furthermore present possible mitigation strategies for each proposed attack.

In summary, we make the following contributions:

- We present a novel attack using Android Custom Tabs. In particular, the attack allows a malicious application to use the Custom Tab feature as a cross-site oracle to

infer information about the victim on vulnerable target websites. We also discuss possible mitigation strategies. To the best of our knowledge, this is the first attack identified on Android Custom Tabs (Section IV).

- We discover a vulnerability on the `WebView` component of two third-party plugins of popular Android frameworks. Particularly, we demonstrate how malicious websites loaded in vulnerable applications can use the camera and microphone permission to access those devices stealthily. We then discuss mitigation strategies for the attack (Section V).
- We perform a preliminary small-scale evaluation to understand the compatibility of the proposed mitigation for the Custom Tab attack. Furthermore, we evaluate the collected applications to quantify the risks posed by the `WebView` attack (Section VI). Our evaluation shows that 5% of the top 250 free Android applications that request either camera or microphone permissions are potentially vulnerable to the `WebView` attack.

II. BACKGROUND

Traditionally, websites on mobile devices are accessed by opening them in a standalone browser application like Google Chrome, Firefox, or Safari. Web pages can, however, also be viewed through in-app browsers that render web pages inside a native application. These embedded browsers are called `Web Views` [9]. Both Android and iOS offer different `Web View` components, that will be discussed in this section.

A. *Web Views on Android*

Android supports three different mechanisms for embedding web content in native apps, namely the `WebView` class, Custom Tabs and Trusted Web Activities [10].

Android `WebView`. The Android `WebView` component allows developers to create their own in-app browser. It does not offer standard browser features like the URL bar and navigation control buttons but rather just renders the web content. `WebViews` do not share state with either the browser that powers them, nor with other `WebView` instances, but instead are completely isolated. This means that user data, e.g., the history, cache, service workers, cookies, is not shared between the `WebViews` and the browser that is used for traditional web navigation [11]. The browser used for `WebView` is based on Chrome, nevertheless it does not support all features that Chrome supports [12].

It is straightforward for developers to use the `WebView` component inside an application, as shown in Listing 1. The Android app that hosts the `WebView` can interact with the website that is loaded, as well as vice versa. In the former case, the application can inject JavaScript code into the web content by calling `loadUrl(javascript:<code_to_inject>)` on the `WebView` object. Note that JavaScript is disabled by default and must be enabled explicitly by setting `settings.javaScriptEnabled` on the target `WebView` to `true`. The `WebView` executes the injected code in the context of the opened website [13]. In the latter case, the

```
1 val webView: WebView = findViewById(R.id.webview)
2 webView.settings.javaScriptEnabled = true
3 webView.loadUrl("https://example.com")
```

Listing 1: Opening `https://example.com` in a `WebView`.

web page can invoke Java or Kotlin code of the application. To do so, a `WebView` registers Java/Kotlin objects by calling the `addJavascriptInterface` function given a target class and a name as parameters. This makes all methods annotated with `@JavascriptInterface` inside the given class available from content loaded in the `WebView` [13]. JavaScript code of the website can then call these native functions.

Furthermore, applications embedding a `WebView` can listen to events happening inside the loaded web page and react appropriately. This can be achieved by so-called hooks. `WebViewClient` [14] implements hook functions with a default behavior that are triggered when the corresponding event fires. Those functions can be overridden which makes it possible to implement custom event handlers. Using this functionality, developers can, for example, restrict the possible domains that can be loaded inside a `WebView`.

Android Custom Tabs. Custom Tabs on Android provide a way to customize a browser `Activity` [15] to fit the host application's theme. It is a browser feature that browser vendors need to implement themselves. This also means that different browsers support different features of Custom Tabs. When a Custom Tab is launched, a new `Activity` of the browser is started. In comparison to `WebView`, Custom Tabs support all browser features that the underlying browser supports. They also offer performance advantages over `WebView` instances. This is achieved by features such as a pre-warming of the browser in the background and by informing it about likely future user navigation using the `mayLaunchUrl` [16] method so that the browser can perform preparatory work [10]. The method takes a list of URLs of likely future user navigation.

It is important to note that, compared to `WebViews`, Custom Tabs share state with the underlying browser, i.e., they use the same cookie jar, service workers, caches etc. [10]. For security reasons, the application that starts the Custom Tab cannot inject JavaScript code, neither can JavaScript code call Java/Kotlin code of the application [17]. The native application can, however, communicate with the web page via `postMessage` using the `CustomTabsSession` [16] class and can listen to navigation events inside the Custom Tab using the `CustomTabsCallback` [18] class. Those events are fired when the page has started loading, finished loading, aborted while loading, when an error occurred while loading the page, and when the tab becomes visible or hidden.

Android Trusted Web Activities. Trusted Web Activities display a full-screen browser tab without the browser UI [19]. They are built upon Custom Tabs and thus also share state with the underlying browser. Differently from Custom Tabs, `postMessage` is not supported [20]. The app that uses the

Trusted Web Activity and the website loaded in it must come from the same developer. This relationship is proven by Digital Asset Links (DALs) [8].

B. Web Views on iOS

Although this paper’s focus is on Android Web Views, we give, for the sake of completeness, a brief overview of Web Views on iOS. Web content inside an application on iOS can be rendered in two different ways: either by using `WKWebView` [21] or `SFSafariViewController` [22].

iOS WKWebView. This can be seen as the iOS-equivalent to `WebView`. As `WebView` on Android devices, `WKWebView` allows the native host application to inject JavaScript code into the rendered website [21]. In comparison to `WebView`, JavaScript is enabled by default [23]. It is also possible to call a native function of the application from JavaScript code inside the web page by exposing it using the `WKUserContentController.addMethod` [24].

iOS SFSafariViewController. This class, on the other hand, is the equivalent of Android Custom Tabs. It allows for handing over the responsibility of rendering a web page to Safari. Because Safari is used, `SFSafariViewController` can use the Safari features AutoFill, content blocking, Fraudulent Website Detection and Reader. Until iOS 10, `SFSafariViewController` shared state with the browser. This was changed from iOS 11 (released in 2018) on to not share any state with Safari [22].

III. THREAT MODELS

We consider two threat models under which an attacker can perform attacks using a mobile Web View: (1) either the content loaded in the Web View is malicious and attacks the benign application or (2) the application is malicious and attacks the legitimate content loaded in the Web View.

A. App Attacker Using a Web View (*AppAtk*)

An app attacker using a Web View (*AppAtk*) operates a malicious mobile application that attacks a benign website loaded in the Web View controlled by the application. The malicious application only requires permission to access the Internet and therefore seems relatively unobtrusive. On Android, the Internet access permission is a *normal permission*, i.e., it is automatically granted at install time [25]. The malicious application can be disguised as a useful, legitimate application to lure the user into downloading and using it. It is also plausible that an attacker operates a library or SDK that if used in a benign application, turns it malicious [26], [27].

The definition of this attacker is based on the *attacks from malicious apps* threat model proposed by Luo et al. [3] and illustrated in Fig. 1a.

B. Web-based Attacker Using a Web View (*WebBAtk*)

A web-based attacker using a Web View (*WebBAtk*) manages to load malicious content into a Web View. In contrast to an *AppAtk*, the application that embeds the Web View is benign. We note that a *WebBAtk* is different from a classical

web attacker known in the web security literature, since a *WebBAtk* can have capabilities that are out of scope for a standard web attacker. A *WebBAtk* is enabled by one of the following capabilities:

Malicious website. A *WebBAtk* hosts a malicious website and manages to load it into the benign Web View. Although this assumption sounds hard to fulfill, the literature has found various ways to do so, such as by improper usage and sanitization of Intent Filters² by the benign app. The attacker can also use other kinds of social engineering to navigate the user inside the Web View to the malicious website. This definition is similar to the *attacks from malicious web pages* threat model proposed by Luo et al. [3].

Another manifestation of this enabling condition is a gadget attacker as proposed by Barth et al. [29] that operates a malicious `iframe`, which is embedded in a benign website loaded in a Web View. This can be achieved through malicious advertisements [3]. Fig. 1b shows the threat model based on a malicious website.

Machine-in-the-middle attack (MITM). The *WebBAtk* manipulates the network traffic between a Web View and the server. This way, one can change the website loaded into the Web View and inject malicious code to attack the benign application. This specific attacker was proposed by Neugschwandtner et al. [4]. Fig. 1c illustrates the threat model.

Compromised server. A *WebBAtk* can also aim for the server of the website that is loaded inside a Web View of a benign application. The attacker can leverage web vulnerabilities of a website, such as persistent XSS or SQL injections, to manipulate the content of the server and turn the benign website malicious in order to attack the benign application [4]. Fig. 1d depicts the threat model based on a server compromise.

IV. CUSTOM TAB ATTACK

In this section, we present a novel attack that abuses the capabilities of Android’s Custom Tab component to infer user information on target websites, such as the user’s authentication status. The vulnerability is enabled by the fact that Custom Tabs share state with the underlying browser and by the availability of a callback mechanism for navigation events. The attack is not restricted to probing the authentication status of users, but can be extended to further information leakage, such as to detect whether a user has visited a website before, thus it potentially also opens doors for history sniffing attacks. To the best of our knowledge, research has not yet covered the security of Custom Tabs, and this is the first security analysis of this component.

We first discuss the threat model. Afterward, we present three techniques for inferring user information on target websites as well as how stealthiness can be achieved. In particular, we introduce the *status-code-based*, *redirection-based*, and *timing-based* approaches and compare vulnerable browsers. Subsequently, we argue the severity of the attack by stating

²Intents are used to communicate between two activities. Intent Filters declare what kind of intents an app can receive [28].

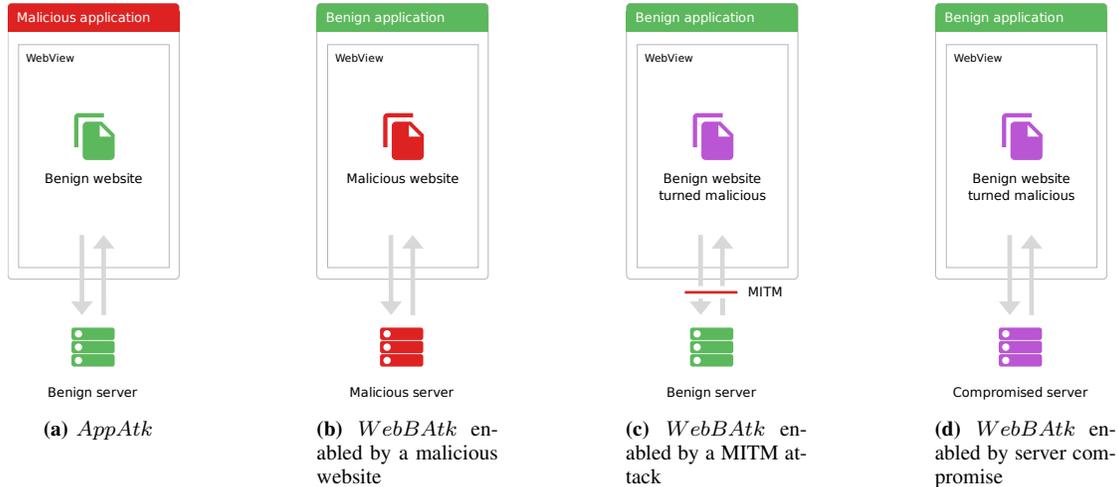


Fig. 1: Threat models on mobile Web Views considering an app attacker (*AppAtk*) and a web-based attacker (*WebBAtk*).

security implications that are not common in other similar approaches and propose possible mitigation strategies. We also provide a proof of concept of the attack.³

A. Threat Model

We consider an *AppAtk* as described in Section III that operates a malicious Android application that is installed on the victim’s device and is actively used. The goal of the app is to launch the attack while it is in use and stealthily load target websites into a Custom Tab by the application.

B. Attack Description

Custom Tabs share state with the underlying browser. This means that cookies, caches, service workers etc., are shared between the Custom Tab and the browser used by the Custom Tab. Consequently, if a user is authenticated on `example.com` in browser B_1 and a Custom Tab provided by that browser opens `example.com`, the user is also authenticated in the Custom Tab. However, if a Custom Tab provided by browser B_2 opens the same website, the user is not authenticated, since another browser is used.

To keep the application that launched it informed about ongoing events, Custom Tabs offer the `CustomTabsCallback` [18] class that provides a callback mechanism to the application. Custom Tabs distinguish six different types of navigation events [18]:

- `NAVIGATION_STARTED` fires when the website has started loading,
- `NAVIGATION_ABORTED` fires when the website loading is aborted due to a user event. This is the case when the user reloads the page or cancels the loading,
- `NAVIGATION_FAILED` fires when the website could not be loaded,
- `NAVIGATION_FINISHED` fires when the website has finished loading,

- `TAB_SHOWN` fires when the tab in which the website is loaded becomes visible and
- `TAB_HIDDEN` fires when the tab in which the website is loaded becomes hidden.

An *AppAtk* can infer sensitive information about a user by analyzing the sequence in which the different types of events are fired and measuring the time between events. This attack is comparable to an emerging class of web vulnerabilities called cross-site leaks (XS-Leaks) [30], in which a cross-site oracle is used to leak user information. In the Custom Tab attack, the callback mechanism of Custom Tabs serves as the cross-site oracle.

We have identified three techniques for inferring user information of the victim on a target website. For all the techniques, a malicious application launches the target website in a Custom Tab and analyzes the callback events. Although this section focuses on detecting the user’s authentication status, the attack is by no means restricted to that. Information of a user can be inferred whenever the sequence of the callback events and the timing of the events on a target resource depend on the state of a user on a target website. Such states can potentially also describe whether a user is logged in with a specific user account or to reconstruct the social graph of a user. The identified techniques are detailed below.

1) *Status-code-based Attack*: The status-code-based attack is enabled by the way Custom Tabs provided by Chrome, Edge and Brave handle websites with specific HTTP response codes. Our experimental evaluation showed that a website loaded with response code `4xx` or `5xx` and an empty response body triggers a different sequence of navigation events than a response with another status code, such as `200 OK`. The specific sequence of navigation events that are fired when such a website is loaded is depicted in Fig. 2. The website `example.com` loaded with status code `4xx/5xx` and empty response body triggers the `NAVIGATION_FAILED` event directly followed by the `NAVIGATION_FINISHED` event. A

³<https://github.com/secweb22-brokenbridge/custom-tab>

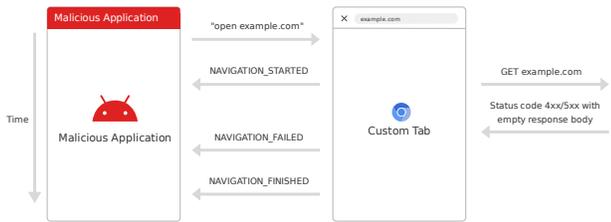


Fig. 2: Status-code-based Custom Tab attack.

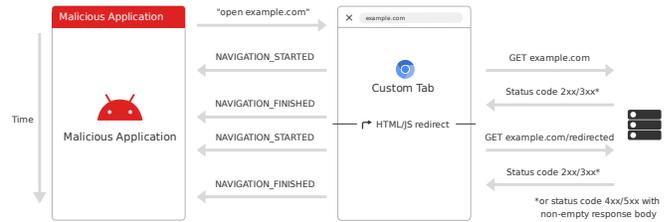


Fig. 3: Redirection-based Custom Tab attack.

website loaded with status code 2xx/3xx, independent of the response body, or 4xx/5xx with non-empty response body only triggers the NAVIGATION_FINISHED event.

Websites that follow the standard of HTTP response codes according to RFC 2616 [31] and return 401 UNAUTHORIZED when an unauthorized user is requesting a restricted resource are vulnerable to such an attack if the response body is empty. 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 impact of this method to leak information is significantly reduced. Responses with a non-empty body only trigger the NAVIGATION_FINISHED event, regardless of their status code.

2) *Redirection-based Attack*: An attacker can also infer user information on a target website by checking whether requesting a specific resource on this website 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. We have also identified websites that redirect users to a website for giving consent to using third-party cookies on the first visit. This allows an attacker to probe if a user has accepted the cookies and hence has visited the website before.

The behavior of a Custom Tab on a redirection depends on the type of redirection. MDN [32] lists three techniques to perform a redirection:

- **HTTP redirection.** HTTP responses with a status code in the 3xx range indicate a redirection to the client. The server can set a Location header to tell the browser the URL to redirect to.
- **HTML redirection.** A <meta> tag in the HTML head of the website with the http-equiv tag set to Refresh and the URL tag set to the target URL makes the browser redirect to the target when the page is loaded.
- **JS redirection.** Also JavaScript can be used for redirection. This can, for instance, be achieved by calling `window.location=<redirection_url>`.

Our experiments on Custom Tabs provided by Chrome, Brave and Edge have shown that the

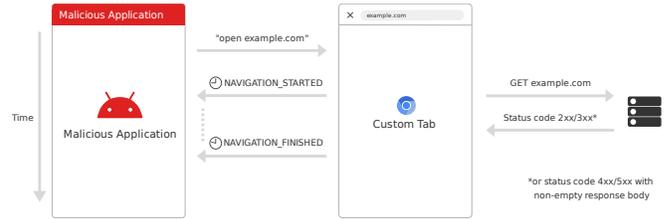


Fig. 4: Timing-based Custom Tab attack.

NAVIGATION_STARTED and NAVIGATION_FINISHED events are fired on redirection when HTML or JavaScript redirection is used, as can be seen in Fig. 3. Consequently, if the page `example.com/login` redirects to `example.com/home` using HTML or JavaScript redirection, two NAVIGATION_STARTED and, respectively, 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 vulnerable to the redirection-based attack.

3) *Timing-based Attack*: Using a timing-based attack, an attacker determines the amount of time a website takes to load by measuring the time interval between the NAVIGATION_STARTED and NAVIGATION_FINISHED events, as we show in Fig. 4. The underlying assumption is that the time required for a specific resource to be loaded in the Custom Tab depends on the status of the user on a website, e.g., the user’s authentication status. This is a valid assumption since the website server, if authenticated, may need to either do some heavy computation or load resources from the database.

To run the attack described above, one needs to compare the measured loading time with a baseline value. The baseline value represents the loading time when the user is certainly not authenticated. This could be a fixed value, however, this will not lead to consistent results, since the loading time depends on various factors, such as the speed of the network the user is connected to. One approach is to load the target website in a hidden WebView in the application, measure the loading time and compare it with the loading time in the Custom Tab. Since WebViews do not share state with the browser, the user is surely not authenticated in them. Although `onPageLoadStarted` and `onPageLoadFinished` in the `WebViewClient` class can be overridden to measure the loading time, our experiments have shown that the results are not always accurate. To get around this, one can use



Fig. 5: Procedure to hide the Custom Tab.

the PerformanceNavigationTiming Web API [33]. It allows retrieving information about the navigation events in a browser, such as the loading time of a document. In particular, the `duration` field can be used for the purpose of the attack. It returns the difference between the `loadEventEnd` property, that holds a timestamp when the document has completely finished loading, and the `startTime` property.

Because WebViews allow for the injection of JavaScript code, the malicious application can query the `duration` field by providing a JavaScript bridge to the application.

The timing-based attack may also be used when HTTP redirections to a target website are used. Even though HTTP redirections cannot be identified by using the redirection-based attack, every redirection adds an additional round-trip which is reflected in the loading time.

C. Achieving Stealthiness

An attacker aims to run the attack in a stealthy manner so that the user does not notice it. For that purpose, the Custom Tab that opens the target resource needs to be hidden from the user. As soon as the Custom Tab is opened by the application by calling `launchUrl` with the target resource as a parameter on the `CustomTabsIntent` object, a Custom Tab Activity is launched. The Custom Tab Activity is loaded full-screen and cannot be opened in the background or closed other than by a user click on the close button. To work around this behavior, the attack can be conducted as shown in Fig. 5. (1) Activity *A* of the malicious application launches the Custom Tab with the target resource `example.com`. (2) Activity *A* listens to the `TAB_SHOWN` event of the Custom Tab and launches Activity *B* as soon as it is received. (3) Even though Activity *B* is now in the foreground, the navigation events are still reported to Activity *A*.

One aspect that an attacker needs to consider is the back stack. The back stack stores all open activities in the order in which they are opened and is used for navigation and handling presses on the device’s back button. If the back button is pressed, the Activity on top of the back stack is removed, and the previous Activity continues in the foreground [34].

Since the Custom Tab Activity is right below Activity *B* in the back stack, the user would see the Custom Tab on a back button press. Although Custom Tabs can be started with the `Intent.FLAG_ACTIVITY_NO_HISTORY` flag that prevents the Activity from being added to the back

stack, using this does not result in the expected behavior. Our experiments have shown that employing this flag prevents Activity *A* from receiving callbacks from the Custom Tab when it is hidden by Activity *B*. Alternatively, one can override the `onBackPressed` [35] function in Activity *B* that is called by the Android system when the user presses the back button. The function can be overridden so that it launches a new Intent that starts Activity *A* again.

This approach to hide the Custom Tab leads to good results in practice. However, slower devices may take a while to start the Custom Tab and the underlying browser, which leads to cases in which the Custom Tab is shortly visible before Activity *B* is launched. The number of these cases can be significantly reduced by calling `warmup` [36] on the Custom Tab some time before opening the target resource. Calling the `warmup` method pre-initializes the underlying browser application in the background and contributes to a faster start after `launchUrl` is called on the Custom Tab.

Listing 2 shows a minimal implementation of the attack. Lines 1–19 implement the callback and save it to the `callback` variable. We specify the action when a specific event is received. Line 5 opens the overlay activity to hide the Custom Tab as soon as the `TAB_SHOWN` event is fired. The functions `onNavigationStarted`, `onLoadingFinished` and `onLoadingFailed` can be used to capture the sequence of fired events and measure the time between the events. For brevity, we omit the implementation of those. By implementing the `CustomTabsServiceConnection` class in lines 21–27, we specify the details of the Custom Tab that we want to launch. In line 23, we specify to create a Custom Tab session used for Custom Tab communication. We also tell the browser to call the `warmup` method in line 24. Line 26 provides an empty implementation for the abstract function `CustomTabsServiceConnection`. Line 29 binds the Custom Tab service and also specifies which browser to use. This is specified in the `packageName` string. The `context` variable is provided by the Android system. In line 30, we create the Custom Tab Intent using the session created earlier. Line 31 launches the URL given in `url` in the Custom Tab.

Improving the attack performance. We have experimentally evaluated whether multiple websites can be opened in a single attack run (i.e., in one activity transition). This can be achieved by sequentially launching Custom Tabs as soon as the `NAVIGATION_FINISHED` event of the preceding Custom Tab is fired and launching the overlay activity when the `TAB_SHOWN` event of the last Custom Tab is fired. In our experiments, we were successful to launch 5 Custom Tabs in about 2 seconds on a mid-end device. This, however, leads to an unresponsive UI for this time period.

D. Vulnerable Browsers

We tested the three attack techniques on Chrome 91, Firefox 89, Brave 1.24, Edge 46 and Opera 64 on Android 11. As summarized in Table I, our experiments have shown that Custom Tabs are supported by Chrome, Firefox, Brave and

```

1  val callback = object : CustomTabsCallback() {
2      override fun onNavigationEvent(
3          navigationEvent: Int, extras: Bundle?) {
4          when(navigationEvent) {
5              TAB_SHOWN -> {
6                  startActivity(Intent(this,
7                      OverlayActivity::class.java)
8                  )
9              }
10             NAVIGATION_STARTED -> {
11                 onNavigationStarted()
12             }
13             NAVIGATION_FINISHED -> {
14                 onLoadingFinished()
15             }
16             NAVIGATION_FAILED -> {
17                 onLoadingFailed()
18             }
19             else -> { }
20         }
21     }
22 }
23
24 val connection = object :
25     CustomTabsServiceConnection() {
26     override fun onCustomTabsServiceConnected(
27         name: ComponentName, client:
28         CustomTabsClient) {
29         session = client.newSession(callback)
30         client.warmup(0)
31     }
32     override fun onServiceDisconnected(
33         componentName: ComponentName?) { }
34 }
35
36 CustomTabsClient.bindCustomTabsService(context,
37     packageName, connection)
38
39 val cctIntent: CustomTabsIntent.Builder =
40     CustomTabsIntent.Builder(session).build()
41 cctIntent.launchUrl(context, Uri.parse(url))

```

Listing 2: Minimal Custom Tab attack implementation.

TABLE I: Vulnerability of major mobile browsers to Custom Tab attack (● vulnerable, ○ vulnerable but no stealth, ○ unaffected).

Browser	CT support	Status code	Redirection	Timing
Chrome 91	✓	●	●	●
Firefox 89	✓	○	○	○
Edge 46	✓	○	○	○
Brave 1.24	✓	●	●	●
Opera 64	✗	-	-	-

Edge. Even though the tested Opera, Brave and Edge versions are based on Chromium, Opera is the only one of those browsers that does not support Custom Tabs. Although Firefox supports them, no callbacks are reported and the browser is therefore not vulnerable. Chrome as well as Brave and Edge report events also on redirection, and therefore are vulnerable to the attack. Even though the attack itself can be carried out in Edge, stealthiness cannot be achieved. Custom Tabs provided by Edge are shortly visible before the overlay activity starts, even on high-end devices. A possible explanation for this is that the firing of the `TAB_SHOWN` event may be deferred.

E. Web Security Implications

It is important to stress, that opening a website in a Custom Tab represents a top-level navigation. Many security mitigations in place nowadays, such as SameSite cookies, X-Frame-Options, the frame-ancestors CSP directive, and Fetch Metadata try to tackle cross-origin attacks. The Custom Tab attack is, per se, however, not a cross-origin attack. There is no “cross-origin” component involved, i.e., there is no malicious website that tries to leak or access information from another cross-origin website. Instead, the Custom Tab attack uses capabilities of the (partially) user-controlled browser to infer information of top-level contexts. We note the following distinct advantages of the Custom Tab attack compared to other known XS-Leaks:

Bypassing SameSite cookies. XS-Leaks such as the ones identified by Bortz et al. [37] and Cardwell [38] use `img`, `iframe` and `script` HTML tags to load the target resources into a malicious website and use the `onerror` and `onload` event handlers to infer information. Depending on the status code or the time between the events, an attacker can infer information in a similar way as in the Custom Tab attack.

The SameSite attribute of the `Set-Cookie` [39] header can be used to set if a cookie should also be sent in cross-origin contexts or should be restricted to same-site contexts. When the attribute is set to either `Lax` (default value) or `Strict`, cross-site cookies are not sent on cross-origin requests, such as when an image is loaded in a `img` tag, a website is loaded in a `iframe` tag or a script is loaded in a `script` tag [40].

Setting the attribute to `Lax` or `Strict` thus prevents the attacks that use the `img`, `iframe` or `script` tags. Since the Custom Tab attack does not rely on cross-origin cookies, employing SameSite cookies does not prevent this attack.

Bypassing framing protection. Employing response headers that restrict whether a resource can be embedded in an attacker’s website, such as the obsolete `X-Frame-Options` (set to `DENY` or `SAMEORIGIN`) [41] and `Content-Security-Policy` (frame-ancestors directive) [42] headers do not prevent the Custom Tab attack. These headers only prevent or mitigate attacks that rely on the `iframe` tags for inferring user information.

Bypassing Cross-Origin-Opener-Policy. Other approaches on XS-Leaks, such as one identified by Masas [43], use `Window.open()` to load target websites in a new window and the window reference as a side-channel to infer user information. The `Cross-Origin-Opener-Policy` (COOP) [44] can be used to make the browser set the reference to the newly opened website to `null`. Since the reference is set to `null`, the channel cannot be used for inferring user information anymore.

Since the Custom Tab attack does not use the `Window.open()` method, COOP does not prevent this attack.

Bypassing Fetch Metadata. The Fetch Metadata HTTP request headers [45] are a set of HTTP headers that provide the context of the HTTP request to the server. The server can

use the context to determine whether the request is malicious or it should be allowed. This allows employing a Resource Isolation Policy [46] or a Navigation Isolation Policy [47] on the server-side. The `Sec-Fetch-Site` header can be used to inform the server on how the origin of the request and the origin of the requested resource are related. The header distinguishes `same-origin`, `same-site`, `none` and `cross-site`. The `Sec-Fetch-Mode` header specifies the mode of the request, such as whether it was a top-level navigation request, a CORS request etc. Moreover, there are the `Sec-Fetch-Dest` and `Sec-Fetch-User` headers that can also be used to employ the Isolation Policies.

Since HTTP requests of a website opened in a Custom Tab behave as they are open in the underlying browser and no cross-origin component is involved, Resource Isolation Policies and Navigation Isolation Policies employed at the server cannot distinguish malicious requests of the Custom Tab from legitimate ones.

F. Mitigation

As already discussed in the last section, classical mitigation strategies for XS-Leaks do not protect against the Custom Tab attack. Possible mitigation strategies for the attack that we have identified can be categorized into two classes: (1) measures that can be taken by Custom Tab providers and (2) measures that can be taken by the Android operating system.

Custom Tab Providers. Since the main cause of the attack is the fact that Custom Tabs offer a callback mechanism for reporting navigation events, a simple approach is to deactivate this mechanism. Although this would make all three attacks no longer available, the mitigation would break the intended functionality of the API.

By implementing a change of the behavior of callbacks on HTTP and JS redirections, Custom Tab providers can prevent the redirection-based attack. When only the initial `NAVIGATION_STARTED` and `NAVIGATION_FINISHED` events are fired, an attacker cannot detect a redirection anymore.

Custom Tab providers can also enforce opening websites in a private mode that does not share state with the browser. Since no state is shared, no information about the browsing activity in the underlying browser can be inferred. This strategy is used in `SFSafariViewController` on iOS.

Android Operating System. Another mitigation strategy that does not impact the functionality of Custom Tabs is to restrict the callback mechanism to Custom Tabs in the foreground. The Android operating system could block navigation callbacks from the Custom Tab when they are in the background hidden by another activity. This strategy does not prevent the attack per se, but hinders the stealthiness. It has no obvious downside, since Custom Tabs are used to display content for users to see; hence, there is no need to receive callbacks in the background.

V. WEB VIEW ATTACK

In this section, we discuss another attack on Web Views. In particular, the vulnerability stems from how plugins of

Android frameworks handle permission requests of content inside Android `WebViews`. This flaw allows an attacker to trick the user into loading a malicious website in a `WebView` of a vulnerable application and secretly access the victim's microphone and camera. We identified the vulnerability in two popular plugins of two third-party Android frameworks. We outline the vulnerability in detail and suggest strategies for mitigation. We also provide a proof of concept of the attack for both plugins.⁴

A. Threat Model

We assume a *WebBAtk* and a vulnerable application that is either based on React Native using the React Native `WebView` plugin, or on Unity using `unity-webview`, as described in more detail further below. The benign application is granted permission to access the device's camera and/or microphone.

B. Attack Description

In the Android operating system, every application runs in a sandbox that isolates it from other applications, as well as system resources and media devices, such as the microphone and camera. For this purpose, Android assigns a unique user id (UID) to every application [48]. To break out of this sandbox, applications can request permissions to access specific system capabilities. Consequently, if an app wants to access the device's microphone or camera, it must request the associated permissions. A permission request can be granted or denied by the user. Since Android 11 (released in 2020), permissions for access to the device's location, microphone and camera can also be granted on a one-time basis [49].

By default, the content inside the `WebView` is prevented from accessing the media devices that are granted to the application. The permissions need to be granted separately by the developer. This can be done by overriding the `onPermissionRequest` function of the `WebChromeClient` [50] class. Inside this function, `grant` must be called with the corresponding permissions as parameters.

For a `WebView` to access a media device, two conditions must thus hold: (1) the user must grant the application containing the `WebView` access to the media device and (2) the application must grant the `WebView` permission to access the media device.

We discovered that both React Native `WebView` and `unity-webview` implement the access control in a faulty way. React Native [51] is an open-source UI framework for Android and iOS. React Native `WebView` [52] is an open-source, cross-platform `WebView` for React Native that has about 4.000 stars and 2.000 forks on GitHub as of August 2021 and is therefore extensively used. Unity [53] is a cross-platform game engine. The `unity-webview` [54] plugin adds `WebViews` to the Unity View. `unity-webview` has about 1.600 stars and over 550 forks on GitHub as of August 2021.

Our research has shown that both plugins, by default, grant permission to the content inside the `WebView` to access

⁴<https://github.com/secweb22-brokenbridge/webview>

TABLE II: Vulnerable Web View plugins in the top 250 apps.

Permissions	RN WebView	unity-webview	Others
 ^ 	0	1 (< 1%)	113 (46%)
 ^ 	0	0	28 (11%)
 ^ 	2 (< 1%)	0	32 (13%)
 ^ 	5 (2%)	0	66 (27%)
 v 	7 (3%)	0	126 (51%)

the camera and microphone, if the permissions are already granted to the application. Even though unity-webview offers the `UNITYWEBVIEW_ANDROID_ENABLE_CAMERA` flag to explicitly enable the camera permission in a `WebView` and the `UNITYWEBVIEW_ANDROID_ENABLE_MICROPHONE` flag for microphone access [54], those media devices can also be accessed without setting the flags, if the application uses the microphone or camera for another purpose.

Neither React Native `WebView` nor unity-webview offers the possibility to deny access to media devices from the `WebView`. This coding style poses a considerable security risk. An attacker can open a malicious website in a `WebView` of a vulnerable application and access those media devices. To access the camera and microphone of the victim, the malicious website can use the `MediaDevices` [55] interface.

Since the vulnerable frameworks automatically grant the content inside the `WebView` access to the media devices, if the application has permission to do so, the victim is not asked for permissions again when the website is opened. The attacker-controlled website can also disguise itself as a website associated with the application to not raise any suspicion.

C. Mitigation

One possible mitigation strategy is to deny the `WebView` access to the media devices by default and provide an access control mechanism to application developers. This would allow developers to grant access to specific media devices, possibly only to some explicitly specified origins while preventing arbitrary origins from accessing sensitive media devices.

Another approach that is already employed in Android 12 (released in 2021) is to display a camera or microphone icon in the status bar when the camera or microphone, respectively, is being used [56]. This strategy does not prevent the attack from happening, but makes the victim aware that the application is accessing the camera or microphone and raises suspicion.

VI. PRELIMINARY EVALUATION

We performed a preliminary evaluation on real-world Android applications to assess (1) the number of applications for which the preconditions of the attack on `WebViews` hold and (2) the percentage of applications that use the callback mechanism of Custom Tabs to evaluate the impact of the mitigation strategy we propose in Section IV-F.

Our dataset comprises the top 250 most downloaded free Android applications listed in the Google Play Store as of February 2022, according to Android Rank [57]. We used the `apkeep` [58] downloader to obtain 247 APK files which

we first automatically analyzed to identify the permissions requested by each application and whether one of the two vulnerable `WebView` plugins are used. We then checked every application for the use of the Custom Tabs `launchUrl` function and `CustomTabsCallback`. A subsequent manual analysis of the apps that use callbacks gave us some insight on the context in which the feature is used.

WebView plugins. Our results are summarized in Table II. A total of 7 applications use React Native combined with the React Native `WebView` component and, at the same time, request camera or microphone permissions. Only one application uses Unity with unity-webview, however it requests neither of those permissions. We thus found 7 applications that are potentially vulnerable to the Web View attack. This amounts to 3% of all analyzed applications and 5% of the applications that either request microphone or camera access.

Custom Tabs callbacks. Out of the analyzed 247 APK files, 85 (34%) applications use Custom Tabs. We discovered that 57 (23%) of those make use of the callback mechanism, 13 of which use it in advertisement contexts. Deactivating the Custom Tabs callbacks would thus affect the functionality of these applications.

We are currently extending our evaluation on Android applications and we plan to perform a large-scale study on the Web to measure the number of websites vulnerable to the Custom Tab attack. This way we will quantify the threat posed by an *AppAtk* using CT with respect to a traditional Web attacker performing XS-Leaks.

VII. ETHICAL CONSIDERATIONS

We reported the Custom Tab attack to Google through the Chrome Vulnerability Reward Program. The Google Chrome team initially marked the attack as “intended behavior.” After further clarification, they reopened the bug report and stated that they will review the proposed mitigation strategies. There has, since then, been no activity on it for more than half a year. We have not received further information from Google on this matter.

We also disclosed the Web View attack to the React Native Web View and unity-webview developers. Following our recommendation, unity-webview introduced an access control mechanism, in which the boolean properties `SetCameraAccess` and `SetMicrophoneAccess` can be set on the `WebView` to allow access to the corresponding media devices. The React Native `WebView` developers have not yet responded to our bug report.

VIII. RELATED WORK

There has been extensive research on mobile Web View security so far. Because mobile Web Views are also heavily connected with PWAs, we also discuss identified attacks on those. We furthermore present related work on Web Views and XS-Leaks.

A. Web Views

Steiner [9] gives an overview of what mobile Web Views are and discusses the support of PWAs by different mobile Web Views. Luo et al. [3] are one of the first to analyze vulnerabilities in mobile Web Views. They discuss how malicious websites loaded in Web Views can attack vulnerable applications and how malicious applications can attack websites loaded in a Web View. Neugschwandtner et al. [4] show the magnitude of Web View attacks that target vulnerable applications. They conduct a large-scale evaluation on how many applications expose interfaces to the web content. They also propose new threat models on how web content can turn rogue. Mutchler et al. [2] also conduct a large-scale security analysis of applications that embed mobile Web Views and find that 28% of the studied applications have at least one vulnerability.

Stealing session cookies by using Android `WebViews` is discussed by Bhavani [5]. The attack is based on the assumption that a vulnerable application uses the `CookieManager` API to get cookies of specific URLs. The implication of the attack is, however, rather low, since Android `WebView` does not share state with other instances and the underlying browser and the victim would need to log in to websites in the malicious application.

Chin et al. [6] propose two new attacks on Web Views and implement a static analyzer to scan for them. Most recently, Rizzo et al. [59] propose `BabelView`, an information flow analysis tool to evaluate the impact of code injection attacks on Web Views. Using `BabelView`, they find 4,997 out of 25,000 analyzed applications from the Google Play Store contain vulnerabilities.

B. Progressive Web Applications (PWAs)

Karami et al. [60] show how service workers, an essential component of PWAs, can be abused for history sniffing purposes. Squarcina et al. [61] show how the `Cache` API used by service workers can escalate an XSS attack into a machine-in-the-middle attack. Papadopoulos et al. [62] draw attention to how features of PWAs can enable an attacker stealthy and persistent computation in the browser. This allows an attacker to mine crypto-currencies on the user's device, run distributed denial of service (DDoS) attacks against a target, etc.

C. Cross-Site Leaks

The proposed Custom Tab attack is comparable to the class of cross-site leaks. The `XS-Leaks Wiki` [30] contains a summarization of known `XS-Leak` attack vectors as well as possible mitigation strategies and is a perfect source to get an overview of the multitude of cross-site leaks.

The first identified cross-site leaks targeted the detection of the authentication status of a user. Such attacks were, among others, proposed by Grossman and Hanson [63] and Cardwell [38]. Bortz et al. [37] showed that it is even possible to detect fine-grained properties such as the amount of items in a shopping cart by using a time-channel.

Gelernter et al. [64] refined the time-channel attack, which makes it possible to infer even more fine-grained sensitive user information, such as if a user has sent another user an email containing a specific keyword. They use the search feature of services such as Gmail and Bing and listen to the time it takes to receive the response to the search query. Depending on the response time, they can draw the respective conclusions.

Sudhodanan et al. [65] conduct a systematization of known cross-site leak attacks, categorize them in attack classes and propose a tool called `Basta-COSI` to automatically find cross-site leak attacks on target websites.

IX. CONCLUSION

In this paper, we have proposed two new attacks on Android Web View components, which are used to render web content inside native applications.

In the first attack, we showed how the Custom Tab feature can be abused to leak user information on target websites stealthily. The attack is enabled by the callback mechanism of Custom Tabs, which reports navigation events, such as when and if a page finished loading successfully, to the host application. Stealthiness can be achieved because the Android operating system lets other activities overlay Custom Tab activities. Since the attack is not a classical cross-origin attack, web-based prevention mechanisms in place nowadays do not prevent the attack. Instead, we have presented mitigation strategies that can be taken by the Custom Tab providers and the Android operating system.

In the second attack, we showed how a vulnerability in two popular third-party `WebView` plugins for two Android frameworks enables an attacker operating a malicious website loaded in a `WebView` to use the application's camera and microphone permission to access those media devices. A possible mitigation strategy is to deny the content inside the `WebView` access to those media devices by default and employ an access control mechanism or use microphone and camera indicators to indicate access to the media device.

We are currently extending our preliminary evaluation to a large-scale analysis on the most popular Android applications and websites to assess the prevalence of the identified attacks and the impact of the proposed mitigations.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback. This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC). This research has further received funding from the Vienna Science and Technology Fund (WWTF) through project ICT19-056 (IoTIO), 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] Statcounter. Desktop vs Mobile vs Tablet Market Share Worldwide. Accessed on: Aug. 31, 2021. [Online]. Available: <https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide>
- [2] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, “A Large-Scale Study of Mobile Web App Security,” in *Proceedings of the IEEE Mobile Security Technologies Workshop (MoST)*, 2015.
- [3] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, “Attacks on WebView in the Android System,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.
- [4] M. Neugschwandner, M. Lindorfer, and C. Platzer, “A View to a Kill: WebView Exploitation,” in *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.
- [5] B. Anantapur Bache, “Cross-site Scripting Attacks on Android WebView,” *International Journal of Computer Science and Network*, vol. 2, 04 2013.
- [6] E. Chin and D. A. Wagner, “Bifocals: Analyzing WebView Vulnerabilities in Android Applications,” in *Proceedings of the International Workshop on Information Security Applications (WISA)*, 2013.
- [7] MDN Web Docs. Add to home screen. Accessed on: March 1, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Add_to_home_screen
- [8] Chrome Developers. (2020, Feb.) Trusted Web Activity. Accessed on: June 27, 2021. [Online]. Available: <https://developer.chrome.com/docs/android/trusted-web-activity/overview/>
- [9] 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 *Proceedings of The Web Conference (WebConf)*, 2018.
- [10] Chrome Developers. (2021, Feb.) Custom Tabs. Accessed on: Apr. 18, 2021. [Online]. Available: <https://developer.chrome.com/docs/android/custom-tabs/overview>
- [11] Android Developers. Web-based content. Accessed on: Apr. 20, 2021. [Online]. Available: <https://developer.android.com/guide/webapps>
- [12] Chrome Developers. (2014, Feb.) WebView for Android. Accessed on: June 27, 2021. [Online]. Available: <https://developer.chrome.com/docs/multidevice/webview/>
- [13] Android Developers. Building web apps in WebView. Accessed on: July 11, 2021. [Online]. Available: <https://developer.android.com/guide/webapps/webview>
- [14] —. WebViewClient. Accessed on: July 9, 2021. [Online]. Available: <https://developer.android.com/reference/android/webkit/WebViewClient>
- [15] Introduction to Activities. Accessed on: March 1, 2022. [Online]. Available: <https://developer.android.com/guide/components/activities/intro-activities>
- [16] Android Developers. CustomTabsSession. Accessed on: June 27, 2021. [Online]. Available: <https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsSession>
- [17] A. C. Bandarra. (2020, July) Best Practices. Accessed on: June 14, 2021. [Online]. Available: <https://developer.chrome.com/docs/android/custom-tabs/best-practices/>
- [18] Android Developers. CustomTabsCallback. Accessed on: June 5, 2021. [Online]. Available: <https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsCallback>
- [19] P. Conn. (2019, Aug.) Quick Start Guide. Accessed on: July 11, 2021. [Online]. Available: <https://developer.chrome.com/docs/android/trusted-web-activity/quick-start/>
- [20] GitHub. postMessage support? - Issue #55 - GoogleChrome/android-browser-helper. Accessed on: July 11, 2021. [Online]. Available: <https://github.com/GoogleChrome/android-browser-helper/issues/55>
- [21] Apple Developer Documentation. WKWebView. Accessed on: Apr. 18, 2021. [Online]. Available: <https://developer.apple.com/documentation/webkit/wkwebview>
- [22] —. SFSafariViewController. Accessed on: Apr. 19, 2021. [Online]. Available: <https://developer.apple.com/documentation/safariservices/sfsafariviewController>
- [23] —. allowsContentJavaScript. Accessed on July 11, 2021. [Online]. Available: <https://developer.apple.com/documentation/webkit/wkwebpagepreferences/3552422-allowscontentjavascript>
- [24] —. WKUserContentController. Accessed on: Apr. 18, 2021. [Online]. Available: <https://developer.apple.com/documentation/webkit/wkusercontentcontroller>
- [25] Android Developers. Connect to the network. Accessed on: June 5, 2021. [Online]. Available: <https://developer.android.com/training/basics/network-ops/connecting.html>
- [26] 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 *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2020.
- [27] 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 *Proceedings of the USENIX Security Symposium*, 2021.
- [28] Android Developers. Intents and Intent Filters. Accessed on: June 30, 2021. [Online]. Available: <https://developer.android.com/guide/components/intents-filters>
- [29] A. Barth, C. Jackson, and J. Mitchell, “Securing Frame Communication in Browsers,” *Communications of The ACM (CACM)*, vol. 52, pp. 17–30, 01 2008.
- [30] XS Leaks Wiki. Accessed on: Aug. 10, 2021. [Online]. Available: <https://xsleaks.com/>
- [31] IETF. RFC2616: Hypertext Transfer Protocol – HTTP/1.1. Accessed on: July 16, 2021. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc2616>
- [32] MDN Web Docs. Redirections in HTTP. Accessed on: June 10, 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections>
- [33] —. PerformanceNavigationTiming API. Accessed on: June 14, 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/PerformanceNavigationTiming>
- [34] Android Developers. Understand Tasks and Back Stack. Accessed on: June 17, 2021. [Online]. Available: <https://developer.android.com/guide/components/activities/tasks-and-back-stack>
- [35] —. ComponentActivity. Accessed on: June 9, 2021. [Online]. Available: [https://developer.android.com/reference/androidx/activity/ComponentActivity#onBackPressed\(\)](https://developer.android.com/reference/androidx/activity/ComponentActivity#onBackPressed())
- [36] —. CustomTabsClient. Accessed on: June 09, 2021. [Online]. Available: [https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsClient#warmup\(long\)](https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsClient#warmup(long))
- [37] A. Bortz and D. Boneh, “Exposing Private Information by Timing Web Applications,” in *Proceedings of the International Conference on World Wide Web (WWW)*, 2007.
- [38] M. Cardwell. (2011, Jan.) Abusing HTTP Status Codes to Expose Private Information. Accessed on: June 14, 2021. [Online]. Available: https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information
- [39] MDN Web Docs. Set-Cookie. Accessed on: Feb. 19, 2022. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>
- [40] —. Using HTTP cookies. Accessed on: June 14, 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
- [41] —. X-Frame-Options. Accessed on: June 14, 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>
- [42] —. CSP: frame-ancestors. Accessed on: June 14, 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors>
- [43] R. Masas. (2018, Nov.) Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends. Accessed on: Feb. 19, 2022. [Online]. Available: <https://www.imperva.com/blog/facebook-privacy-bug/>
- [44] MDN Web Docs. Cross-Origin-Opener-Policy. Accessed on: Feb. 19, 2022. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy>
- [45] W3C. (2021, Jul.) Fetch Metadata Request Headers. Accessed on: Mar. 30, 2022. [Online]. Available: <https://www.w3.org/TR/fetch-metadata/>
- [46] L. Weichselbaum. (2020, Jun.) Protect your resources from web attacks with Fetch Metadata. Accessed on: Feb. 19, 2022. [Online]. Available: <https://web.dev/fetch-metadata/>
- [47] XS Leaks Wiki: Navigation Isolation Policy. Accessed on: Feb. 23, 2022. [Online]. Available: <https://xsleaks.dev/docs/defenses/isolation-policies/navigation-isolation>
- [48] Android Open Source Project. Application Sandbox. Accessed on: July 2, 2021. [Online]. Available: <https://source.android.com/security/app-sandbox>

- [49] Android Developers. Request app permissions. Accessed on: July 2, 2021. [Online]. Available: <https://developer.android.com/training/permissions/requesting>
- [50] ——. WebChromeClient. Accessed on: July, 2021. [Online]. Available: <https://developer.android.com/reference/android/webkit/WebChromeClient>
- [51] React Native. React Native · Learn once, write anywhere. Accessed on: July 23, 2021. [Online]. Available: <https://reactnative.dev/>
- [52] GitHub. react-native-webview/react-native-webview. Accessed on: July 23, 2021. [Online]. Available: <https://github.com/react-native-webview/react-native-webview>
- [53] Unity. Unity Real-Time Development Platform. Accessed on: July 23, 2021. [Online]. Available: <https://unity.com/>
- [54] GitHub. gree/unity-webview. Accessed on: July 23, 2021. [Online]. Available: <https://github.com/gree/unity-webview>
- [55] MDN Web Docs. MediaDevices. Accessed on: July 4, 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices>
- [56] Android Developers. Behavior changes: all apps. Accessed on: Feb. 18, 2022. [Online]. Available: <https://developer.android.com/about/versions/12/behavior-changes-all>
- [57] Android Rank. List of Android Most Popular Google Play Apps. Accessed on: Apr. 25, 2021. [Online]. Available: <https://www.androidrank.org/android-most-popular-google-play-apps?price=free>
- [58] GitHub. EFXorg/apkeep. Accessed on: Feb. 20, 2022. [Online]. Available: <https://github.com/EFXorg/apkeep>
- [59] C. Rizzo, L. Cavallaro, and J. Kinder, “BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews,” in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.
- [60] S. Karami, P. Iliä, and J. Polakis, “Awakening the Web’s Sleeper Agents: Misusing Service Workers for Privacy Leakage,” in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2021.
- [61] M. Squarcina, S. Calzavara, and M. Maffei, “The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches,” in *Proceedings of the IEEE Workshop on Offensive Technologies (WOOT)*, 2021.
- [62] P. Papadopoulos, P. Iliä, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, “Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation,” in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2019.
- [63] J. Grossman and R. Hansen. (2006, Nov.) Detecting States of Authentication With Protected Images. Accessed on: Feb. 19, 2022. [Online]. Available: <https://web.archive.org/web/20150417095319/http://hackers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/>
- [64] N. Gelernter and A. Herzberg, “Cross-Site Search Attacks,” *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
- [65] A. Sudhodanan, S. Khodayari, and J. Caballero, “Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.