Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities

GuangLiang Yang, Jeff Huang, Guofei Gu
Texas A&M University
{ygl, jeffhuang, guofei}@tamu.edu

Abstract

In this paper, we present a novel class of Android WebView vulnerabilities (called *Differential Context Vulnerabilities* or *DCVs*) associated with web iframe/popup behaviors. To demonstrate the security implications of DCVs, we devise several novel concrete attacks. We show an untrusted web iframe/popup inside WebView becomes dangerous that it can launch these attacks to open holes on existing defense solutions, and obtain risky privileges and abilities, such as breaking web messaging integrity, stealthily accessing sensitive mobile functionalities, and performing phishing attacks.

Then, we study and assess the security impacts of DCVs on real-world apps. For this purpose, we develop a novel technique, *DCV-Hunter*, that can automatically vet Android apps against DCVs. By applying DCV-Hunter on a large number of most popular apps, we find DCVs are prevalent. Many high-profile apps are verified to be impacted, such as Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, and U.S. Bank. To mitigate DCVs, we design a multi-level solution that enhances the security of WebView. Our evaluation on real-world apps shows the mitigation solution is effective and scalable, with negligible overhead.

1 Introduction

Nowadays, mobile app developers enjoy the benefits of the amalgamation of web and mobile techniques. They can easily and smoothly integrate all sorts of web services in their apps (*hybrid* apps) by embedding the browser-like UI component "WebView". WebView is as powerful as regular web browsers (e.g., desktop browsers), and well supports web features, including the utilization of *iframes/popups*.

In the web platform, iframes/popups are frequently used, but also often the root cause of several critical security issues (e.g., frame hijacking [11] and clickjacking [23, 43]). In past years, in regular browsers, their behaviors have been well studied, and a variety of mature iframe/popup protection solutions (e.g., Same Origin Policy (SOP) [6], HTML5 iframe sandbox [4], and navigation policies [11]) have been deployed.

Inconsistencies Between Browsers and WebView. However, in WebView, a totally different working environment is provided for iframes/popups, due to WebView's own programming and UI features. Although these features improve app performance and user experience, they extensively impact iframe/popup behaviors and introduce security concerns. In particular, WebView enables several programming APIs (Figure 1) to help developers customize iframe/popup behaviors. For example, the setting APIs allow developers to configure their WebView instances. Thus, in the customized web environment (WebView), it is unclear whether existing iframe/popup protection solutions are still effective.

Furthermore, WebView UI is designed in a simple style (Figure 2) that only one UI area for rendering web content is provided. Due to the lack of the address bar, it is difficult for users to learn what web content is being loaded; due to the lack of the tab bar, it is unknown how multiple WebView UI instances (WUIs) are managed. Therefore, if an iframe/popup has abilities to secretly navigate the main frame (the top frame) or put their own WUI to the foremost position for overlaying the original WUI, phishing attacks occur and may cause serious consequences. Consider the scenario shown in Figure 3 and 4. The Huntington banking app (one million+ downloads) uses WebView to help users reset passwords (Figure 3-a,b). Inside WebView, the main frame contains an iframe for isolatedly loading untrusted third-party tracking content (Figure 4). However, if the untrusted web content inside the iframe obtains the ability of stealthily redirecting the main frame to a fake website (Figure 3-c), serious security risks are posed. For example, users' personal (e.g., SSN info and Tax ID) and bank account information may be stolen, and further financial losses may also be caused.

***Differential Context Vulnerability* (*DCV*)**. Motivated by above security concerns, we conduct the first security study of iframe/popup behaviors in the *context* of Android WebView. In this paper, we use the term "context" to refer to a web environment that includes GUI elements (e.g., the address and tab bars), corresponding web management APIs (e.g., the setting APIs in WebView), and security policies (e.g., SOP

Table 1: A Summary of Differential Context Vulnerabilities (DCVs)

Critical Features & Behaviors	Different Contexts		Attacks	Explanations	Consequences
	Browsers	WebView			
Main-Frame Creation	Address Bar	Java APIs	Origin Hiding Attack	Special common origins (e.g., null) Of Main-Frame	Sensitive functionalities behind postMessage and JavaScript Bridges can be leveraged, which may cause the leakage of sensitive information (e.g., location), and risky access on Hardware (e.g., camera and microphone)
Management of new popups	Tab Bar	Android Frameworks	WUI overlap attack	No protection on the WUI rendering sequence	Phishing attacks
			WUI closure attack		
Main-Frame Navigation	Address Bar	Java APIs	Traditional navigation based attack	Permissive navigation policies	
			Privileged navigation attack	Harmful conflict between WebView Customizations and web APIs	

and navigation policies).

As a consequence, our study uncovers a novel class of vulnerabilities and design flaws in WebView. These vulnerabilities are rooted in the inconsistencies between different contexts of regular browsers and WebView. As summarized in Table 1, several critical web features and behaviors (i.e., main-frame creation, popup creation, and main-frame navigation) are involved (see more details in Section 3). These features and behaviors are harmless or even safe in the context of regular browsers, but become risky and dangerous in the context of WebView. To demonstrate their security implications, we devise several concrete attacks. We show through these attacks, remote adversaries (e.g., web or network attackers on iframes/popups) can obtain several unexpected and risky privileges and abilities:

1) *Origin-Hiding*: hiding the origin when
   - breaking the integrity of web messaging (i.e., postMessage) [8], which allows the communication between mutually distrusted web frames; and
   - secretly accessing web-mobile bridges [21], which links the web layer and the *mobile* or *native* layer (e.g., Java for Android) together (Figure 1);

   Existing work has shown that postMessage's message receivers [44, 47] and web-mobile bridges [21, 49, 53] often carry sensitive functionalities. Thus, these functionalities can be further stealthily accessed by the untrusted iframe/popup through the attack. As a result, sensitive information (e.g., GPS location) may be stolen, and important hardware (e.g., microphone) may be unauthorizedly accessed.
2) *WebView UI Redressing*: performing phishing attacks by overlapping the foremost benign WUI with an untrusted WUI;
3) *(Privileged) Main-Frame Navigation*: freely redirecting the main frame to a fake website.

Moreover, we examine the effectiveness of existing protection solutions, which include not only the solutions designed for regular browsers (inherited by WebView), but also the solutions proposed for Android UI and WebView. We find that these solutions are ineffective to defend against the above attacks:

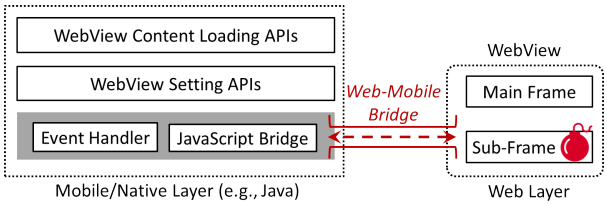
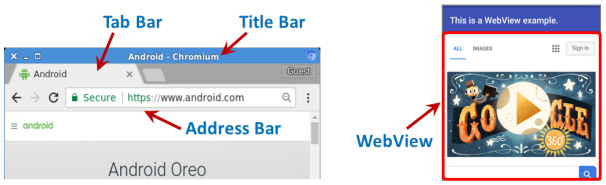


Figure 1: WebView Programming Features

1) For origin-hiding attacks, existing defense solutions for postMessage [11, 44, 47, 52] and web-mobile bridges [18, 21, 38, 45, 49] usually provide security enforcement relying on origin validation. However, unfortunately, the key origin information of the untrusted iframe/popup can be hidden during attacks, which leads to the bypass of the security enforcement.
2) For WUI redressing attacks, they are similar to Android UI redressing attacks [15, 20, 35]. However, the associated Android UI protection solutions (e.g., [13, 41]) are circumscribed to prevent WUI addressing attacks. This is mainly because that these protections work by monitoring exceptional Android UI state changes between different apps, while the WUI state change occurs within an app during attacks.
3) For main-frame navigation attacks, one related solution is the iframe sandbox security mechanism, which can effectively limit the navigation capability of an arbitrary iframe. However, through DCV attacks, an untrusted iframe can still break the above limitation and cause privilege escalation.

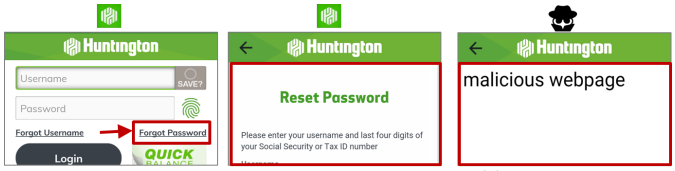
More details about the vulnerabilities and the weakness of existing defense solutions are presented in Section 3. For convenience, especially considering the root reason of this new type of vulnerability (i.e., the inconsistencies between the contexts of regular browsers and WebView), we refer to the vulnerability as *Differential Context Vulnerability* or *DCV*, and the associated attacks as DCV attacks.

DCV-Hunter & Findings. We next study and assess the security impact of DCV on real-world hybrid apps. To achieve the goal, we develop a novel static vulnerability detection technique, *DCV-Hunter*, to automatically vet given apps against



(a) Regular Browser UI (b) WebView UI

Figure 2: UI Comparison



(a) Click "Forgot password" (b) Run WebView (c) Secretly Navigate the Main Frame

Figure 3: Attacking the Huntington Bank App

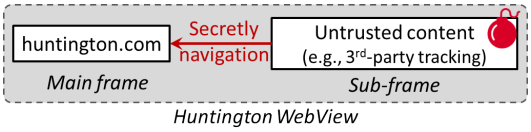
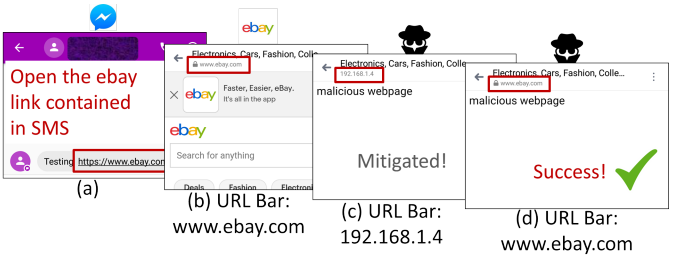


Figure 4: Attack Scenario



(a) (b) URL Bar: www.ebay.com (c) URL Bar: 192.168.1.4 (d) URL Bar: www.ebay.com

Figure 5: Attacking Facebook Messenger

DCV. Then, by applying DCV-Hunter on a number of most popular apps, we show that DCVs are prevalent. More specifically, we find 38.4% of 11,341 hybrid apps are potentially vulnerable, including 13,384 potentially vulnerable WebView instances and 27,754 potential vulnerabilities. Up to now, the potentially impacted apps have been downloaded more than 19.5 Billion times in total. Furthermore, our evaluation shows DCV-Hunter is scalable and effective, and has relatively low false positives (~1.5%).

We also manually verify that many high-profile apps are vulnerable (a list of video demos of our attacks can be found online [2]), including Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, WeChat, Kayak, ESPN, McDonald's, Kakao Talk, and Samsung Mobile Print. Several popular third-party development libraries, such as Facebook Mobile Browser and Facebook React Native, are also vulnerable and they influence hundreds of apps. Several special sensitive categories of apps are affected including leading password management apps (such as dashlane, lastpass, and 1password), and popular banking apps (such as U.S. bank, Huntington bank, and Chime mobile bank).

In our analysis, we also find that some apps implement their own URL address and title bars, which reduce the inconsistencies between regular browsers and WebView. However, these home-brewed URL bars hardly eliminate DCVs due to several limitations. One major limitation is that their implementation is often error-prone. For example, Facebook Messenger (Figure 5, one billion+ downloads) is equipped with the library "Facebook Mobile Browser" to handle URLs contained in messages (e.g., SMS). The browser library implements its own address bar (Figure 5-b) to reflect the change of web content (Figure 5-c) and mitigate DCV attacks (e.g., the WUI overlap attack). However, this address bar contains a design flaw (race condition). By combining a couple of DCV attacks, untrusted iframes/popups can still launch phishing attacks (Figure 5-d). Due to the inclusion of the vulnerable library, many high-profile apps are impacted, such as Facebook and Instagram. In addition to the vulnerable library, we find this design flaw is shared by many other popular apps that are not equipped with that library, such as Kakao Talk (100 million+ downloads).

We have reported our findings to the Android security team and many app developers. Up to now, a number of them (e.g., the Android and Facebook security teams) have confirmed our findings.

DCV Mitigation. DCVs are not caused by programming mistakes. It is extremely difficult for developers to eliminate the DCV security issues, especially considering the existence of the limitations in WebView (Section 3.6). To mitigate the problem, we propose a multi-level protection solution by enhancing the security of WebView programming and UI features. Our defense solution is implemented by instrumenting WebView's independent library, but without touching the source code of Android frameworks. Our solution is easy to use, and can simply work after developers involve our instrumented library, and provide a list of trusted domains. Our evaluation on real-world apps shows that our solution is effective and scalable, and introduces negligible overhead. Furthermore, considering the Android version fragmentation issue, we also test the compatibility of our solution. The result shows our solution is available in many major popular Android versions (5.0+), and covers almost 90% of Android devices in use.

Contributions. In sum, we make the following contributions:

- We investigate the security of iframe/popup in Android WebView, and discover several novel and fundamental design flaws and vulnerabilities in WebView (i.e., DCVs).
- We design a novel automatic vulnerability detection tool "DCV-Hunter" to quantify the prevalence of DCVs.
- We apply DCV-Hunter on a set of popular apps, and confirm that DCVs have severe security impacts.
- We further propose a multi-level solution to mitigate DCV attacks.

2 Background and Threat Model

Before we dive into our study of iframe/popup security, we first introduce necessary background information and our threat model.

2.1 Iframes/Popups and Related Protections

Iframes/popups are frequently used in web apps, for example, to view files in various formats (e.g., images, videos and PDFs), or load third-party untrusted web content (e.g., ads). They are easy to use. To create an iframe, developers can 1) either use the HTML element <iframe>; 2) or run JavaScript code to dynamically build an iframe DOM node.

Furthermore, to enable a popup, developers can use the following HTML code to generate a link:

```
<a href="URL" target="_blank|_top|frame_name|...".
```

When users click the link, "URL" will be opened in the frame that is determined by the "*target*" attribute. If *target* is "_blank", a new popup window will be opened to show "URL". Moreover, if *target* is "_top" or a specific frame name, "URL" will be loaded in the main frame or the specific frame determined by "frame_name". Developers can also use JavaScript code to open or close a web window:

```
window.open(URL, <target>, ...)  or window.close().
```

Similar to the usage of the HTML element <a>, "*window.open()*" can also determine where to open popup content.

Related Protections. Up to now, several practical protection solutions were designed and deployed in regular browsers:

- *Same origin policy (SOP)*: SOP isolates web frames whose origins are different. Note that SOP causes side effects that different origins are not allowed to communicate with each other. To mitigate the problem, the postMessage mechanism is designed in HTML5.
- *Built-in security policies*: Several built-in policies are available. For example, remote web code is not allowed to create a new sub-frame for loading local files, and the main frame is not allowed to load the data scheme URL.
- *HTML5 iframe sandbox*: The iframe sandbox mechanism can limit iframes' abilities, mainly including the enablement of JavaScript, main-frame navigation ("<a>" or "*window.open()*"), and popup-creation. Since the security of the popup behavior is one of our research objectives, we assume the popup-creation ability is allowed in iframe sandbox. Thus, in this paper, we mainly consider the abilities related to JavaScript enablement and main-frame navigation.
- *Navigation policies*: As studied in existing work [11], in regular browsers, the main frame is often exempt from strict navigation policies, which means any sub-frame can directly navigate the main frame by using "<a>" or "*window.open()*". There are several reasons for such a design. First, this type of navigation is frequently used by benign web apps, for example, for preventing framing attacks [43]. Second, even though the main frame is navigated, the consequence is quite limited in consideration of the stealthiness: any navigation can be explicitly reflected by URL indicators (e.g., the URL address bar).

2.2 WebView and Related Protections

WebView is an embedded, browser-like UI component. Android WebView is equipped with the newest kernel of the regular browser "Chrome/Chromium", and performs as powerful as regular browsers.

As discussed in Section 1, there are several inconsistencies between regular browsers and WebView. First, WebView UI is like a small and compacted version of a regular browser. It does not contain several common UI elements, including the address, tab, title and status bars.

Second, WebView UI is a case of view group, a collection of multiple Android UI components. More than that, it can also be added to an existing view group. A view group may consist of a set of WUIs with the same size. It manages multiple WUIs with a *rendering queue*, and only rendering the foremost WUI to users.

Third, the manners of initializing web content are different. Compared to regular browsers, which allow users to manually type the address of a website, WebView initializes web content through programming APIs (Figure 1), including

- *loadUrl(URL/file/JS)*: loading content in the main frame;
- *loadData(HTML, ...)*: loading code with the "null" origin;
- *loadDataWithBaseURL(origin,HTML,...)*: loading HTML code with a specified origin.

Last, as shown in Figure 1, developers can customize a WebView instance through several programming features, such as settings, and web-mobile bridges. Settings can manage WebView configurations, while Web-mobile bridges can link the web and mobile layers together. Generally, the bridges include 1) event handlers, which let mobile code handle web events that occur inside WebView; and 2) JavaScript bridges, which can allow JavaScript code to directly access mobile methods.

Furthermore, as shown in Table 2, several programming features can impact iframe/popup behaviors. To enable the creation of a popup, the setting *SupportMultipleWindows* should be set as true, and the event handler *onCreateWindow()* is also required to be implemented and return true. This event handler should create or open a WUI for rendering this popup, and also return the WUI to Android. Otherwise, the popup-creation operation will be ignored. This also means that *different popup windows are rendered by different WUIs at one time*. Besides, to support the closure of a WUI, the event handler *onCloseWindow()* should be also implemented. Note that when any web frame, including the main frame, loads content, the content should be approved by the event handler *"shouldOverrideUrlLoading()"*.

Summary of Related Protections. In past years, WebView security, especially the security of web-mobile bridges, has drawn more and more attention [12, 16, 21, 27, 30, 33, 34, 50, 53–55]. Several defense solutions [18, 21, 38, 45, 49, 50]

Table 2: Iframe/Popup-Related Programming Features

Features	Content	Explanation
Settings	*OpenWindowsAutomatically*	Enable "*window.open()*"
	SupportMultipleWindows	Enable the event handler "*onCreateWindow()*"
Event Handlers	*onCreateWindow()*	Handle window-creation
	onCloseWindow()	Handle window-closure
	shouldOverrideUrlLoading()	Handle URL-loading

were proposed to enhance the security of WebView by providing the security enforcement and access control mechanisms. However, we find they are ineffective against our new attacks. Section 7 provides a review of these existing work.

2.3 Threat Model

In this paper, we mainly focus on the hybrid app whose WebView contains an untrusted sub-frame. In our threat model, we assume the native or mobile code (e.g., Java code), and the main frame loaded in its WebView are secure and trusted. The main frame usually loads web content from the *first-party* benign domains (e.g., developer.com). For the embedded untrusted sub-frames, we mainly consider two *possible* attack scenarios:

Network attacks. When the sub-frames use HTTP network, attackers may perform man-in-the-middle (MITM) attacks to inject attack code into the sub-frames, and then launch DCV attacks. Although HTTPS have been widely adopted in modern web apps, there is still much legacy code using HTTP.

This scenario is feasible, especially considering many public unsafe WiFi hotspots are available [24]. Consider a possible scenario: attackers may set up a free WiFi hotspot in a crowded place. Nearby smartphone users may use this WiFi. If these users open vulnerable apps (e.g.,Facebook and skype) and click web links, apps' WebView may load these links. If the loaded web content embeds iframes/popups using unsafe network channels (e.g., HTTP), attackers may inject malicious code into the iframes/popups and launch attacks.

Web attacks. The inclusion of *third-party* content usually introduces security implications [26, 36]. Hence, we assume web attackers may be the owner of a third-party domain (e.g., ads.com) severing an embedded untrusted iframe/popup. Our empirical study on a set of popular hybrid apps and mobile websites shows iframes/popups are frequently used to load third-party content, especially third-party advertising and tracking content. Existing work has demonstrated that third-party advertising [28, 56] and tracking [14, 32, 37, 42, 46] services often causes serious security concerns. More than that, as figured out by existing work [39, 48], a third-party iframe may even directly work as a malicious entry point for malware.

This scenario is also possible in practice. For example, as demonstrated in prior work (e.g., [36]), some domains may expire, which still commonly occurs in recent years. Attackers may register and get the control of these domains. If these domains are embedded by some websites in iframes/popups, attackers may broadcast these websites to lure users to access them using corresponding vulnerable apps (e.g., Facebook or Facebook Messenger). In the vulnerable apps, WebView may be started, and also access the domains controlled by attackers. Thus, attackers obtain chances to inject malicious code and launch attacks.

Furthermore, as discussed in Section 2.1, considering the security of the popup behavior is one of our research objectives, we assume the popup-creation ability of an iframe/popup is enabled in its sandbox attribute.

3 Differential Context Vulnerabilities

In this section, we mainly focus on DCVs, and also explain why existing defense solutions are ineffective to prevent DCV attacks. We first show the overview of our security study, and then present the details of each vulnerability. Last, we discuss the advantages of DCV attacks over existing attacks, also with the analysis of the root causes of DCVs.

3.1 Study Overview

Guided by the inconsistencies between regular browsers and WebView (Section 2.2), our security study of iframe/popup behaviors is mainly concerned with the following three dimensions:

The application of common origins. As introduced in Section 2.2, WebView content initialization APIs may create the main frame with common origins, such as "file://" and "null". For example, the invocation

```
WebView.loadurl('file:///android_asset/index.html')
```

can load a local file with the origin "file://", while *WebView.loadData()* and *WebView.loadDataWithBaseURL()* may create a main frame to load web data with the "null" origin.

However, these common origins are not unique for the main frame, and may be reproduced by untrusted iframes/popups in their inside sub-frames for launching attacks. More specifically, if an untrusted sub-frame can generate a new nested sub-frame "F_{nested}" with above common origins, the untrusted sub-frame may place its essential attack code inside F_{nested} to make risky operations, which are aimed to attack all potential objectives, including the main frame, other sub-frames, or WebView itself. In the attack process, the victims may validate the operations by checking the corresponding origins. However, the origin information they can obtain is F_{nested}'s origin, rather than the real origin (i.e., the origin of the untrusted sub-frame). Considering F_{nested} have the same origin as the main frame, the origin validation process fails. Finally, the victims may treat untrusted operations as benign operations and handled them as usual.

Our study confirms that a sub-frame is not allowed to generate a new sub-frame with the "file://" origin, due to built-in security policies (Section 2.1). However, a nested sub-frame with a "null" origin can still be generated by using the data scheme URL (e.g., *<iframe src="data://...">*), which is frequently used to load simple HTML code (such as images)

in the web platform. Although SOP can prevent cross-frame scripting between two "null" origins (e.g., the main frame and F_{nested}), untrusted sub-frames can still leverage the "null" origin to make several nefarious actions (Section 3.2).

Concise WebView UI design. As discussed in Section 1, WebView's UI design causes security risks that untrusted iframes/popups may perform phishing attacks, if they have the abilities of 1) manipulating the rendering order of multiple WUIs; 2) navigating the main frame. To verify the former potential ability, we first conduct an empirical study on a set of popular hybrid apps. This study is aimed to understand how WUIs are managed in practice. We find Android takes the responsibility of managing multiple WUIs. Our study also shows when a popup is created, Android place its WUI behind current WUI *at default*.

This WUI management strategy seems safe. However, it does not meet app development requirements. Instead, some apps manage WUIs by themselves, which is yet error-prone due to the design flaws of the WebView event handler system (Section 3.6). As a result, the crucial ability of manipulating the WUI rendering order is exposed (Section 3.3.1). Thus, an untrusted iframe/popup can get the ability of overlapping begin WUIs with its own WUI. Our study also shows that even when Android's default WUI management strategy is adopted, it is still possible for untrusted iframes/popups to change the WUI rendering order by combining WUI creation and closure operations (Section 3.3.2).

Second, to confirm the latter potential navigation ability, we study the navigation policies of WebView. We find WebView inherits permissive navigation policies from Chrome/Chromium. These navigation policies have been well investigated in the context of regular browsers (Section 2.1), but rarely scrutinized in the context of WebView. These navigation policies allow an untrusted sub-frame to navigate the main frame. Due to the lack of the address bar, the navigation based attack is stealthier and more powerful in the context of WebView (Section 3.4.1).

Note that the above navigation can be disabled by iframe sandbox (Section 2.1). But considering iframe sandbox is hardly used in practice, the attack is still prevalent and has negative security impacts in real-world hybrid apps. This is also verified in our evaluation (Section 5.2).

WebView programming features. As discussed in Section 1, WebView's programming features may impact the effectiveness of existing defense solutions. To verify it, we extensively test these protection solutions' performance, when different programming features are enabled. Consequently, we identify a critical conflict between WebView programming features and web popup-creation manners. By leveraging this conflict, untrusted iframes/popups can perform **privileged** main-frame navigation attacks, even when this sub-frame's navigation capability is disabled by iframe sandbox (Section 3.4.2).

DCVs and DCV attacks are summarized in Table 1. More details are discussed below.

3.2 Origin Hiding Attacks

As introduced in Section 3.1, in the context of WebView, security risks are introduced that untrusted iframes/popups may leverage the "null" origin (created through the data scheme URL) to hide their own origins while making stealthy risky actions. In this section, we introduce two extended attacks: attacking web messaging integrity (Section 3.2.1) and stealthily accessing web-mobile bridges (Section 3.2.2).

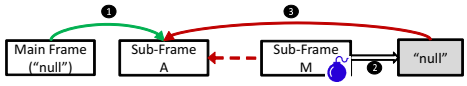


Figure 6: Attacking Web Messaging

3.2.1 Attacking Web Messaging

Figure 6 shows an attack scenario for web messaging. Assume the main frame whose origin is "null" sends web messages to a benign victim sub-frame. Meanwhile, the main frame also contains an untrusted sub-frame. If the untrusted sub-frame spawns a new nested sub-frame F_{nested} with the "null" origin, and let F_{nested} send a fake message to the victim sub-frame, the victim sub-frame may be fooled.

As shown in Listing 1, the victim sub-frame may validate the origin of the received message to ensure the message is from an authorized frame. However, this may not still recognize the fake message because the fake message has the same origin as the main frame. As a result, the victim sub-frame may handle the message as normal. If the victim sub-frame carries sensitive functionalities, these functionalities may be leveraged, and serious consequences may be caused.

```
1 // Message Handler
2 onmessage = function (e) {
3     // Validating the message source origin
4     if (e.origin == "null") { // From main frame?
5         // Making sensitive actions here
6     }
```

Listing 1: Validating the Message Origin in the Victim Sub-frame

In addition to the above origin validation based protection, the above attack cannot also be prevented by other defense solutions, such as [11,44,47,52], because it is challenging for them to distinguish between the main frame and F_{nested}.

3.2.2 Accessing Web-Mobile Bridges

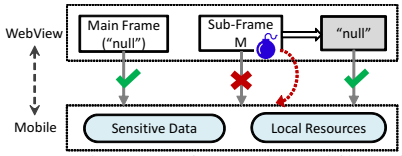


Figure 7: Freely Accessing Web-Mobile Bridges

As shown in Figure 7, the security risks are also posed that untrusted iframes/popups can also secretly access web-mobile bridges by leveraging the "null" origin (Listing 2), but without being blocked by existing defense solutions. This is because existing defense solutions are coarse-grained, and the

origin they can obtain is F_{nested}'s (i.e., "null"), rather than the origin of the untrusted iframes/popups. Hence, they would approve the untrusted operation.

To verify the attacks, we develop two proof-of-concept (POC) apps that can launch the attacks. Then, we test their performance when the-state-of-the-art protection solution "NoFrak" [21] and "Draco" [49] are enforced respectively. NoFrak extends SOP to the mobile layer of a third-party development framework, while Draco implements the access control in WebView. In the first POC app, we integrate the popular third-party hybrid development framework "Apache Cordova" and instrument its plugin manager to implement NoFrak. In the second POC app, we use our instrumented WebView library, which implements Draco's prototype system [49]. In both POC apps, we find that untrusted accesses by DCV attacks on web-mobile bridges, especially JavaScript bridges, cannot be prevented.

```
// Creating a nested sub-frame with the data scheme URL
var ifrm = document.createElement('iframe');
// Triggering onJsAlert()
ifrm.setAttribute('src', 'data:text/html;charset=UTF-8,<
      html>...<script>alert(\I am the main frame\', \'*\')</
      ' + 'script>'...
document.body.appendChild(ifrm);
```

Listing 2: Accessing the Event Handler onJsAlert() in the Untrusted Iframe/Popup

3.3 WebView UI Redressing Attacks

The root cause of the attacks is that there is no protection on the WUI rendering order and WebView UI integrity. Hence, the security risks exist that untrusted iframes/popups can freely manipulate it and perform phishing attacks. In this section, we illustrate two extended attacks: the WUI overlap attack (Figure 8-a), and the WUI closure attack (Figure 8-b). We next describe them in detail.

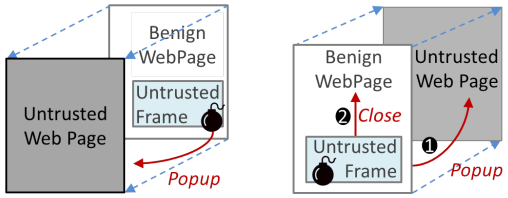


Figure 8: WebView UI Redressing Attacks

3.3.1 WebView UI Overlap Attack

```
// Customizing onCreateWindow() to enable popup-creation
boolean onCreateWindow(WebView view, ...) {
    // Creating a new WebView UI
    WebView myNewWebView = new WebView(getContext());
    // Initializing the new WebView UI

    // Putting the new WebView UI before current WebView UI
    view.addView(myNewWebView);

    // Providing the new WebView UI to Android
    ...
```

Listing 3: Vulnerable *onCreateWindow()*

Listing 3 shows a representative but vulnerable implementation of the event handler *"onCreateWindow()"*. When a popup is created, the event handler is triggered and may select to put the new WUI in the front of current benign WUI by calling "*ViewGroup.addView(new WebView)*" (Line 8). Thus, the new WUI is presented to users. However, this ability of changing the WUI rendering order can also be obtained by untrusted web code. This is mainly because the event handler *onCreateWindow()* cannot distinguish between benign and untrusted requests, due to its design flaws (Section 3.6).

As a result, untrusted iframes/popups obtain the ability of performing phishing attacks by simply triggering a popup-creation event, and letting the created WUI load fake web content and overlap the benign WUI. Due to the lack of the address and tab bars, this risky popup-creation operation may be hardly noticed by users. As shown in Listing 4, the overlap attack can be easily set up in practice.

```
// Using HTML Code
<a href="https://attacker.com" target="_blank" ...
// or Calling JavaScript code
window.open("https://attacker.com", "_blank" ...)
```

Listing 4: Exploit Code of the WUI overlap attack and the privileged navigation attack

We note that the key API name "addView" also appears in existing work on Android UI redressing attacks such as [35]. However, these APIs are totally different. In existing work, "addView" means "WindowManager.addView()", which is used to change UI layout between different apps. In this paper, "addView" means "ViewGroup.addView()", which is used to change a specific UI layout inside an app. To our knowledge, we are the first to discuss the security risk of the latter API.

3.3.2 WebView UI Closure Attack

When apps use the default Android WUI management strategy, it is still possible for an untrusted iframe/popup to change the WUI rendering order (Section 3.1). As shown in Figure 8-b, the untrusted iframe/popup may first create a new popup window, whose corresponding WUI is placed behind current benign WUI. Then, the untrusted code triggers the window-closure event, which is handled by the event handler "onCloseWindow()". If the event handler is vulnerable and removes the foremost benign WUI (Line 8 in Listing 5) from the WUI rendering order, the former untrusted WUI appears instead and phishing attacks may occur. Similar to the WUI overlap attack, due to the lack of the address and tab bars, such attacks are stealthy, and can be easily launched in practice (e.g., using the code in Listing 6).

```
// Customizing onCloseWindow() to enable WebView UI closure
public void onCloseWindow(WebView window) {
    super.onCloseWindow(window);
    // Destroying the WebView UI being closed
    ...

    // Removing the WebView UI being closed from current
        view layout
    myRootWebViewLayout.removeView(window);
}
```

Listing 5: Vulnerable *onCloseWindow()*

```
// Creating a new WebView UI
window.open("https://attacker.com", "_blank" ...)
// Closing current WebView UI
window.close()
```

Listing 6: Exploit Code of the WUI Closure Attack

We note that as introduced in Section 1, WebView UI redressing attacks cannot be defended by existing Android UI protection solutions. These two UI redressing attacks are different. Android UI redressing is performed between different apps, while WebView UI redressing occurs within one app.

3.4 Main-Frame Navigation Attacks

3.4.1 Traditional Navigation Attack

Untrusted iframes/popups can leverage traditional navigation policies (Section 2.1) to launch phishing attacks (e.g., using the code in Listing 7 to perform phishing attacks), when their navigation capabilities are not disabled. Due to the lack of URL indicators (e.g., the address bar), the attack is stealthier and may be hardly noticed by users.

```
// Using HTML Code
<a href="https://attacker.com" target="_top" ...
// Or Calling JavaScript code
window.open("https://attacker.com", _top, ...
```

Listing 7: Leveraging Traditional Navigation Policies

3.4.2 Privileged Navigation Attack

Even when the navigation capability is disabled by iframe sandbox (which prevents the above traditional navigation-based attack directly), it is still possible for untrusted iframes/popups to launch privilege escalation attacks and obtain the ability of performing navigation attacks. This is mainly caused by the inconsistencies between the WebView programming features and web regular navigation actions. When web popup creation code (e.g., <*a*> and *window.open()*) is executed in a sub-frame, Android always tries to select a WUI to show the popup content. Note that the WUI selection *always* occurs, even when popup-creation is disabled in the mobile layer (e.g., the setting *SupportMultipleWindows* is false). However, when popup-creation is not allowed, there is not a new WUI for rendering. Instead, Android selects current WUI for showing the popup content, which means the main frame is navigated to the popup. Thus, phishing attacks may occur.

In practice, the privileged navigation attack can be easily launched by using the exploit code shown in Listing 4. Note that this code is also used for launching the WUI overlap attack. When popup-creation is disabled (*by default*), the code may launch the navigation attack. Otherwise, the WUI redressing attack may be available.

3.5 Advantages of DCV Attacks

Compared to existing Android attacks (such as Trojan attacks [5]), DCV attacks do not require declaring permissions, or carrying payload. Compared to other WebView-based attacks (e.g., [21, 25, 30, 51]), which require JavaScript or JavaScript-bridges to be enabled, DCV attacks do not have these requirements and limitations. More importantly, DCV attacks are more powerful that attackers may obtain abilities to not only access web-mobile bridges, but also directly leverage critical web features.

Furthermore, different from existing MITM attacks on a sub-frame inside WebView, DCV attacks cannot be prevented by existing web protections (e.g., SOP). Unlike existing touch hijacking in WebView [31], DCV attacks do not need to control the mobile code, and craft the placement of multiple WebView components in Activity layout XML.

In addition, DCVs can be leveraged to boost other attacks. For example, event-oriented attacks [53] rely on triggering WebView event handlers, but it is difficult to trigger several critical event handlers (e.g., *onPageStarted()* and *onPageFinished()*). This problem can be well solved through exploiting DCVs, such as the privileged navigation attack (Section 3.4.2).

3.6 Root Causes of DCVs

DCVs are rooted in the inconsistencies between WebView and regular browsers in terms of UI and programming features (Section 1 and 3.1). We demonstrate several critical and frequently used web features and behaviors are harmless and safe in the context of regular browsers, but they become risky in the context of WebView.

In addition, we also find the design of the event handler features is also flawed. In theory, through event handlers, developers have chances to reject DCV attacks. However, unfortunately, the design flaws of event handlers make it extremely difficult to achieve the goal. For example, when the WUI overlap attack is performed, the event handler *''onCreateWindow(view,isDialog,isUserGesture,resultMsg)''* is always triggered. If the event handler could deny the creation of an untrusted WUI, attackers would fail to launch the WUI redressing attack. However, this is very difficult because the event handler *onCreateWindow()* does not provide the victim app any origin information about who is creating a popup and what content is being loaded in the popup. Thus, the victim app has to *blindly* allow or deny *all* popup-creation operations, no matter whether the operations are made by benign or untrusted code. In addition to *onCreateWindow()*, other event handlers such as *onCloseWindow()* face similar problems.

Another event handler *shouldOverrideUrlLoading(view,request)* (as introduced in Section 2.2) is always triggered when a URL loading event occurs. This event handler provides the information of the URL that is being accessed, which may be used as a complement of other event handlers to prevent DCV attacks (e.g., allow the victim app to deny untrusted URLs). However, the combination is hardly used in practice. Even when the associated URL is identified and denied, the new WUI is already created and still in the control of untrusted iframes/popups. Untrusted iframes/popups may still use the new WUI to consume the resources (such as CPU and memory) of the victim devices in background. Hence, to avoid this, it is required for the victim app to always explicitly destroy the new WUI.

In addition, *shouldOverrideUrlLoading()* often has its own implementation problems in origin validation. For example, our empirical study shows some hybrid apps do not even per-

form any check, and some of them only check the domain of the URL but ignore the scheme (e.g., "HTTP" or "HTTPS").

4 DCV-Hunter

There are several tools for analyzing hybrid apps [22, 53, 55], however, it is challenging to directly apply these tools to detect DCVs. On the one hand, existing static analysis tools are not designed for the analysis of iframe/popup behavior (e.g., [22, 55]), and they are often coarse-grained (e.g. [33]). More specifically, they can hardly extract and reconstruct the context information of each WebView instance. When there are multiple WebView instances in a hybrid app, which is common in practice, these tools can produce high false positives. On the other hand, existing dynamic analysis tools (e.g., [53]) have high false negatives, as it is very difficult to trigger a WebView instance at runtime. For example, as shown in Figure 5, to trigger WebView inside the Facebook Messenger app, the analysis tools need to automatically log in and open a URL link.

We propose a novel static detection tool, *DCV-Hunter*, that utilizes program analysis to automatically vet apps. As shown in Figure 9, DCV-Hunter's approach is four-fold. Given an app, DCV-Hunter first generates its complete call graph (CG). Next, DCV-Hunter leverages CG to reconstruct the context of each WebView instance. Then, DCV-Hunter verifies if untrusted sub-frames exists. Finally, DCV-Hunter determines if the given app is potentially vulnerable or not.

4.1 Complete Call Graph Construction

We leverage FlowDroid [10] to generate call graphs (CG) of the target app. However, we find FlowDroid face challenges to analyze WebView related function invocations. This is mainly due to the missing of type information and semantics related to WebView (e.g., the semantics of WebView event handlers). To mitigate this issue, we patch the target app during CG construction by inserting extra instructions, which provide necessary type and semantic information of WebView. Thus, FlowDroid can generate necessary edges and construct complete CG.

4.2 WebView Context Reconstruction

In this phase, DCV-Hunter re-constructs the whole context for each WebView instance. First, DCV-Hunter identifies all WebView instances from CG. Then, DCV-Hunter separately reconstructs each WebView instance's own context, which includes 1) the URL or HTML code to be loaded; 2) settings (e.g., the enablement of popup creation); 3) implementation of event handlers (e.g., "*onCreateWindow()*" and "*onCloseWindow()*"). To reconstruct the WebView context, points-to analysis is applied [33]. For example, when an event handler class that contains the implementation of event handlers is configured through the API "*setWebChromeClient(...)*", DCV-Hunter can check the points-to information of the API's parameter, and retrieve the parameter's actual class name.

However, points-to analysis does not scale well, especially when the target app is complex. To mitigate the problem, we also apply the data flow tracking technique (also provided by FlowDroid) as a complement. For example, when an event handler class is instantiated, the corresponding instance is treated as source. Then, the event handler configuration APIs (e.g., "*setWebChromeClient(...)*") are treated as sink. Finally, if there is a flow between above source and sink, the event handler class should be a part of the context of the corresponding WebView instance.

In addition to an event handler class, several context-related objects (e.g., URL strings, WebView settings) can also be analyzed using data flow tracking. These objects and their corresponding APIs are treated as source and sink, respectively. More details are shown in Table 3. Note that different from WebView settings and event handlers, which are often class instances, the URL source may have several different formats, such as 1) HTML code or URL string; 2) Intent messages (inter-component communication in Android). Both formats are often used in real-world apps. For example, as shown in Figure 5, in Facebook Messenger, when a link is clicked, an Intent message that includes the link is sent out to an activity (Andrioid UI) to start WebView and show that link.

Table 3: Source and Sink APIs

Source	Sink
URLs	WebView content loading APIs
Settings	WebView Setting APIs
Event Handlers	setWebViewClient()
	setWebChromeClient()
WebView	WebView content loading APIs
	WebView Setting config APIs
	Event handler registration APIs

4.3 Untrusted Iframe/Popup Detection

In this phase, given a WebView instance, DCV-Hunter checks whether an untrusted iframe/popup is included in its loaded content. To achieve the goal, DCV-Hunter first extracts the URLs of the untrusted iframe/popup, and then examine the event handler "shouldOverrideUrlLoading()" (Section 2.2) through path constraint analysis to determine whether extracted URLs are approved.

4.3.1 Untrusted URL Extraction

Given a WebView instance, the web content loaded in WebView is analyzed based on its formats:

- *HTML code*: This format is usually used by the content loading APIs "*loadData()*" and "*loadDataWithBaseURL()*" (for origin-hiding attacks). Based on the patterns of iframes/popups (Section 2.1), all internal associated links can be extracted and then checked. On the one hand, if a link is unsafe, such as using HTTP, code injection surface should exist, and the link is untrusted. On the other hand, if a link uses HTTPS, it is difficult to determine if the link is third-party, considering the main frame does not have an explicit domain (i.e., the "null"

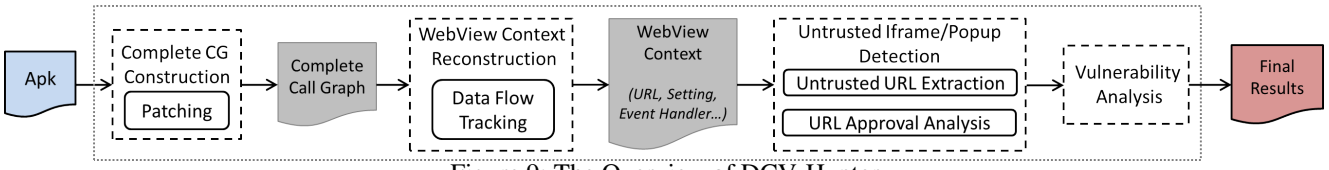


Figure 9: The Overview of DCV-Hunter

origin).

To mitigate the problem (i.e., determine the first-party URLs), we leverage several heuristics: 1) inside the target app, WebView class name and its internal package names are usually related with developers' website. Hence, we reverse them as first-party URLs. Please also note that the reversed class and package names should not be related to third-party URLs (e.g., [3]). 2) We also check the app information that is provided by developers in Google Play. This information includes the links of developers' home page, email and "privacy policy". Finally, these links are also treated as first-party URLs, since they are likely trusted by developers.

- *URL links*: DCV-Hunter handles URL links, based on their formats. If a URL is a network link, we build a crawler based on Selenium [7] to automatically collect the webpages (the mobile version) that can be navigated to from the URL within three depth levels. For each collected web page, its sub-frame is checked based on our threat model (Section 2.3).

  If URL is a local file link (e.g., "file://..."), DCV-Hunter first dumps the corresponding local file from the target app, and then handles it like above regular HTML code. This is mainly because the file scheme link is similar with the null origin and does not provide any first-party domain information.
- *Intent*: Our empirical study on a set of popular hybrid apps shows that the values of the links saved in an intent message may be arbitrary. Hence, to avoid potential false negatives, DCV-Hunter assumes that this format of web content contains untrusted iframes/popups.

4.3.2 URL Approval Analysis

To determine whether an extracted untrusted URL is approved by the event handler "shouldOverrideUrlLoading()" or not, we perform a path-sensitive constraint analysis on the event handler code. The key observation behind the idea is that based on the specification of the event handler [9], when untrusted iframes/popups are opened or created, the event handler is triggered, and should return false (Please note returning true is usually used for denying the link or other purposes [53]).

Below is our solution. We construct the conditions (constraints over strings) of the paths to "returning false", and check whether the extracted URL can satisfy the conditions. More specifically, based on the CG and control-flow graph of the event handler, we first find all the possible paths to the key instruction "returning false". Then, starting from each key instruction, we perform a fast backward slicing along each path

Table 4: APIs for the Analysis of WUI redressing problems

Attacks	Sensitive APIs
Overlap	*ViewGroup.addView()*
Closure	*ViewGroup.RemoveView()*
	WebView.setVisibility()
	...

to construct the path constraints. The unknown variables in the constraints are all over the string parameters (i.e., URL or request) of "shouldOverrideUrlLoading()". After that, based on our threat model and the content of extracted URLs, we add more constraints to the collected constraints, including

```
    1) <parameter>.scheme == "HTTP"
or  2)  <extracted_URL>.domain == <parameter>.domain.
```

The first constraint is aimed to check if attackers can freely inject code into the sub-frame through MITM attacks. The second constraint is used to verify if the domain of the extracted URL is approved. Finally, we use an SMT solver (i.e., z3 [19]) to solve all constraints. If path constraints can be satisfied, it indicates that the extracted URL should be approved.

Our path constraint analysis is implemented by embedding and extending the symbolic execution module of our previous work "EOEDroid" [53]. Please also note we also model several frequently used Java classes (e.g., WebResourceRequest, URL, and String) to support the related operations.

4.4 Vulnerability Analysis

To determine each vulnerability, DCV-Hunter checks its conditions respectively:

- *Origin-hiding*: DCV-Hunter first verifies whether the origin of the main frame is "null". This is done by checking the corresponding WebView content loading APIs and their associated parameters. Then, for convenience, the valuable attack targets are also checked, such as web messaging or web-mobile bridges.
- *WUI redressing*: DCV-Hunter first verifies WebView's settings and event handlers to check whether WUI creation and closure are enabled. Then, DCV-Hunter checks whether the corresponding event handlers *onCreateWindow()* or *onCloseWindow()* are vulnerable or not. This is done by checking the existence of the sensitive APIs listed in Table 4. Based on the analysis of the design flaws of these event handlers (Section 3.6), which have to blindly approve or deny all requests, these simple checks can obtain high accuracy.
- *Main-frame navigation*: For the traditional navigation based problem, iframe sandbox is checked. If iframe sandbox is used, DCV-Hunter then verifies if the navigation capability is disabled. For the privileged navigation attack,

DCV-Hunter checks whether multiple window mode is disabled, which is done by directly checking associated settings.

5 Security Impact Assessment

To assess DCVs's security impacts on real-world popular apps, we collected 17K most popular free apps from Google Play. They are gathered from 32 categories, and each category contains 540 most popular apps. By applying DCV-Hunter on these collected apps, we found that 11,341 apps contained at least one path from their entry points to WebView content loading APIs. Among them, 4,358 apps (38.4%) were potentially vulnerable, including 13,384 potentially vulnerable WebView instances and 27,754 potential vulnerabilities (Table 5). This indicates DCVs widely impact real-world apps.

We evaluated the accuracy of DCV-Hunter by measuring its false positives. We randomly selected 400 apps from the apps flagged as "potentially vulnerable" by DCV-Hunter, and manually checked them (see more details in Section 5.1). We find that 6 of them (1.5%) are false positives. Our further inspection revealed that in four of these apps, during the reconstruction of the URL loaded by WebView (Section 4.2), some unrelated URLs were accounted, due to the imprecise taint analysis (i.e., overtaint). For the remaining two apps, "URL Approval Analysis" (Section 4.3.2) on untrusted iframe/popup links faced difficulty in handling constraints that contained string regular expressions. We leave addressing these weaknesses as our future work.

All experiments were run on a high-performance computer. We ran DCV-Hunter with 100 processes in parallel and each process was assigned with limited resources (two regular computing cores and 8GB memory). Our time cost showed that each process needed 144 seconds for each app.

5.1 Manual Verification

To manually verify target apps, we firstly modify Android source code (version 6) to let it print necessary WebView related information. Next, we install the modified Android system in a real device (Nexus 5). Then, we test target apps. For each app, when internal WebView instances are started, we inject attack code to target iframes/popups. Last, based on the web content shown in WebView and the logs printed by Android, we determine if the attack code works and the app is vulnerable.

Please note that different from prior work, we do not use proxy for code injection. We find proxy has several shortcomings. For example, it is time consuming to locate the target iframes/popups for code injection. Instead, we leverage Chrome's USB debug interfaces to ease our test. Since we run test in a real device, we connect the device with PC using USB. Then, we open Chrome in PC to inject code to target WebView instances. For example, we select a WebView instance and then open console (in Chrome) to run extra attack code for code injection. But please always keep in mind that before executing any code, we must select a (target) sub-frame as the code execution environment in console.

Table 5: Potential Vulnerability Details

Potential Attacks	Impacted WebView	Impacted Apps	App Downloads
Origin-Hiding	1,737	1,238	3.5 Billion
WUI Overlap	138	89	8 Billion
WUI Closure	5	5	13 Million
Traditional Navigation	13,384	4,358	19.5 Billion
Privileged Navigation	12,490	4,161	17.8 Billion
Total	13,384	4,358	19.5 Billion

5.2 Findings

Many high-profile apps are impacted by DCVs. DCVs widely exist in hybrid apps. Up to now, the potentially vulnerable apps have been downloaded more than 19.5 Billion times (the fourth column of Table 5). Furthermore, these also include many manually verified popular apps (some examples are shown in Table 6) such as Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, U.S. Bank.

Almost all categories of apps are affected. Figure 10 shows the related distribution data. The light blue line and the bars respectively represent the distribution of potentially vulnerable apps and each potential vulnerability in each category. Almost all categories of apps are impacted, including several sensitive categories (e.g., password management and banking apps). This indicates DCVs are common.

We observe that some categories are more subject to DCV attacks than others, such as news, dating, and food-drink. We manually analyze a set of apps in these categories, and find that these categories of apps use WebView more often to load third-party untrusted content in iframes/popups. For example, the Google News app (one billion+ downloads) provides the news collections to users. It allows any website to be loaded in its WebView. We manually check several news links and find that it is common for these news web pages to embed third-party content, especially ads and tracking services.

We also find that in some apps, their loaded web pages are safe, and do not include any untrusted content. However, after the web pages are fully loaded, these apps run extra JavaScript code through the API "*WebView.evaluateJavascript()*" to created and embedded new iframes/popups for loading ads content, which introduces security risks.

Furthermore, we find that the events and news apps are more likely to suffer from WUI redressing attacks. This is mainly because these apps tend to manage WUIs by themselves. For example, in some news apps, when a user scrolls down to the bottom of the web page, the apps will directly append and show more content, without letting the user click a "next page" button. When the user clicks a concrete news link, a new WUI is created and placed in the front of current WUI to show that link. When the user finishes that web page, developers can close current WUI and show previous WUI. In this way, the state of previous WUI is not changed, and the dynamically appended content is also kept. This render-

Table 6: Summary of Example (Manually Verified) Vulnerable Apps/Libraries
(* can be any domain, while OH, WO, WC, TN, PN, and BA respectively mean Origin-Hiding, WUI Overlap, WUI Closure, Traditional Navigation, Privileged Navigation, and Blended attacks.)

Apps/Libraries	Possible Attack Scenarios		Vulnerabilities						Downloads
	Main-Frame	Untrusted Sub-frame	OH	WO	WC	TN	PN	BA	
Facebook	*	*		✓		✓		✓	1 Billion+
Instagram	*	*		✓		✓		✓	1 Billion+
Facebook Messenger	*	*		✓		✓		✓	1 Billion+
Kakao Talk	*	*		✓		✓		✓	1 Billion+
Google News	*	*				✓	✓		1 Billion+
Skype	*	*				✓	✓		1 Billion+
WeChat	*	*				✓	✓		100 Million+
Yelp	*	*				✓	✓		10 Million+
Kayak	*	*		✓		✓			10 Million+
Uber	uber.com	third-party tracking		✓		✓			100 Million+
ESPN	espn.com	third-party tracking		✓		✓			10 Million+
McDonald's	mcdonalds.com	third-party tracking				✓	✓		10 Million+
Samsung Mobile Print	*	*				✓	✓		5 Million+
lastpass	*	*				✓			5 Million+
dashlane	*	*				✓	✓		1 Million+
1password	*	*				✓	✓		1 Million+
U.S. bank	*	*				✓	✓		1 Million+
Huntington bank	huntington.com	third-party tracking				✓	✓		1 Million+
Chime mobile bank	*	*				✓	✓		1 Million+
Facebook Mobile Browser Library	*	*		✓		✓		✓	
Facebook React Native Library	*	*				✓	✓		

ing strategy improves user experience. However, as described in Section 3.6, due to the design flaws of the event handler system, such a WUI management strategy is also exposed to untrusted iframes/popups, and cause security issues.

Traditional and privileged navigation attacks impact more apps than other DCV attacks. As summarized in the second and third columns of Table 5, navigation based attacks are more popular than the other vulnerabilities. It is mainly because the security assumptions of these two attacks are more easily satisfied. For example, many WebView instances prefer using the default configuration (e.g., disabling popup-creation), and suffer from privileged navigation attacks.

The traditional navigation based attack causes more serious consequences in the context of WebView. This type attack almost affects all potentially vulnerable apps. One important reason is that the effective defense solution "iframe sandbox" is hardly used in practice. There are several reasons. First, it may be difficult to add the sandbox attribute to an iframe, especially considering developers have to find the corresponding web code of that frame from a large amount of web files and code. Second, it is difficult to manage the sandbox configurations for each iframe. Each iframe has its own specific security configurations, including disabling JavaScript or navigation. When the iframe number rapidly rises, the configuration management may become quite difficult. Third, iframe sandbox is not flexible. Its configurations are often bound with iframes, rather than origins. If an iframe is navigated to a different origin, it is hard for developers to update the sandbox restriction policies.

5.3 Case Studies

We have successfully manually launched DCV attacks in many popular apps (some examples are shown in Table 6). Readers can find also several video demos at [2] (the website is anonymized). In this section, we present two example apps (Skype and Kayak) in detail, and also briefly discuss other examples listed in Table 6.

5.3.1 Skype

This is a very popular communication app (one billion+ downloads). Our study shows it suffers from traditional and privileged main-frame navigation attacks. A possible attack scenario is shown in Figure 11. An attacker sends the victim user a message containing a benign but vulnerable link (e.g., ebay.com). When the user clicks the link, a WebView instance is started to render that link (Figure 11-b). However, the loaded web page includes third-party untrusted tracking web content (e.g., double-click) in iframes. The embedded untrusted content has the ability to secretly navigate the main frame through traditional or privileged navigation attacks, which may result in stealthy phishing attacks (Figure 11-d).

We also observe that when a web page is opened, its URL (e.g., ebay.com) is shown in the top of the app. This is relatively helpful to mitigate DCV attacks. However, after the web content is fully loaded by WebView (Figure 11-c), we find the URL is replaced by the title of the loaded web page. After that, the URL will not be shown again, even when a nav-

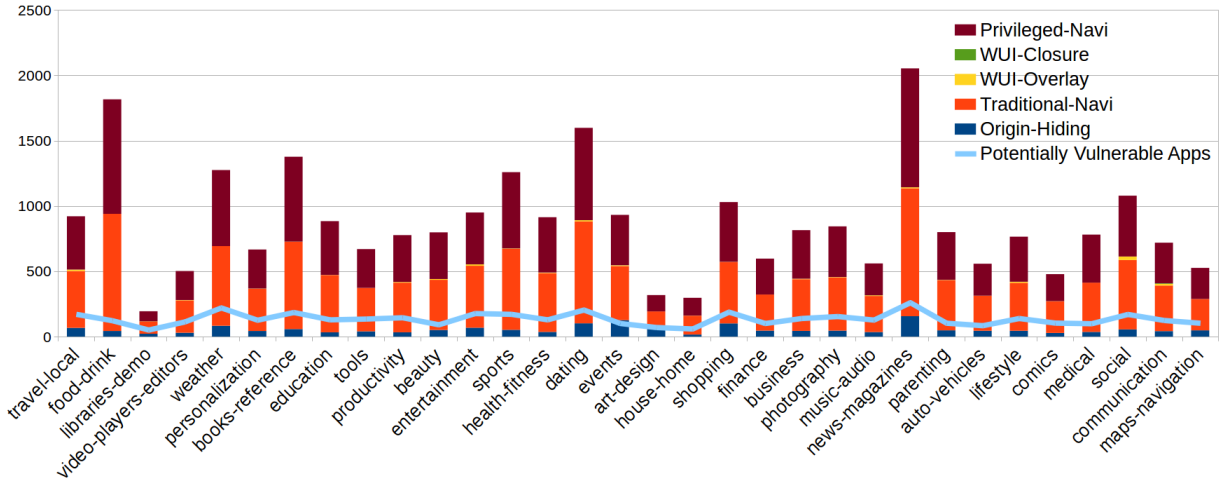


Figure 10: Distribution of Potentially Vulnerable Apps and Potential Vulnerabilities

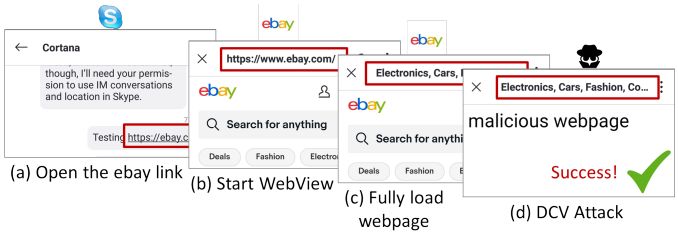


Figure 11: Attacking Skype

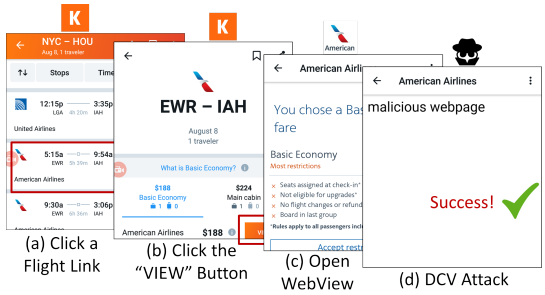


Figure 12: Attacking Kayak

igation event occurs. Hence, when the phishing attack occurs, the victim user may hardly be aware of it.

5.3.2 Kayak

It is a leading app (ten million+ downloads) for providing traveling-relevant searching services, which are aimed to help users find better prices of flights, hotels, rental cars, and so on. However, as shown in Figure 12, it suffers from WebView UI redressing attacks, which may cause account information leakage and financial losses. Consider a possible scenario that a user is searching a flight. The user clicks one of the searching results (Figure 12-a), such as the AA flight, and then clicks the "View" button to get more details (Figure 12-b).

Next, a customized WebView instance is triggered to show more flight details from "aa.com" (Figure 12-c). However, in the AA web page, an extra iframe is embedded to load third-party tracking content (tag management). In the Kayak app, the untrusted iframe obtains the ability of performing phishing attacks by leveraging the WUI overlap issue (Figure 12-d).

In addition, similar with the Skype app, the Kayak app also provides a title bar to reduce the UI inconsistencies. However, this is limited to defend against DCV attacks, since the opened fake web pages often have the same title content.

5.3.3 More Examples

In addition to Skype and Kayak, more examples listed in Table 6 are discussed below.

- *Facebook Mobile Browser, Facebook, Instagram, and Facebook Messenger*: The Facebook Mobile Browser library is frequently used in Android apps, such as Facebook, Instagram, and Facebook Messenger. In our study, the traditional navigation and WUI overlap vulnerabilities exist. As shown in Section 1 and Figure 5, an address bar is provided in the library and is helpful to mitigate DCV attacks. However, as discussed in Section 5.4, the address bar may face pixel and race condition flaws. By leveraging these flaws, untrusted sub-frames can still obtain the ability of launching phishing attacks.
- *Kakao Talk*: Kakao Talk is a popular instant messaging app. Although Kakao Talk is not equipped with the Facebook Mobile Browser library, it is also impacted by the above race condition flaw. Similar with the Facebook Mobile Browser library, the app may still be attacked by untrusted sub-frames through the combination of the traditional navigation and WUI overlap attacks.
- *Google News*: As introduced in Section 5.2, the Google News app can show any news websites. When there is an untrusted sub-frame in the rendered news web page, which is common in practice, the untrusted sub-frame can perform traditional or privileged navigation attacks.
- *WeChat*: WeChat is another popular instant messaging app. Similar with Skype (Section 5.3.1), WeChat also faces traditional and privileged navigation vulnerabilities.

- *Yelp*: The Yelp app are also impacted by traditional and privileged navigation vulnerabilities. Different with Skype and WeChat, Yelp's WebView is triggered by clicking the homepage link of a restaurant or a store. When the opened "homepage" web page contains an untrusted sub-frame, the untrusted sub-frame can launch traditional or privileged navigation attacks.
- *Uber*: Uber's WebView can be started to show "Terms and Conditions" from its own website by sequentially clicking the buttons "menu", "legal" and "terms&conditions". Our analysis shows the term and condition webpage contains an untrusted iframe for loading third-party tracking content (market analyst). The untrusted iframe can launch traditional or privileged navigation attacks.
- *ESPN*: The ESPN app shows news from its own website. However, its web pages load third-party tracking content from Google in an iframe. Hence, the untrusted sub-frame can also do phishing attacks by leveraging traditional navigation and WUI overlap vulnerabilities.
- *McDonald's*: In the app, several events are listed. When an event link (such as "trick n' treat") is clicked, WebView is started to show more details from its own website. However, an untrusted sub-frame is also contained that it may exploit traditional or privileged navigation vulnerabilities.
- *Samsung Mobile Print, lastpass, dashlane*: These apps provide an internal web browser to improve user experience. These internal browsers suffer from main-frame navigation attacks. Although they also offer address bars, unfortunately, the length of their address bars is much short than the average length "29 letters" (Section 5.4. For example, in the same environment (Nexus 5), Samsung Mobile Print only shows 23 letters, and lastpass only display 18 letters.
- *1password*: DCV-Hunter finds several paths to WebView content loading APIs. Because we do not have an account to login, this app is not fully tested. However, when we click its discount link, we still find a vulnerable WebView instance is launched. The WebView instance can show any content, and suffers from traditional or privileged navigation attacks.
- *The U.S., Huntington and Chime Mobile Bank apps*: These bank apps provide WebView to load content from their websites. Note that some of their WebView can be navigated to any websites. The loaded content can include third-party (tracking) content, which can launch traditional or privileged navigation attacks.
- *The Facebook React Native library*: This library is designed to help JavaScript developers implement cross-platform mobile apps. In its WebView, the related default configurations are applied. It suffers from traditional and privileged navigation vulnerabilities.

5.4 Security Impacts of Home-Brewed URL Address Bars

Our study shows that some hybrid apps implement their own URL address and title bars (such as those in our case studies), which could reduce the UI inconsistencies between WebView and regular browsers. To better evaluate the security impacts, we conducted an empirical study of 100 apps that contain home-brewed address bars. These apps are collected by filtering the DCV-Hunter analysis results (by checking if there is a path or flow from WebView's real-time URLs (such as the API "*WebView.getUrl()*" and the second parameter of the event handler "*onPageFinished(view, url)*") to UI components' updating APIs such as "*TextView.setText()*").

We find that the home-brewed address bars are ineffective to prevent DCV attacks, for two main reasons: limited address bar lengths, and implementation errors.

Limited Address Bar Lengths. In our study on a real phone (Nexus 5), which has the representative screen width, we find that typical address bars averagely show 29 letters. When domains, including sub-domains, being accessed exceed that length, security risks could be caused, even when some existing solutions such as showing the rightmost/leftmost of origin/URL are in use (e.g., Chrome/Chromium). This is also partially verified by existing work (e.g., [29]).

Implementation Errors. Some apps/libraries, such as "Facebook Mobile Browser", use very small fonts to show origins (Figure 5). This mitigates the above length limitation problem. As Figure 5-c shows, this address bar can effectively mitigate a DCV attack, such as the WUI overlap attack, since the address bar can show the origin of the fake web page in real time. However, it also has several flaws. First, due to the small font, it faces the pixel problem. Attackers may build a fake and confusing URL by replacing few letters of the benign URL with confusing letters (such as replacing the letter "O" with the number "0"). The fake URL may still spoof users.

Moreover, in these apps, our analysis finds a race condition flaw, which can be utilized to show fake web content in WebView, while still presenting the benign URL (e.g., ebay.com) in the address bar (Figure 5-d). This issue is rooted in the design flaw that several WUIs share only one address bar, while all these WUIs have abilities to update the content of the address bar. Hence, attackers can still perform phishing attacks by combining a couple of DCV attacks. For example, in the Facebook Mobile Browser library, which suffers from the WUI overlap attack, attackers may open a WUI to load fake content, and then immediately update the overlapped benign WUI in background. As a result, the address bar only show attackers' URL in a very short time and is quickly updated to display the benign URL. In our test, we find sometimes the bad URL may not even appear (see our online demo [2]). This indicates the blended attack is stealthy. In practice, the blended attack can be easily launched by using the code shown in Listing 8.

```
// Opening a fake web page (WUI overlap attack)
window.open("https://attacker.com", "_blank")
// Refreshing the address bar (Traditional navigation attack)
window.open("https://eaby.com", "_top")
```

Listing 8: Exploit Code of Blended Attacks

6 Vulnerability Mitigation

6.1 Mitigation Solution

To mitigate DCV attacks, we propose a multi-level solution that enhances the security of WebView. First, we enhance the security of event handlers by addressing their design flaws (Section 3.6). For example, in *onCreateWindow()*, necessary information is provided, including the operator origin who is creating a popup, and the URL the created popup is going to load. Thus, based on the provided information, developers can reject an unauthorized request. To ease the deployment of our solution, we also provide security enforcement. If developers provide the list of trusted URLs in a configuration file inside their apps (located in the app folder "assets"), the untrusted requests can be automatically denied.

Second, we also mitigate the UI inconsistencies by providing floating URL indicators. For example, when the main frame is navigated to a different domain by an iframe/popup, the URL indicator can provide users an alert. Furthermore, when users longly press a WebView instance, the origin of the main frame being loaded by the WebView instance is presented.

Note this URL indicator is locally bound with a WUI, which is helpful to avoid the race condition flaw (Section 5.4). When there are multiple WUIs available, only the foremost WUI's URL indicator is visible.

Third, to mitigate origin-hiding attacks, in critical operations (e.g., accessing web-mobile bridges), we replace the "null" origin with the origin who creates the "null" origin. This makes existing defense solutions effective again, since they can enforce security checks or policies on the new origin.

Fourth, to counter the WebView UI redressing problem, changes of the WUI rendering order are monitored. When a change is performed by an iframe/popup, an alert is offered. Last, to limit the navigation based attacks, we introduce same origin restrictions into navigation, and also fix the conflict.

6.2 Mitigation Solution Implementation

Our implementation is mainly done by instrumenting the WebView library, without modifying the source code of Android frameworks.

6.2.1 Enhanced Event Handlers

To achieve the goal, event handlers related implementation is instrumented. Take the event handler *onCreateWindow()* as the example. To obtain the origin who is creating a popup, the call site is scanned to locate the last popup-creation operation. Next, the corresponding operator's web frame information (e.g., origin) is retrieved. However, if the web frame's origin is "null", DCV-Hunter checks the web frame tree to get the real frame who create the "null" frame. Then, to learn the URL the created popup is going to load, the parameter of the related API (e.g., *window.open()*) is also extracted. Furthermore, to implement the security enforcement of denying untrusted requests, the default implement of *onCreateWindow()* is also instrumented. When the configuration file (providing the list of trusted domains) exists, the trusted URLs are extracted and also used to match the URLs that trigger popup-creation requests.

6.2.2 URL Indicators

To present current origin loaded in a WebView instance, the long-click event of the WebView instance is handled. When the event occurs, the origin of the main frame is presented as a notification. However, the long-click event may also be used by developers. To avoid potential conflicts, we create an event handler wrapper, which first shows the origin information, and then calls the essential event handler registered by developers.

To monitor the main-frame navigation, the event handler "shouldOverrideUrlLoading()" is leveraged. When the event handler is triggered, the URL is checked. If the main frame is redirected to a different domain by a sub-frame, an alert can be given. Furthermore, considering WebView is also a view group (Section 2.2), we make the indicator *local*: we temporary add a text view to WebView as the indicator.

6.2.3 Replacing the "null" Origin

Since the "null" origin is meaningless, we replace it with the origin who creates the "null" origin. To achieve the goal, we scan the frame tree from bottom to top, and get the root frame, or the last frame whose origin is not "null". Then, the corresponding origin *O* is extracted for the replacement.

Next, to replace the "null" origin with *O* in postMessage, we instrument the associated methods of the class "*WebDOMMessageEvent*" and "*MessageEvent*". If the source origin is specified as "null", it will be replaced. Then, the security of web-mobile bridges is enhanced as follows. Take the event handler *onJsAlert(view, url, ...)* as the example. We instrument the event handler's relevant caller (i.e., "*AwJavaScriptDialogManager::RunJavaScriptDialog*") inside WebView. In the caller, if *url* is the data scheme URL, it will be replaced by *O*.

6.2.4 Popup Indicator

To mitigate the WebView UI redressing problem, all associated key APIs are monitored, such as *addView()*. When the WUI rendering order is changing by a sub-frame, an alert will be offered (implemented in the associated enhanced event handlers).

6.2.5 Safe Navigation

To avoid traditional navigation problem, we narrow down the navigation policy that navigation occurs only when two frames have the same origins. To achieve the goal, we instrument the key method "*LocalDOMWindow::open()*" to add the origin checks.

Furthermore, to fix privileged navigation problem, the conflict between WebView features and web APIs is handled. More specifically, in the key method "*RenderFrameHost Impl::CreateNewWindow*", we add more security restrictions. When the setting "*SupportMultipleWindows*" is false, the popup behavior will be ignored.

6.3 Mitigation Evaluation

In our evaluation, we first test the usability of our defense solution, especially about how easy to deploy and apply our solution in practice. To do that, we select 10 real-world vulnerable apps for testing. We find our solution can simply work, if developers involve our own WebView header files, including the declarations of new function prototypes (e.g., *onCreateWindow()*), and also provide the configuration file with the list of third-party domains. Please note that because these real apps lack source code, we repackage them to involve necessary files.

Next, we verify the correctness of our mitigation solution by testing above ten apps. We test them in stock (vulnerable) WebView and the WebView that implements our mitigation solution, respectively. We find that 1) there are no errors introduced by our mitigation solution. Apps work well as usual; 2) DCV attacks are mitigated.

Then, we measure the overhead to check if our mitigation solution impacts user experience. We create a vulnerable app for testing. In the app, we call the WebView API *loadUrl()* to run associate HTML/JavaScript code to trigger all vulnerabilities. Meanwhile, all time costs are recorded. Similarly, we run the app in stock (vulnerable) WebView and the WebView that implements our mitigation solution. By comparing time costs, we find our mitigation solution only introduces tiny overhead: 2ms on average.

Last, considering the Android version fragmentation issue, we also test the compatibility of our mitigation solution by installing our own WebView library and running above the created app in major Android versions. The result shows our solution is available in many major popular Android versions (5.0+), and covers 89.3% of Android devices in use (based on the Android version distribution data of May 2019 [1]).

7 Related Work

Iframe/popup Security. In web apps, iframes/popups are often the cause of security issues, such as frame hijacking [11], clickjacking [43], and double-click clickjacking [23]. In past years, in the context of regular browsers, iframe/popup behaviors and these security issues were well studied. Many defense solutions were proposed. For example, the HTTP header "X-Frame-Options" and the frame busting [43] solution can prevent being framed. In this work, we mainly focus on the exploration of the abilities of untrusted iframes/popups. The more related security mechanisms, such as SOP, and navigation policies, are discussed in Section 2.1. As shown in Section 1 and 3, existing solutions are circumscribed to prevent DCV attacks.

WebView security. WebView security has attracted more and more attention. [17, 30, 33] generically studied WebView security. [21, 25, 27, 40, 49, 53] explored the security of web-mobile bridges, and also discovered several extended attacks. In Section 3.5, we compare DCV attacks with several related attacks, and show DCV attacks may have a set of advantages.

Several static analysis based approaches [22, 55] were proposed to vet hybrid apps. However, they were limited to analyze iframe/popup behaviors and event handlers (also see our discussion in Section 4). Several defense solutions were designed to provide protection for WebView and web-mobile bridges, such as NoFrak [21], Draco [49], MobileIFC [45], WIREframe [18], and HybridGuard [38]. NoFrak and MobileIFC extended SOP into the mobile layer, while other solutions provided security enforcement on web-mobile bridges. However, as discussed in Section 1 and 3, they were quite limited to prevent DCV attacks.

In addition, many solutions [13, 41] are also designed to mitigate the Android UI deception problems [15, 20, 35]. However, as discussed in Section 1 and 3.3, they cannot monitor the state change of WebView UI, and circumscribed to prevent WUI redressing attacks.

8 Discussion

Research scope. In this work, we mainly focus on Android, which is currently the most popular mobile OS. However, there are also other WebView formats in other platforms (e.g., WKWebView for iOS). The research on other platforms would be complementary to our work, and we leave this as our future work.

False negatives. Like any static analysis based approach, our DCV-Hunter inevitably has false negatives in some situations. For example, in mobile apps, some URLs loaded in WebView are encrypted, some WebView related code is dynamically loaded, and some URL related data goes through implicit flows. We leave the improvement of our tool to reduce false negatives as our future work.

9 Conclusion

Iframes/popups are often the root cause of several critical web security issues, and have been well studied in regular browsers. However, they are rarely understood and scrutinized in WebView, which has a totally new working environment. In this paper, we fill the gap and identify several fundamental design flaws and vulnerabilities, named differential context vulnerabilities (DCVs). We find that by exploiting DCVs, an untrusted iframe/popup becomes very dangerous in Android WebView. We have designed a novel detection technique, DCV-Hunter, to assess the security impacts of DCVs on real-world apps. Our measurement on a large number of popular apps shows that DCVs are prevalent. We have also presented a multi-level protection solution to mitigate DCVs, which is shown to be scalable and effective.

Acknowledgments

We want to thank our shepherd Yinzhi Cao and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation (NSF) under Grant no. 1642129 and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. We also thank Cong Zheng and Yuchen Zhou for the helpful discussions about our threat model and the design of DCV-Hunter.

References

[1] Android version distribution dashboard. https://developer.android.com/about/dashboards.

[2] Dcv-attacks. https://sites.google.com/view/dcv-attacks.

[3] Easyprivacy tracking protection list. https://easylist.to/tag/tracking-protection-lists.html.

[4] iframe - html standard. https://html.spec.whatwg.org/dev/iframe-embed-object.html#attr-iframe-sandbox.

[5] Mcafee mobile threat report. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf.

[6] Same origin policy. https://en.wikipedia.org/wiki/Same-origin_policy.

[7] Selenium - web browser automation. https://www.seleniumhq.org.

[8] Web messaging standard. https://html.spec.whatwg.org/multipage/web-messaging.html.

[9] Webview client. https://developer.android.com/reference/android/webkit/WebViewClient.html.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[11] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *USENIX Security*, 2009.

[12] A. B. Bhavani. Cross-site Scripting Attacks on Android WebView. *IJCSN International Journal of Computer Science and Network*, 2(2):1–5, 2013.

[13] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *IEEE Symposium on Security and Privacy*, 2015.

[14] T. Bujlow, V. Carela-Español, J. Solé-Pareta, and P. Barlet-Ros. A survey on web tracking: Mechanisms, implications, and defenses. *Proceedings of the IEEE*, 2017.

[15] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security*, 2014.

[16] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications*, 2013.

[17] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *WISA*. 2013.

[18] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha. Secure integration of web content and applications on commodity mobile operating systems. In *ASIA CCS*, 2017.

[19] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS, pages 337–340. Springer-Verlag, 2008.

[20] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *IEEE Symposium on Security and Privacy*, 2017.

[21] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS*, 2014.

[22] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang. Web-to-application injection attacks on android: Characterization and detection. In *ESORICS*, 2015.

[23] L. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[24] InfoSecurity. Public wifi hotspots ripe for mitm attacks. https://www.infosecurity-magazine.com/news/public-wifi-hotspots-ripe-for-mitm-attacks/.

[25] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *CCS*, 2014.

[26] A. Lerner, T. Kohno, and F. Roesner. Rewriting history: Changing the archived web from the present. CCS, 2017.

[27] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *CCS*, 2017.

[28] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *CCS*, 2012.

[29] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. CCS, 2017.

[30] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*, 2011.

[31] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, iOS, and windows phone. In *Foundations and Practice of Security*. 2013.

[32] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy*, 2012.

[33] P. Mutchler, A. DoupÃ, J. Mitchell, C. Kruegel, G. Vigna, A. Doup, J. Mitchell, C. Kruegel, and G. Vigna. A Large-Scale Study of Mobile Web App Security. In *MoST*, 2015.

[34] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: Webview exploitation. In *LEET*, 2013.

[35] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. *Black Hat*, 2012.

[36] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. CCS, 2012.

[37] X. Pan, Y. Cao, and Y. Chen. I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser. In *NDSS*, 2015.

[38] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. In *MoST*, 2017.

[39] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. Usenix Security, 2008.

[40] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces. *NDSS*, 2016.

[41] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, 2015.

[42] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *NSDI)*, 2012.

[43] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *IEEE Oakland Web 2.0 Security and Privacy*, 2010.

[44] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[45] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *RAID*. 2013.

[46] D. F. Somé, N. Bielova, and T. Rezk. Control what you include! - server-side protection against third party web tracking. In *Engineering Secure Software and Systems*, 2017.

[47] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[48] K. Tian, Z. Li, K. D Bowers, and D. Yao. Framehanger: Evaluating and classifying iframe injection at large scale. In *SecureComm*, 2018.

[49] 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 *CCS*, 2016.

[50] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS*, 2013.

[51] T. Wei, Y. Zhang, H. Xue, M. Zheng, C. Ren, and D. Song. Sidewinder targeted attack against android in the golden age of ad libraries. In *Black Hat*. 2014.

[52] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. In *USENIX Security*, 2015.

[53] G. Yang, J. Huang, and G. Gu. Automated generation of event-oriented exploits in android hybrid apps. In *NDSS*, 2018.

[54] G. Yang, J. Huang, G. Gu, and A. Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *IEEE Symposium on Security and Privacy*, 2018.

[55] G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In *RAID*, 2017.

[56] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna. The dark alleys of madison avenue: Understanding malicious advertisements. IMC, 2014.