WHEN WEB MEETS MOBILE: NOVEL SECURITY THREATS AND DEFENSES IN WEB/MOBILE HYBRID APPS

A Dissertation
by
GUANGLIANG YANG

Submitted to the Office of Graduate and Professional Studies of Texas A&M University in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY

Chair of Committee, Guofei Gu
Committee Members, Jyh-Charn Liu
Radu Stoleru
Narasimha Reddy
Head of Department, Dilma Da Silva

August 2019

Major Subject: Computer Science

Copyright 2019 Guangliang Yang
ABSTRACT

Nowadays, mobile app developers are enjoying the benefits of the amalgamation of web and mobile platforms. Developers can easily and smoothly integrate all sorts of web services in their mobile apps by embedding a browser-like UI component, called “WebView”, which can render web content and run JavaScript code within mobile apps (call hybrid apps for convenience). WebView is easy to use and popular. A recent study showed ~80% of Android apps used WebView. WebView is also as powerful as regular browsers (e.g., Chrome/Chromium), and well supports web features and behaviors.

In regular browsers, there exist several sensitive web behaviors that are often the root reason of critical security issues. In past years, they have been well studied, and a variety of mature defense solutions have been deployed. However, these sensitive web behaviors are seldom understood and scrutinized in WebView, which provides a totally new working environment. Different from regular browsers, WebView offers mobile developers freedom to customize their WebView instances by enabling several unique programming features. For example, WebView allows mobile code to control and customize web behaviors through WebView setting and event handler APIs. Considering these WebView features may heavily impact above sensitive web behaviors, it is unclear whether the corresponding defense solutions are still effective in WebView.

Motivated by above security concerns, in this dissertation, we conduct the systematic security study of several sensitive web behaviors (e.g., web events, web messaging, and the utilization of iframes and popups) in WebView of the Android platform, which is open and the biggest mobile operating system (OS). As a consequence, we discover several novel security vulnerabilities and fundamental design flaws. To demonstrate the security implications, we devise several concrete attacks. Through these attacks, untrusted code (e.g., ads) loaded in WebView can open holes on existing defense solutions, and obtain risky privileges and abilities, such as stealing users’ private data (e.g., GPS location), unauthorizedly accessing sensitive hardware (e.g., microphone), and performing phishing attacks. Then, we study and assess the security impacts of these security issues on real-world hybrid apps. For this purpose, we develop novel tools that can automatically apply program analysis techniques to vet Android apps. By analyzing a large number of most popular apps...
collected from the official Android marketplace, we find the vulnerabilities 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 these security issues from the root, we design multi-level defense solutions that enhance the security of WebView. Our evaluation on real-world apps shows our mitigation solutions are effective and scalable, with negligible overhead.
To my beloved family

for their endless love, support, and encouragement
I owe my heartfelt gratitude to my advisor, Dr. Guofei Gu, for his extraordinary guidance, support, patience, understanding, tolerance, and encouragement throughout my whole Ph.D study. In particular, he guided me to mature as a researcher and as a person, was always being there whenever I need his help, gave me freedom for developing and pursuing my own research interests, and encouraged me to pursue the academic career after graduation. He has had a significant influence on my professional development. I am so fortunate of having him as my advisor, and I will cherish the time of working with him. I would also like to thank other members of my committee, Dr. Jyh-Charn (Steve) Liu, Dr. Radu Stoleru, and Dr. Narasimha Reddy for their insightful and constructive comments and suggestions of improving my research and the dissertation.

I am also very grateful to Dr. Jeff Huang. Thank him for sharing his optimism and enthusiasm, inspiring me to pursue creativity, and improving my writing skills. Without his help, I cannot find the right way of fully developing my research skills, and publishing papers in top-tier venues.

I would like to thank my colleagues and my friends. I enjoy working with all SUCCESS lab members. Thank Chao Yang, Zhaoyan Xu, Jialong Zhang, Lei Xu, Abner Mendoza, Phakpoom Chinprutthiwong, and Jin Hao, for their comments, suggestions and help in my research and in my life. I would also like to express my appreciation and gratitude to Zheng Cong (from Palo Alto Networks) for his continuous support and encouragement. Their friendships are precious.
CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Guofei Gu (advisor), Jyh-Charn Liu and Radu Stoleru of the Department of Computer Science and Engineering, and Professor Narasimha Reddy of the Department of Electrical and Computer Engineering.

The paper organization and writing for Chapter 3, 4 and 5 were revised by Professor Jeff Huang of Computer Science and Engineering. The overview story for Chapter 4 was revised by Abner Mendoza.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

This work was made possible in part by National Science Foundation (NSF) under Grant no. 1617985, 1642129, 1740791, and 1700544, Air Force Office of Scientific Research (AFOSR) under FA-9550-13-1-0077 and a Google Faculty Research award.

The content is solely the responsibility of the authors and do not necessarily represent the official views of the NSF, AFOSR and Google.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>iv</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>CONTRIBUTORS AND FUNDING SOURCES</td>
<td>vi</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF LISTINGS</td>
<td>xiv</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Web Events</td>
<td>3</td>
</tr>
<tr>
<td>1.3 postMessage</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Iframes/Popups</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Dissertation Contributions and Organization</td>
<td>6</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Android</td>
<td>7</td>
</tr>
<tr>
<td>2.2 WebView</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Event Handlers</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Iframes/Popups and Related Protection</td>
<td>8</td>
</tr>
<tr>
<td>2.5 Inconsistencies Between Regular Browsers and WebView</td>
<td>9</td>
</tr>
<tr>
<td>2.6 postMessage and Hybrid postMessage</td>
<td>11</td>
</tr>
<tr>
<td>3. STUDYING AND PROTECTING WEB EVENTS</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>13</td>
</tr>
<tr>
<td>3.1.1 Motivation</td>
<td>13</td>
</tr>
<tr>
<td>3.1.2 Security Issues</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3 Vulnerability Detection</td>
<td>15</td>
</tr>
<tr>
<td>3.1.4 Evaluation</td>
<td>17</td>
</tr>
</tbody>
</table>
3.2 Problem Statement ................................................................. 17
3.2.1 Motivating Example ......................................................... 17
3.2.2 Threat Model ................................................................. 19
3.2.3 Security Issues .............................................................. 19
3.2.4 Problem Definition .......................................................... 20
3.2.5 Critical Functionalities ...................................................... 20
  3.2.5.1 URL Loading API ....................................................... 20
  3.2.5.2 Source and Sink APIs ................................................ 21
  3.2.5.3 APIs Accessing Local Resources .................................... 21
  3.2.5.4 APIs Sending Intent messages ....................................... 21
3.3 System Overview ................................................................. 22
3.3.1 Overview ....................................................................... 22
3.3.2 Analyzing the Example ..................................................... 23
3.4 Technical Approaches ......................................................... 25
  3.4.1 Understanding Event Handler Triggering Constraints ................. 27
  3.4.1.1 Event Triggering Constraints ........................................ 27
  3.4.1.2 Event Handler Triggering Constraints .............................. 28
  3.4.1.3 Adversaries’ Capability: Playing Gadgets .......................... 29
  3.4.2 Selective Symbolic Execution ........................................... 29
  3.4.2.1 Heuristic Generation .................................................. 31
  3.4.2.2 Analysis Sandbox .................................................... 32
  3.4.2.3 Intent Handler ........................................................ 32
  3.4.2.4 Array-Indexing Type Implicit Flow ................................ 33
  3.4.3 Program State Analysis .................................................. 34
  3.4.3.1 Event Handler Input Generation .................................... 34
  3.4.3.2 Event Handler Execution Order Generation ....................... 34
  3.4.4 Exploit Code Generation ............................................... 35
  3.4.4.1 JavaScript Code Syntax Analysis ................................... 35
3.5 Evaluation ................................................................. 37
  3.5.1 Dataset ................................................................. 38
  3.5.2 Findings ................................................................. 38
  3.5.2.1 Usage of Event Handlers ............................................ 38
  3.5.2.2 EOE in Event Handlers .............................................. 39
  3.5.3 Case Studies ............................................................ 41
  3.5.3.1 Case 1: Cross-Frame DOM Manipulation .......................... 41
  3.5.3.2 Case 2: Leveraging a Closed Backdoor ........................... 42
  3.5.4 Performance and Accuracy .............................................. 43
  3.6 EOE Countermeasure Discussion ........................................ 44
3.7 System Limitations and Future Work ..................................... 44
3.8 Summary ........................................................................ 44

4. STUDYING AND PROTECTING HYBRID POSTMESSAGES ................. 45
  4.1 Overview .................................................................. 45
  4.2 The OSV Problem Definition .............................................. 50
5. STUDYING AND PROTECTING IFRAMES/POPUPS ........................................ 73

5.1 Overview ......................................................... 73
  5.1.1 Motivation .................................................. 73
  5.1.2 Inconsistencies Between Browsers and WebView .................. 73
  5.1.3 Differential Context Vulnerabilities (DCV) ...................... 74
  5.1.4 DCV-Hunter & Findings .................................. 76
  5.1.5 DCV Mitigation ............................................ 78
5.2 Threat Model .................................................. 78
5.3 Differential Context Vulnerabilities ................................ 80
  5.3.1 Study Overview ........................................... 80
  5.3.2 Origin Hiding Attacks ..................................... 82
    5.3.2.1 Attacking Web Messaging .............................. 82
    5.3.2.2 Accessing Web-Mobile Bridges ...................... 83
  5.3.3 WebView UI Redressing Attacks .............................. 84

4.3 OSV-Hunter Design and Implementation .................................... 51
  4.3.1 Design Observations ....................................... 51
  4.3.2 Design Details ............................................. 53
    4.3.2.1 Hybrid postMessage Identification ....................... 53
    4.3.2.2 Message Origin Analysis ............................... 55
  4.3.3 Implementation ............................................. 56
4.4 Study of Hybrid postMessage and OSV .................................... 56
  4.4.1 Data Set .................................................. 56
  4.4.2 Results .................................................... 57
  4.4.3 Findings ................................................... 59
  4.4.4 Case Studies .............................................. 60
    4.4.4.1 Facebook React Native ................................. 60
    4.4.4.2 Google Cloud Print ................................... 64
4.5 The Mitigation Solution : OSV-Free APIs .................................. 65
  4.5.1 Goals ........................................................ 65
  4.5.2 Overview ................................................... 66
  4.5.3 Design and Implementation .................................. 67
    4.5.3.1 postMessageToMainFrame() .............................. 68
    4.5.3.2 receiveMessageFromMainFrame() and
             postMessageToNativeFrame() ......................... 68
  4.5.4 Evaluation .................................................. 69
    4.5.4.1 Performance ........................................... 69
    4.5.4.2 Effectiveness ........................................... 69
    4.5.4.3 Compatibility .......................................... 70
    4.5.4.4 Case Study : Patching the Facebook React Native
             Framework ......................................... 70
4.6 Discussion .................................................... 71
  4.6.1 OSV-Hunter’s goal .......................................... 71
  4.6.2 OSV-Hunter’s weakness .................................... 71
  4.6.3 Other ways to defend against V4 .................................... 71
4.7 Summary ....................................................... 71

ix
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The Architecture of Hybrid Apps</td>
<td>3</td>
</tr>
<tr>
<td>3.1 Event-Oriented Exploits</td>
<td>14</td>
</tr>
<tr>
<td>3.2 The Design of EOEDroid</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Vulnerable Code from a Real-World App</td>
<td>18</td>
</tr>
<tr>
<td>3.4 Intent in Inter-Apps Communications</td>
<td>30</td>
</tr>
<tr>
<td>3.5 The Internal Structure of HashMap</td>
<td>31</td>
</tr>
<tr>
<td>3.6 AST of $I + J$</td>
<td>37</td>
</tr>
<tr>
<td>3.7 Usage of Event Handlers</td>
<td>39</td>
</tr>
<tr>
<td>4.1 Overview of Regular and Hybrid postMessage</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Sending Messages Through Regular and Hybrid postMessage</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Receiving Messages Through Regular and Hybrid postMessage</td>
<td>46</td>
</tr>
<tr>
<td>4.4 Communication Among Three Frames</td>
<td>49</td>
</tr>
<tr>
<td>4.5 Attacks on $V_2$</td>
<td>51</td>
</tr>
<tr>
<td>4.6 OSV-Hunter’s Workflow</td>
<td>51</td>
</tr>
<tr>
<td>4.7 Hybrid postMessage in Facebook React Native</td>
<td>60</td>
</tr>
<tr>
<td>4.8 OSV-Free’s Design</td>
<td>68</td>
</tr>
<tr>
<td>5.1 UI Comparison</td>
<td>76</td>
</tr>
<tr>
<td>5.2 Attacking the Huntington Bank App</td>
<td>76</td>
</tr>
<tr>
<td>5.3 The Scenario of Attacking the Huntington Bank App</td>
<td>77</td>
</tr>
<tr>
<td>5.4 Attacking Facebook Messenger</td>
<td>78</td>
</tr>
<tr>
<td>5.5 Attacking Web Messaging</td>
<td>82</td>
</tr>
</tbody>
</table>
5.6 Freely Accessing Web-Mobile Bridges ................................................. 83
5.7 WebView UI Redressing Attacks .......................................................... 84
5.8 The Overview of DCV-Hunter ................................................................. 88
5.9 Distribution of Potentially Vulnerable Apps and Potential Vulnerabilities .... 96
5.10 Attacking Skype ....................................................................................... 97
5.11 Attacking Kayak ..................................................................................... 98
## LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Summary of Novel Vulnerabilities</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Iframe/Popup-Related Programming Features</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>The Systematic Study Result. The third column ‘JS?’ means: ‘Is JavaScript required to trigger the event?’ , and the forth column ‘E0?’ means: ‘Does the event handler only deal with events from E0?’ . In answers, we use ✓ and blank to indicate ‘Yes’ and ‘No’, respectively.</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Vulnerabilities Found by EOEDroid</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>The Input Format of the Two Vulnerable Apps Shown in Case 1</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Definitions of Four Sub-Types of OSV</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>The Evaluation Result</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>OSV-Free APIs</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>The Performance of OSV-Free APIs</td>
<td>70</td>
</tr>
<tr>
<td>5.1</td>
<td>A Summary of Differential Context Vulnerabilities (DCV)</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Source and Sink APIs</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>APIs for the Analysis of WUI Redressing Problems</td>
<td>93</td>
</tr>
<tr>
<td>5.4</td>
<td>Potential Vulnerability Details</td>
<td>94</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary of Example (Manually Verified) Vulnerable Apps/Libs (* 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.)</td>
<td>95</td>
</tr>
</tbody>
</table>
# LIST OF LISTINGS

## LISTING

### Page

<table>
<thead>
<tr>
<th>LISTING</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Usage of postMessage</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Path Constraints in Executing getId()</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Exploit Code</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Code Snippet Extracted From the Example in Case 2</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Implementing W→N in Facebook React Native</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Example Code of Creating a WebView Object in DJ</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Sending Messages to the Main Frame Through WebView.loadUrl() in the Native Context</td>
<td>61</td>
</tr>
<tr>
<td>4.4 Example Attack Code to Let Apps Forcely Join Any Events</td>
<td>62</td>
</tr>
<tr>
<td>4.5 Code Snippet of onMessage() in Case1</td>
<td>63</td>
</tr>
<tr>
<td>4.6 Code Snippet of onMessage() in Case2</td>
<td>64</td>
</tr>
<tr>
<td>4.7 The Source Code of Registering a Message Handler in Google Print</td>
<td>64</td>
</tr>
<tr>
<td>4.8 Source Code of the Message Handler in Google Print</td>
<td>65</td>
</tr>
<tr>
<td>4.9 The Prototype of onMessage</td>
<td>67</td>
</tr>
<tr>
<td>5.1 Validating the Message Origin in the Victim Sub-frame</td>
<td>82</td>
</tr>
<tr>
<td>5.2 Accessing the Event Handler onJsAlert() in the Untrusted Iframe/Popup</td>
<td>83</td>
</tr>
<tr>
<td>5.3 Vulnerable onCreateWindow()</td>
<td>84</td>
</tr>
<tr>
<td>5.4 Exploit Code of the WUI Overlap Attack and the Privileged Navigation Attack</td>
<td>85</td>
</tr>
<tr>
<td>5.5 Vulnerable onCloseWindow()</td>
<td>85</td>
</tr>
<tr>
<td>5.6 Exploit Code of the WUI Closure Attack</td>
<td>86</td>
</tr>
</tbody>
</table>
5.7 Leveraging Traditional Navigation Policies ................................................................. 86
5.8 Exploit Code of Blended Attacks ................................................................................ 100
1. INTRODUCTION

1.1 Overview

Nowadays, mobile app developers are enjoying the benefits of the amalgamation of web and mobile platforms. Developers can easily and smoothly integrate all sorts of web services in their mobile apps by embedding a browser-like UI component, called “WebView” [1], which can render web content and run JavaScript code within mobile apps. WebView is easy to use and popular. A recent study [2] showed ~80% of Android apps used WebView. WebView is also as powerful as regular browsers (e.g., Chrome/Chromium), and well supports web features and behaviors. Thus, through WebView, developers can also easily and smoothly re-use existing web code in their apps. For convenience, in this dissertation, we refer to these WebView-enabled mobile apps as hybrid apps.

In the context of regular browsers, there exist several sensitive web behaviors that are often the root reason of critical security issues (e.g., frame hijacking [3] and clickjacking [4, 5]). In past years, they have been well studied, and a variety of mature defense solutions have been deployed, such as Same Origin Policy (SOP) [6], HTML5 iframe sandbox [7], and navigation policies [3]. However, these sensitive web behaviors are seldom understood and scrutinized in the context of WebView, which provides a totally new working environment. Different from regular browsers, WebView offers mobile developers freedom to customize their WebView instances by enabling several unique UI and programming features. For example, as shown in Figure 1.1, WebView enables web/mobile (cross-platform) bridges that link web and mobile layers. Through the event handler bridge, mobile code can control and handle web events (e.g., link clicking). WebView also provides the setting APIs for developers to configure their WebView instances (e.g., disabling the execution of JavaScript code). Considering the customization of WebView instances may heavily impact sensitive web behaviors, it is unclear whether the corresponding defense solutions are still effective in such a new context of WebView. In this dissertation, we use the term “context” to refer to a web environment that includes GUI elements (e.g., the address and tab bars), corresponding web/mobile APIs (e.g., the setting APIs in WebView), and security defense solutions (e.g., SOP and navigation policies).
Table 1.1: Summary of Novel Vulnerabilities

<table>
<thead>
<tr>
<th>Critical Behaviors</th>
<th>Contexts</th>
<th>Attacks</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Events</td>
<td>Browsers and web code</td>
<td>Mobile code</td>
<td>Event-Oriented Exploits, EOE (Chapter 3)</td>
</tr>
<tr>
<td>postMessage</td>
<td>Regular (safe) postMessage</td>
<td>Hybrid postMessage</td>
<td>Origin Stripping Vulnerabilities, OSV (Chapter 4)</td>
</tr>
<tr>
<td>Iframes/popups</td>
<td>Complete UI</td>
<td>Incomplete UI and conflicted programming APIs</td>
<td>Differential Context Vulnerabilities, DCV (Chapter 5)</td>
</tr>
</tbody>
</table>

Motivated by above security concerns, in this dissertation, we conduct the systematic security study of several sensitive web behaviors in the WebView context of the Android platform, which is open and the biggest mobile OS. These target web behaviors include:

- **web events**: URL clicking and navigation, alert events and so on
- **web messaging (i.e., postMessage [8])**: relaxing SOP and allowing the communication between different web frames
- **the utilization of iframes and popups**: creating and closing iframes/popups

As a consequence, we discover several novel security vulnerabilities and fundamental design flaws (summarized in Table 1.1). To demonstrate the security implications, we devise several concrete attacks. We find when there is untrusted code (e.g., ads) loaded in WebView, which is very common in practice, the untrusted code can launch these attacks to open holes on existing defense solutions, and obtain risky privileges and abilities, such as stealing users’ private data (e.g., GPS location), unauthorizedly accessing sensitive hardware (e.g., microphone), and performing phishing attacks.

Then, we study and assess the security impacts of these security issues on real-world hybrid apps. For this purpose, we develop novel tools that can automatically apply program analysis techniques to vet Android apps. By analyzing a large number of most popular apps collected from the official Android marketplace, we find the vulnerabilities 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 these security issues from the root, we design multi-level defense...
solutions that enhance the security of WebView. Our evaluation on real-world apps shows our mitigation solutions are effective and scalable, with negligible overhead.

In the remaining of this chapter, we present the overview of our research on each sensitive web behavior, with the summary of the dissertation contributions and organization.

1.2 Web Events

Web events are frequently used and faced by web code. However, as shown in Table 1.1, they are differently handled by regular browsers and WebView. In regular browsers, they are dealt with by browsers and web code. In contrast, in WebView, they are often handled by mobile code through the event handler mechanism, a type of web/mobile bridges (see more details in Chapter 2.3). For example, in hybrid apps, when an alert dialog (i.e., alert event) is popped up, developers can use the corresponding event handler API to obtain a chance of redrawing the dialog. When a custom url (e.g., tel:800xxx) is clicked (i.e., url clicking event), developers can leverage the corresponding event handler API to parse the clicked url content, and well handle it by opening the corresponding app for making a phone call.

However, we find the event handler bridge also introduces security risks. The root reason is there is no protection on the event handler bridge, which means any web code loaded in WebView can access event handlers (mobile code). Thus, when event handlers carries sensitive functionalities (e.g., reading location information, and accessing camera) in the mobile layer, and WebView loads
untrusted code (e.g., third-party tracking code) in the web layer, the untrusted code can trigger web events in the web layer, further call event handler in the mobile layers, and finally access the sensitive functionalities inside event handlers. In this dissertation, we refer to this type of attacks as Event-Oriented Exploits (EOEs).

Our study shows the EOE issue causes serious consequences, such as sensitive information leakage, and cross-frame manipulation. What is worse is EOE based attacks cannot be defended against by all existing defense solutions (e.g., [9] [10]). To further evaluate the security impacts of the EOE issue, we design a novel vulnerability detection tool, called EOEDroid, that applies the selective symbolic execution technique to vet Android apps. By applying EOEDroid on real-world apps, we find 97 total vulnerabilities in 58 apps. Base on our study results, we explore potential defense solutions.

1.3 postMessage

WebView inherits the defense solutions from regular browsers, including SOP. However, SOP also introduces side effects: it does not allow the communication between different distrusted web frames. In HTML5, this problem is mitigated by the postMessage mechanism and its corresponding security model.

WebView also supports the postMessage mechanism. However, we find in hybrid apps, postMessage does not meet app developers’ requirements. Developers does want the communication not only between different web frames, but also between web frames and the mobile layer. To mitigate the problem, we find developers extend existing postMessage (i.e., regular postMessage) as a new communication channel between the web and mobile layers. However, due to the strong isolation between the web and mobile layers, the extended postMessage (i.e., hybrid postMessage) cannot follow the security model designed for regular postMessage. In particular, all origin information (e.g., the message sender and receiver) of messages sent through hybrid postMessage is lost. Thus, any web code loaded in WebView can freely accept the message sent by the mobile layer without any limitation or origin validation. If the message contains sensitive information, serious consequences may be caused. In this dissertation, we refer to the security issues as Origin Stripping Vulnerabilities (OSVs).

To evaluate the prevalence and presence of hybrid postMessage and OSVs in Android hybrid
apps, we design a lightweight detection tool, called OSV-Hunter, that can help developers and analysts identify hybrid postMessage and discover potential OSVs. Our analysis on real-world apps shows 74 apps implement hybrid postMessage, and all these apps suffer from OSV, which may be exploited by remote attackers to perform denial of service (DoS), local critical hardware device access (such as real-time microphone monitoring), data race, internal data manipulation, and so on.

To mitigate the OSV issue, we design and implement a set of new hybrid postMessage APIs in the newest WebView, called OSV-Free. Our evaluation shows that OSV-Free is secure and fast, and it is generic and resilient to the notorious Android fragmentation problem. We also demonstrate that OSV-Free is easy to use. OSV-Free is open source, and its source code and more implementation and evaluation details are available online: http://success.cse.tamu.edu/lab/osv-free.php.

1.4 Iframes/Popups

Iframes/popups are frequently used by web code, for example, to show different formats of files (e.g., video and pictures), and load untrusted content (e.g., ads). Our study shows iframes/popups, which are well studied and protected in regular browsers, are dangerous in WebView. Different from other web behaviors (e.g., web events and postMessage), iframes/popups are impacted not only by WebView's programming features (e.g., web/mobile bridges), but also by WebView UI features. More specifically, different from regular browsers, who have complete UI components (e.g., address, status, and title bars), WebView UI is designed in a simple style. Only one area for rendering web content is included in WebView. Hence, if there is an iframe/popup (i.e., untrusted iframe/popup) that carries untrusted content inside WebView, and the untrusted iframe/popup can navigate the main or top frame of WebView, phishing attacks occur.

Our systematic study on Android WebView uncovers a novel class of vulnerabilities (referred to as Differential Context Vulnerabilities or DCV) and design flaws associated with iframe/popup behaviors in WebView. To assess their security impacts on real-world apps, we develop a novel vulnerability detection tool, called DCV-Hunter, for automatically vetting given apps against DCV. Then, by applying DCV-Hunter on a number of most popular apps, we show that DCV are prevalent. More specifically, we find 30.4% of 11,341 hybrid apps are potentially vulnerable, including 9,770 potentially vulnerable WebView instances and 18,459 potential vulnerabilities. Up to now, the
potentially impacted apps have been downloaded more than 23 billion times in total. Furthermore, our evaluation shows DCV-Hunter is scalable and effective, and has relatively low false positives. To mitigate DCV, we propose a multi-level protection solution by enhancing the security of WebView programming and UI features. Our evaluation on real-world apps shows that our solution is effective and scalable, and introduces negligible overhead.

1.5 Dissertation Contributions and Organization

In this dissertation, our contributions include:

- We conduct the systematic security study on several sensitive web behaviors in the context of WebView, and discover several novel security issues and design flaws in Android WebView.

- To demonstrate their security impacts, we devise several novel possible attacks. We find all of these attacks cannot be prevented by existing defense solutions, and cause serious consequences. We show by leveraging these attacks, remote attackers can make several nefarious actions, including stealing users’ private data (e.g., GPS), unauthorizedly accessing sensitive hardware (e.g., microphone), and performing phishing attacks.

- We develop novel vulnerability detection approaches, and apply them on a large number of real-world popular apps to assess the security impacts of these security issues. We find these issues are prevalent, and impact many high-profile apps, such as Facebook and Instagram.

- To mitigate the new problems from the root, we design and implement several novel and effective protection solutions.

The remainder of the dissertation is organized as follows. In Chapter 2, we first introduce necessary background information. Next, we present three security issues in Android WebView with their assessment and mitigation in Chapter 3-5. Then, we discuss related work in Chapter 6 and lessons learned and best WebView integrity practice in Chapter 7. Finally, we conclude the dissertation in Chapter 8.
2. BACKGROUND

2.1 Android

Android apps are typically written in Java and compiled to Dalvik bytecode [11]. At runtime, bytecodes are interpreted and executed by Dalvik virtual machine (DVM) [12]. Generally, an app consists of four components: activity (i.e., the user interface), background service, content providers (i.e., database), and Android native event receivers. Intent can be used in interactions among components and apps.

2.2 WebView

WebView is an embedded UI component used to render web pages and run JavaScript code within mobile apps. Android WebView is equipped with the newest kernel of the regular browser “Chrome/Chromium”, and performs as powerful as regular browsers. WebView is customizable. As shown in Figure 1.1, WebView provide several APIs to let developers customize their WebView instances. In particular, developers can configure WebView through the settings APIs, such as enabling JavaScript, and the access of local files, database, and GPS location. Developers can also use the content loading APIs (e.g., loadUrl()) to load web content into their WebView instances. Developers can also leverage web/mobile bridges to enable the interactions between web and mobile layers. Web/mobile bridges mainly have two formats: JavaScript bridges and event handlers. More details are presented below.

WebView allows JavaScript Bridge, which provides a channel linking web code with native code. More specifically, apps can run the API “addJavascriptInterface(O, N)” to import a Java object O to the JavaScript context. Then, O can be directly accessed by JavaScript code using its name N.

However, WebView does not provide any access control on JavaScript Bridge. Any JavaScript code loaded in WebView can easily access it without any limitations. This has been well studied by existing work [9] [10] [13]. Several defense solutions [9, 10, 14–17] 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.
2.3 Event Handlers

WebView can specify event handlers to handle web events that occur in WebView. The event handler feature makes WebView more powerful. Usually, the function prototypes of event handlers are pre-defined by the Android system in the native (i.e., Java in Android) language. Hence, to implement an event handler, developers need to override the corresponding Java function, and then register the implementation in WebView. When the corresponding event is triggered in the web context, the event handler implemented by developers is called to handle it. For instance, developers can override the event handler `shouldOverrideUrlLoading()` to handle the URL navigation event. If an event handler is not implemented by developers, the default implementation in the Android system will be called.

WebView manages event handlers by either itself or event handler classes. An event handler class is a collection of event handlers. There are mainly two types of event handler classes. One is WebViewClient, which manages the event handlers that are relevant to URL navigation. The other is WebChromeClient, which manages the event handlers that are relevant to UI display, such as handling the alert dialog opened by JavaScript `alert()`.

2.4 Iframes/Popups and Related Protection

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 such as 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:

```html
<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.

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

- **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. It is mitigated by the postMessage mechanism.

- **Browser 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 [3], 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 [4]. 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).

### 2.5 Inconsistencies Between Regular Browsers and WebView

As discussed in Section 1.4, 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 1.1), including:

- **loadUrl(URL/file/JS)**: loading content in the main frame; Through the API `loadUrl()`, WebView renders content in its UI component. The parameter format supported by `loadUrl()` is diverse. It can be a URL, a local HTML file, or JavaScript code. If the parameter is JavaScript code, 1) it must start with the special string “javascript:”, and 2) it is executed in the main web frame. For instance, the following code will popup an alert window to show current cookie in the main frame:

  ```java
  WebView.loadUrl("javascript:alert(document.cookie);").
  ```

- **loadData(HTML, ...)**: loading code with the “null” origin;

- **loadDataWithURL(origin,HTML,...)**: loading HTML code with a specified origin.

The last two APIs are often used to load web content from local storage.

Last, as shown in Figure 1.1, 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 can handle web events that occur inside WebView; and 2) JavaScript bridges, which can allow JavaScript code to directly access native methods.

Furthermore, as shown in Table 2.1, several programming features can impact iframe/popup behaviors. To enable the creation of a popup, the settings `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, load content, the content should be approved by the event handler “`shouldOverrideUrlLoading()`”.

<table>
<thead>
<tr>
<th>Features</th>
<th>Content</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Settings</td>
<td>OpenWindowsAutomatically</td>
<td>Enable “<code>window.open()</code>”</td>
</tr>
<tr>
<td></td>
<td>SupportMultipleWindows</td>
<td>Enable the event handler “<code>onCreateWindow()</code>”</td>
</tr>
<tr>
<td>Event Handlers</td>
<td><code>onCreateWindow()</code></td>
<td>Handle window-creation</td>
</tr>
<tr>
<td></td>
<td><code>onCloseWindow()</code></td>
<td>Handle window-closure</td>
</tr>
<tr>
<td></td>
<td><code>shouldOverrideUrlLoading()</code></td>
<td>Handle URL-loading</td>
</tr>
</tbody>
</table>

### 2.6 postMessage and Hybrid postMessage

```javascript
// Send a message
window.postMessage(m, t)

// Enable the first message handler
function message_handler(e) { ... }
window.addEventListener("message", message_handler, false)

// Enable the second message handler
onmessage = function(e) { ... }
```

Listing 2.1: Usage of `postMessage`

`postMessage`. `postMessage` is frequently used to exchange data between different origins in HTML5-enabled web applications. Listing 2.1 presents the basic usage of `postMessage`. In Line 2, `window.postMessage()` is called to send the message content `m` to the target origin `t`. From Line 4 to Line 9, two message handlers are enabled in two different manners: 1) calling the method `addEventListener()` to register the message handler `message_handler()` (Line 6); 2) or rewriting the global object `onmessage` to enable an anonymous message handler (Line 9). Please note that when a message arrives, both these two message handlers will be called to handle it.

When a message handler is called, the parameter `e` carries all required information, such as the message content `e.data`, the message source origin `e.origin`, and the message sender’s window reference `e.source`. Please note that `e.source` may also be used to identify the message sender. However, in this paper, we mainly focus on `e.origin`.

---

11
The message handler (receiver) is responsible for validating the source origin to ensure the message is from a trusted origin. This requirement is deferred to the message handler implementation and not enforced by the OS or framework. The absence of such validation will cause the client-side validation vulnerability (i.e., CSV), which is well studied by existing work [18–20].

**The Official Hybrid postMessage APIs in WebView.** In Android 6.0, cross-document APIs (such as “WebView.postMessage()”) and channel messaging APIs (such as “WebView.createWebMessageChannel()”) [21] are added. However, both suffer from the Android fragmentation problem [22]. Based on the new Android version distribution data [23] (Nov. 2017), almost 42% of Android devices do not support these official APIs. Furthermore, compared with “postWebMessage()”, “createWebMessageChannel()” can allow bidirectional communication. However, in our empirical study, we found channel messaging was heavy, and rarely implemented and used in hybrid postMessage.
3. STUDYING AND PROTECTING WEB EVENTS*

3.1 Overview

3.1.1 Motivation

There are several potential ways for untrusted code loaded in WebView to leverage the event handler bridge (Figure 3.1). In the first attack scenario, web/mobile bridges may be directly utilized to access the sensitive functionalities behind these bridges by untrusted web code. Prior work (e.g., [24] [2] [25] [26] [27] [28]) has well studied JavaScript Bridge (Figure 1.1), a format of web/mobile bridges that directly allow web code to call mobile code (e.g., Java code for Android). However, we find another type of web/mobile bridges (named “Event Handler”), which allow event handlers pre-defined and registered in the native (or mobile) layer (e.g., Java in Android) to handle web events that occur in WebView, are rarely studied.

3.1.2 Security Issues

We find the utilization of event handler bridges in attack vectors is much more complicated and difficult, compared to JavaScript bridges. There are two main potential attack scenarios based on the event handler features. A possible attack scenario is that an adversary may trigger an event handler with appropriate input to leverage its internal critical functionalities. More details are shown in Figure 3.1. First, the adversary injects malicious HTML/JavaScript code into WebView through web or network attacks (Step 1). Then, the malicious code is executed and triggers a web event (Step 2). After that, the corresponding event handler in the native code is called (Step 3). Finally, the event handler is guided by the injected input to execute its internal critical functionalities (Step 4).

The above possibility is confirmed by our small-scale empirical study of a small set of popular hybrid apps. We found that an event handler in an old but still popular advertisement (ad) library, “millennialmedia” (version 5), contains rich and powerful functionalities, such as reading Android ID, recording audio and opening the camera. However, the access control on that event handler is

* Part of this chapter is reprinted with the permission from “Automated Generation of Event-Oriented Exploits in Android Hybrid Apps”, GuangLiang Yang, Jeff Huang, and Guofei Gu. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018, Copyright © 2018 by Internet Society.
weak. The internal critical functionalities can be utilized by triggering the associated web event and feeding it with appropriate input that follows the format “mmsdk://c1.c2?args=...&callback=k=...”, where c1 and c2 are the native functions to be accessed, args are the function’s parameters and callback is a JavaScript function name to receive the execution result of the native function.

In addition to the above scenario, another potential attack scenario is that a path to a critical functionality inside an event handler may be executed only under a specific program state, but such state may not be simply reached by only feeding that event handler with arbitrary input. Instead, similar to return oriented programming based attacks [29], it is possible for adversaries to play web events as “gadgets” and change an app’s state. Assume the target program state is $S_t$. It may be reached through the transitions $[S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_t]$, which could be achieved by triggering the sequence of web events $[E_1 \rightarrow E_2 \rightarrow ... \rightarrow E_{t-1}]$. Hence, by following the above web event chain, adversaries can still change the program state to $S_t$ and execute the target critical functionality.

We generalize all above attacks as Event-Oriented Exploit (EOE). Due to EOE’s powerful capabilities to access critical functionalities through event handlers, serious consequences may be caused, such as local resource access, users’ private data leakage and web cross-frame DOM manipulation.

Compared with existing attacks on Android (such as Trojan Attack [30]), EOE has multiple
advantages. First, EOE does not require any extra permissions. The malicious web code injected by adversaries fully inherits the target apps’ permissions. Second, EOE does not require malicious payloads. Instead, the functionalities contained in event handlers are utilized.

Furthermore, compared with existing attacks on WebView (such as sidewinder targeted attack [31], fracking attack [9], and code injection attack [32]), EOE is more practical and feasible. Existing attacks usually require JavaScript and JavaScript-bridge to be enabled, but EOE has no such requirements (Section 3.4.1). Even only through HTML code and special HTTP(s) responses, adversaries can still trigger and leverage many event handlers, including the popular event handlers `shouldOverrideUrlLoading()`, which handles the URL navigation event.

The impact of EOE to smartphone security is serious considering the pervasive deployment of hybrid apps today. However, exiting techniques face significant challenges in detecting and verifying apps against EOE. Static analysis suffers from high false positives due to the lack of real data and context. In addition, the limitation of static analysis for handling Java reflection is exacerbated when the reflection operation is combined with `array-indexing type implicit flows`, which occur frequently when parsing the gadgets’ inputs. Dynamic analysis may have low false positives, but is prone to low code coverage. Moreover, generating the required sequence of gadgets to reveal an EOE vulnerability is inherently challenging.

![Figure 3.2: The Design of EOEDroid.](image-url)

### 3.1.3 Vulnerability Detection

The design of EOEDroid is depicted in Figure 3.2. Given a target app, EOEDroid first employs selective symbolic execution to analyze all its event handlers, actively explore all interesting paths.
and identify critical functionalities. The path constraints of each interesting path are collected for further analysis. A significant difference with existing symbolic execution based techniques is that EOEDroid carefully handles all conditional statements, including those whose associated operands are not symbolic (i.e., concrete or constant). This is because those conditional statements can provide hints to generate gadgets’ execution orders.

To mitigate the notorious “path explosion” problem in symbolic execution, we use several heuristics (e.g., scanning “interesting” APIs and instructions to discover interesting paths in Section 3.4.2.1). While these heuristics might cause over-approximation and/or inaccuracy to our analysis, they help us make a good tradeoff between performance and accuracy. In addition, we propose new solutions to address the analysis challenges raised by array-indexing type implicit flows as well as Android features and specifications such as unsupported fork() and inter-component communication (e.g., Android Intent).

Based on the results of selective symbolic execution, EOEDroid then applies static analysis to discover program states that can lead to the execution of a critical functionality, and generates input and execution order of event handlers to reach the program state. The input of an event handler can be generated by solving its path constraints, and the execution order of event handlers can be constructed by solving the event handler dependency problem on those conditional statements whose operands are not symbolic.

Finally, EOEDroid generates exploit code by converting event handlers’ input and execution orders to gadgets’ (i.e., web events). If JavaScript code is required as gadgets’ input, EOEDroid is also aware of its syntax and generates the required code.

Along with this, we conduct a systematic study of events, event handlers, and their triggering code and constraints in WebView. We find that 37 web events are exposed to adversaries, and the constraints on triggering events and event handlers are mainly caused by the status of JavaScript and the level of the web frame the malicious code is injected into. We also find that five event handlers have extra trigger constraints caused by predetermined execution orders of event handlers, and we identify 29 channels that can pass data from web code to native code.
3.1.4 Evaluation

We have implemented EOEDroid based on the Android framework and the Dalvik virtual machine (DVM), and evaluated it with 3,652 most popular apps collected from Google Play. EOEDroid found 97 total vulnerabilities in 58 apps, including 2 cross-frame DOM manipulation, 53 phishing, 30 sensitive information leakage, 1 local resources access, and 11 Intent abuse vulnerabilities. We also found a potential backdoor in a high-profile app that may be used by adversaries to steal users’ sensitive information, such as IMEI. Even though the developers of the app attempted to close the backdoor, EOEDroid found that adversaries were still able to exploit it by triggering two events together and feeding event handlers with appropriate inputs. We show more details in our case study in Section 3.5.3.2 to illustrate this vulnerability.

3.2 Problem Statement

3.2.1 Motivating Example

To illustrate event-oriented exploits, we walk through a real-world vulnerable app with relevant code shown in Figure 3.3. In the activity “WebV iewActivity”, the app initializes a webview component by a class “MyClient”, which implements an event handler “shouldOverrideUrlLoading()”. In the event handler, the input url is firstly parsed by a class “URI”, which is commonly used to analyze URI’s syntax and extract useful information, such as URI’s scheme and host. Then, the url’s content is analyzed, which determines the event handler’s behaviors. If the url’s scheme is “market”, “tel”, or “sms”, the corresponding external apps (such as Google Play, default phone call app, or default text message app) will be opened to handle the input (Path1). If the url’s host is “developer.com” (which means WebView connects to a remote server), the event handler may approve the connection (Path2).

Meanwhile, the event handler implements supports for the customized scheme “sdk”. If the url’s host h is “init”, WebView executes JavaScript code to perform initialization (Path3). If h’s format is “c0,c1,c2”, the app calls the Java method whose class name is determined by c0, method name c1, and execution result is transferred to the JavaScript method c3 (Path4). Note that resolving the Java method relies on the content of the variable “hashmap”, which converts c0 (i.e., commands[0]) to the real class name (i.e., className). Such an operation introduces an implicit flow from c0 to className. “Class2” is one of classes whose methods can be invoked by the event handler. In its
method `getId()`, the device ID is transferred to the web space. In its method `login()`, the activity “LoginActivity” is started through an Intent message to ask users to login. In the Intent message, part of the `url`’s content (i.e., `c2`) is contained and passed to the message receiver.

Note that in the example app there is a critical function: `getDeviceId()` in the method `getId()` of `Class2`. Adversaries cannot directly utilize this functionality, because the operand `tmpbool` is false (i.e., the conditional statement `C6`). However, by manipulating gadgets, adversaries may change the program state (such as `tmpbool`’s value) and drive the app to call `getId()`. In `getId()` a JavaScript function is also required as part of the event handler’s input to receive the device ID.

Although it appears simple to manually analyze this example code, real-world apps are much more complex. Our goal is to develop a technique that can automatically detect such vulnerabilities and construct exploit code.

![Figure 3.3: Vulnerable Code from a Real-World App](image_url)
3.2.2 Threat Model

We assume that WebView is enabled in apps, but JavaScript is not required to be enabled, since HTML code can also trigger event handlers. We assume that adversaries can inject malicious HTML/JavaScript code into WebView. As Figure 3.1 shows, we consider the following two different attack scenarios:

- **Web Attack**: In this scenario, we assume adversaries control several malicious domains and servers, but they are not able to control or monitor the network traffic between apps and other domains.

  The web content loaded from first-parties is trustable. However, the content may further contain subframes (e.g., iframe) to load extra web content from third-parties, which may be malicious.

  Generally, all web frames loaded in WebView are well isolated and protected by same origin policy (SOP) [6].

- **Network Attack**: Adversaries can hijack unsafe network traffic (such as HTTP) through man-in-the-middle attacks. Compared with desktop programs, mobile apps are more likely to suffer from this type of attacks, considering that many unsafe WI-FI hotspots are used [34].

  Note that we do not assume any other abilities of the adversaries. They may not access the users’ device, install any certificate or malware, or change apps’ internal data. The target app itself as well as all the apps pre-installed on the users’ devices may be benign.

3.2.3 Security Issues

Similar to other attacks on WebView [9], the security issues caused by event handlers are rooted in the inconsistency of security models between web and native context. In hybrid apps, the SOP security model for the web context is circumscribed to prevent event handlers from being triggered by malicious web code, because the handlers do not have any way of identifying the origin of an event (so they have no way to distinguish between trusted and untrusted origins). SOP is also ineffective to protect the local resources (such as camera), which are located in the native context. The permission based sandbox model for the native context can protect local resources. However,
it is ineffective to prevent the access to critical functionalities from web code, since the origin information of the access is lost.

3.2.4 Problem Definition

We state that an exploit is successful if it successfully triggers a critical functionality through event handlers defined in the app. A successful exploit must satisfy the constraints in triggering target events and event handlers: it must guide the target app to reach the target state by manipulating the input and execution orders of gadgets, and it must bypass all security checks which are usually located before the critical functionality.

The event-oriented exploit generation problem can be formally defined as follows. Given an app, discover a program state $s$ that leads the app to execute a critical functionality. Such a state should be reached through a sequence of executions of gadgets $((W_0, E_0, I_0, J_0), (W_1, E_1, I_1, J_1), \ldots, (W_n, E_n, I_n, J_0))$, where $W_i$ is the HTML/JavaScript code that triggers the event $E_i$ and passes the input $I_i$ to $E_i$. $I_i$ may also include pre-defined JavaScript code $J_i$.

3.2.5 Critical Functionalities

We define critical functionalities as sensitive APIs in the Android framework. In this dissertation, we mainly consider the following four types of APIs. Nevertheless, EOEDroid is extensible and user customized APIs can be added easily.

3.2.5.1 URL Loading API

If malicious HTML/JavaScript code in subframes leverages the URL loading API (e.g., WebView.loadUrl($p$)) through EOE, the content of the main frame or the whole WebView may be changed. Depending on the value of the API parameter $p$, the following two consequences may be caused:

- **Cross-Frame DOM Manipulation**: If the web code in subframes influences $p$’s value and makes $p$ be starting with “javascript:”, the JavaScript code contained in $p$ may be executed in the main frame. Hence, through EOE, the web code in subframes obtains the capability to bypass SOP and inject malicious code to the main frame.

- **Phishing**: If the web code in subframes can determines $p$’s value through EOE, it may change $p$’s value to the url of a fake web page. Then, WebView is redirected to show the fake web
page. Considering that WebView usually does not have an address bar to indicate the url it is loading, such attacks on WebView are much more stealthy than on regular web browsers.

Compared with other attack channels (such as MITM attacks) which may also be utilized to perform above attacks, EOE over loadUrl() is more powerful. Considering the situation that WebView loads a webpage from developers’ web site using HTTPS, and one of its nested subframes uses HTTP. Due to boundaries between frames, existing attacks may only be able to control the content of the subframe, but not the main frame. However, EOE does not have this limitation. By means of loadUrl(), adversaries can directly change the content of the main frame.

3.2.5.2 Source and Sink APIs

This type of API invocations may result in users’ privacy leakage. We mainly consider two scenarios: (1) there are paths from source to sink in event handlers. (2) source is passed to the web space, and then sent out through HTML/JavaScript code.

We consider the Android ID, device ID, phone number, and serial number, and GPS location information as source, and connecting network and sending text message as sink.

3.2.5.3 APIs Accessing Local Resources

This type of APIs may be leveraged by adversaries to access local resources, such as local files, and hardware resource (e.g., camera). Serious consequences may be caused when these APIs are combined with other sensitive APIs. For instance, adversaries may remotely take a picture and also save it to the local storage using camera APIs. Then, adversaries may obtain the picture in the web context through file reading API and further send the picture out through native sink APIs or HTML/JavaScript code.

3.2.5.4 APIs Sending Intent messages

As demonstrated by Wang et al. [14], the Intent messages that are sent out through WebView may have serious consequences. We consider the following type of APIs as sensitive: the API parameter is totally controlled by adversaries, which means the destination of the Intent message to be sent is totally determined by adversaries. For other Intent-sending APIs, we treat them as regular inter-component communications.
3.3 System Overview

In this section, we provide an overview of EOEDroid and illustrate it with the motivating example described in the previous section. The technical details of EOEDroid are presented in Section 3.4.

We use the following basic concepts and notations:

- **A Symbolic Conditional Statement**: a conditional statement whose operands are symbolic.

- **Path Constraints**: all constraints that must be satisfied when guiding an app to execute a path. Different from prior work, EOEDroid involves both symbolic and non-symbolic conditional statements in path constraints.

- **Input Constraints**: A subset of path constraints but are only related to event handlers’ input.

We assume that $s$ is the target program state that leads to the execution of a critical functionality; $f$ is the target critical functionality; $p_0$ is the path containing $f$; $eh_0$ is the event handler containing $p_0$.

3.3.1 Overview

EOEDroid consists of three modules: event handler analysis, program state analysis, and exploit code generation, as shown in Figure 3.2. In the first module, selective symbolic execution is used to explore paths in the event handlers and collect path constraints. To apply the technique for Android hybrid apps, technical challenges (Section 3.4.2) are addressed by four sub-modules: analysis sandbox, heuristic generation, Intent handler, and array-indexing type implicit flow handler. More specifically, given an app, “selective symbolic execution” is called to repeatedly test each event handler until all the inside interesting paths are traversed. The interesting paths are discovered by the sub-module “heuristic-generation”. Note that when a branch is flagged as interesting, no matter whether the conditional statement is symbolic or not, EOEDroid forcibly traverses this path. Meanwhile, the corresponding path constraint is constructed and saved.

For each round of test, the sub-module “analysis sandbox” is applied to guard the analysis environment from pollution and keep each round of test independent.

In the second phase, the module “program state analysis” runs to discover state $s$ and learn how to reach $s$ by manipulating event handlers’ input and execution order, which are handled by
the sub-modules “event handler input generation” and “event handler execution order generation” respectively. For event handlers’ input, it is generated by applying an SMT solver in the associated input constraints collected in the first phase. For event handlers’ execution order, it is generated by solving the event handler dependency problem.

For each path $p$ that contains critical functionalities, EOEDroid repeatedly resolves all event handler dependencies for $p$ with four steps: (1) it analyzes $p$’s path constraints to identify all non-symbolic conditional statements; (2) it confirms the expected value $v$ for each conditional statement; (3) starting from each conditional statement $c$, it performs backward program analysis to determine the variables $O$ that can influence $c$’s operands, and further computes the required value for each variable in $O$; and (4) it analyzes all paths in all event handlers that contain the instructions changing the variables in $O$ to their corresponding expected values.

In the third phase, the module exploit code generation generates exploit code for each exploitable critical functionality. First, the event handlers’ execution order generated in the second phase is converted to the web event order, and the event handlers’ input is converted to the corresponding web events’. Second, if JavaScript code is required as the event handler’s input (such as the callback function in our motivating example), the syntax of the associated JavaScript code is parsed and analyzed to generated required JavaScript code.

3.3.2 Analyzing the Example

Now we illustrate how EOEDroid works for our motivating example. When the event handler `shouldOverrideUrlLoading()` is triggered, EOEDroid is started. First of all, EOEDroid symbolizes the event handler’s second parameter as ‘InputUrl’, since its value can be controlled by adversaries. Then, EOEDroid analyzes each instruction. As the class `Uri` is frequently used, we model it by symbolizing its instance $u$ as ‘`Uri.<init>(InputUrl)`’. The input’s scheme and host are also symbolized, whose symbolic expressions are ‘`Uri.<init>(InputUrl).getScheme()`’ and ‘`Uri.<init>(InputUrl).getHost()`’, respectively.

When the conditional statement $C1$ is analyzed, “heuristic generation” is started to discover which branches are interesting. In this case, both branches have interesting instructions. So both of them are sequentially traversed. In the true branch, when an Intent message is sent to another app or component, the module “Intent handler” (Section 3.4.2.3) is set up to fill the symbolic information.
gap between the sender and receiver.

Similarly, the conditional statements $C^2$, $C^3$ and $C^4$ are processed. In $C^4$’s true branch, EOEDroid encounters a special conditional statement that is non-symbolic (i.e., $C^5$). As its true branch is interesting, EOEDroid forcibly executes it and also collects necessary information, such as the executed path information, the instruction’s position (such as $<\text{MyClient.java}, C^5>$), the condition expression (i.e., $\text{tmpbool} == 0$), the operand variable (i.e., $\text{tmpbool}$), current value of the variable (i.e., 0) and its selected branch (i.e., 1). Note that in this path the external field variable $\text{Initialized}$ is written. To ensure each round of test is independent, such interaction between the event handler and the external variable is handled by the sub-module “analysis sandbox”.

The conditional statement $C^6$ is then reached. In the true branch, the host name is split to an array, whose symbolic expression is $\text{Uri.<init>(InputUrl).getHost().split(".").}$ Then, an implicit flow is faced, which is caused by the Hashmap accessing operation. To handle it, the sub-module “implicit flow handler” is started to try all possibilities in the Hashmap instance. Therefore, a critical functionality is found in $\text{getId()}$ in $\text{Class2}$, which can be leveraged by adversaries to perform cross-frame DOM manipulation and steal the device ID information. The main associated path constraints are shown in Listing 3.1.

(1) $\text{Uri.<init>(InputUrl).getScheme().equals("market") == 0}$
(2) $\text{Uri.<init>(InputUrl).getScheme().equals("tel") == 0}$
(3) $\text{Uri.<init>(InputUrl).getScheme().equals("sms") == 0}$
(4) $\text{Uri.<init>(InputUrl).getHost().equals("developer.com") == 0}$
(5) $\text{Uri.<init>(InputUrl).getScheme().equals("sdk") \neq 0}$
(6) $\text{tmpbool \neq 0}$
(7) $\text{Uri.<init>(InputUrl).getHost().split(".").length == 3}$
(8) $\text{Uri.<init>(InputUrl).getHost().split(".")[0].equals("c2") \neq 0}$ // generated by implicit flow handler
(9) $\text{Uri.<init>(InputUrl).getHost().split(".")[1].equals("getId") \neq 0}$

Listing 3.1: Path Constraints in Executing $\text{getId()}$

In the second phase, the module “program state analysis” analyzes the path constraints (Listing 3.1) to change the program state. First, the sub-module “event handler input generation” checks if the constraints can be satisfied by feeding the event handler with appropriate input. In this case, all constraints except (6) can be satisfied. Second, the sub-module “event handler execution order generation” runs to check how to influence the program state to satisfy the constraint (6). Starting from the conditional statement $C^6$, EOEDroid backward tracks the operand $\text{tmpbool}$ along the executed path, and confirms the variable (i.e., $\text{Initialized}$) can influence its value. Next, EOEDroid goes through all paths identified in the first phase to check whether there is a path that contains
an instruction changing *Initialized’s* value. It finds that Path3 contains the expected operation. Hence, there is an event handler dependency on C6: $\text{Path3} \xrightarrow{\text{C6}} \text{<shouldOverrideUrlLoading(), Path4>}$.

In the third phase, the module “exploit code generation” generates the exploit code for the critical functionality in `getId()`. To drive the app to execute the critical functionality, event handlers should be executed as follows:

1. `shouldOverrideUrlLoading(webview, "sdk://init")`
2. `shouldOverrideUrlLoading(webview, "sdk://c2.getId.?")`

Then, the above event handler execution order is converted to the web event order, and further transformed to the following HTML/JavaScript code (based on our event handler study presented in Section 3.4.1):

```
<iframe src="sdk://init"/>
<iframe src="sdk://c2.getId.?"/>
```

The above code can change the program state and reach the sensitive API `loadUrl()`. However, part of the event handler’s input is still missing, which is a JavaScript callback function used to receive the sensitive information (i.e., device ID). To address this problem, the sub-module “JavaScript code syntax analysis” runs to analyze the syntax of the parameter of `loadUrl()`, and generate required JavaScript code. Finally, the following exploit code is generated, which can help developers test and verify the EOE problem.

```
1 <script>
2 function steal_device_id(id) {
3     document.write("<" + "img src='" + "http://attacker.com/" + id + "' />")
4 }</script>
5 <iframe src="sdk://init"/>
6 <iframe src="sdk://c2.getId.steal_device_id"/>
Listing 3.2: Exploit Code
```

### 3.4 Technical Approaches

In this section, we first present our study of events and event handlers in WebView to understand their constraints for triggering event-oriented exploits. We then present technical details about the design and implementation of selective symbolic execution, program state analysis, and exploit code generation.
Table 3.1: The Systematic Study Result. The third column ‘JS?’ means: ‘Is JavaScript required to trigger the event?’, and the forth column ‘E₀?’ means: ‘Does the event handler only deal with events from E₀?’. In answers, we use ✓ and blank to indicate ‘Yes’ and ‘No’, respectively.

<table>
<thead>
<tr>
<th>Event Handlers and Main Parameters</th>
<th>Handled Events</th>
<th>JS?</th>
<th>E₀?</th>
<th>Example Trigger Code (HTML/JavaScript/HTTP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>onsubmit</td>
<td>Resubmitting a form</td>
<td>✓</td>
<td></td>
<td>[HTML] &lt;form ...&gt; [JS] form.resubmit()</td>
</tr>
<tr>
<td>onPageFinished</td>
<td>Finishing loading a page</td>
<td>✓</td>
<td>✓</td>
<td>Trigger Constraint</td>
</tr>
<tr>
<td>onReceivedErrorsNav emulation, description, url</td>
<td>Failing to load a page</td>
<td>✓</td>
<td></td>
<td>[HTML] &lt;iframe src=&quot;<a href="http://invalid.url">http://invalid.url</a>&quot; ...&gt;</td>
</tr>
<tr>
<td>onReceivedClientCertRequest [request]</td>
<td>Client cert request</td>
<td>[HTTP]</td>
<td>[JS] navigate to &quot;<a href="http://secure.example.com">http://secure.example.com</a>&quot;</td>
<td></td>
</tr>
<tr>
<td>onReceivedHttpAuthRequest [host, realm]</td>
<td>Authentication request</td>
<td>[HTTP]</td>
<td>[JS] navigate to &quot;<a href="http://secure.example.com">http://secure.example.com</a>&quot;</td>
<td></td>
</tr>
<tr>
<td>onReceivedHttpError [request, response]</td>
<td>HTTP error</td>
<td>[HTTP]</td>
<td>[JS] navigate to &quot;<a href="http://secure.example.com">http://secure.example.com</a>&quot;</td>
<td></td>
</tr>
<tr>
<td>onReceivedLoginRequest [realm, account, arg]</td>
<td>Login request</td>
<td>[HTTP]</td>
<td>[JS] navigate to &quot;<a href="http://secure.example.com">http://secure.example.com</a>&quot;</td>
<td></td>
</tr>
<tr>
<td>onScaleChange [newScale: number]</td>
<td>Updating scale</td>
<td>✓</td>
<td></td>
<td>[JS] document.body.style.zoom = newScale</td>
</tr>
<tr>
<td>onUndeledgeKey [event, keyevent]</td>
<td>Pressing key</td>
<td>✓</td>
<td></td>
<td>[JS] dispatch key-press event</td>
</tr>
<tr>
<td>onUnhandledKeyEvent [keyevent]</td>
<td>Facing an unhandled key</td>
<td>✓</td>
<td></td>
<td>Trigger Constraint</td>
</tr>
<tr>
<td>shouldInterruptRequest [request]</td>
<td>Resources loading</td>
<td>✓</td>
<td></td>
<td>[HTML] &lt;iframe src=&quot;<a href="http://invalid.url">http://invalid.url</a>&quot; ...&gt;</td>
</tr>
<tr>
<td>shouldOverrideUrlLoading [url [or request]]</td>
<td>URL navigation</td>
<td>✓</td>
<td></td>
<td>[HTML] &lt;iframe src=&quot;<a href="http://invalid.url">http://invalid.url</a>&quot; ...&gt;</td>
</tr>
<tr>
<td>onBeforeWindow</td>
<td>Creating a window</td>
<td>✓</td>
<td></td>
<td>[JS] window.open()</td>
</tr>
<tr>
<td>onConsoleMessage [message]</td>
<td>Printing messages</td>
<td>✓</td>
<td></td>
<td>[JS] console.log()</td>
</tr>
<tr>
<td>onGeolocationPermissionsShowPrompt [origin]</td>
<td>GPS request</td>
<td>✓</td>
<td></td>
<td>[JS] navigator.geolocation.getCurrentPosition()</td>
</tr>
<tr>
<td>onGeolocationPermissionsHidePrompt</td>
<td></td>
<td>✓</td>
<td></td>
<td>Trigger Constraint</td>
</tr>
<tr>
<td>onShowCustomView</td>
<td>Entering full screen</td>
<td>✓</td>
<td></td>
<td>[JS] webkitRequestFullScreen()</td>
</tr>
<tr>
<td>onHideCustomView</td>
<td>Quitting full screen</td>
<td>✓</td>
<td></td>
<td>Trigger Constraint</td>
</tr>
<tr>
<td>onBeforeunload [message, result]</td>
<td>Leaving a webpage</td>
<td>✓</td>
<td></td>
<td>[JS] dispatch beforeunload event</td>
</tr>
<tr>
<td>onBeforeQualifiedPrompt [message, result]</td>
<td>Popping an alert box</td>
<td>✓</td>
<td></td>
<td>[JS] alert()</td>
</tr>
<tr>
<td>onBeforePrompt [message, defaultValue, result]</td>
<td>Popping a prompt box</td>
<td>✓</td>
<td></td>
<td>[JS] prompt()</td>
</tr>
<tr>
<td>onPermissionRequest [request]</td>
<td>Permission request</td>
<td>✓</td>
<td></td>
<td>[JS] navigate to &quot;<a href="http://example.com">http://example.com</a>&quot;</td>
</tr>
<tr>
<td>onPermissionRequestCanceled [request]</td>
<td>Request is cancelled</td>
<td>✓</td>
<td></td>
<td>Trigger Constraint</td>
</tr>
<tr>
<td>onProgress [loadProgress]</td>
<td>Page loading status</td>
<td>✓</td>
<td></td>
<td>[JS] dispatch a progress event</td>
</tr>
<tr>
<td>onReceivedResource</td>
<td>Requesting focus</td>
<td>✓</td>
<td></td>
<td>[HTML] &lt;input type=&quot;text&quot; id=&quot;name&quot; ...&gt;</td>
</tr>
<tr>
<td>onShowFileChooser</td>
<td>Browsing file system</td>
<td>✓</td>
<td></td>
<td>[JS] dispatch a click event</td>
</tr>
<tr>
<td>onProgress [progress]</td>
<td>Page loading status</td>
<td>✓</td>
<td></td>
<td>[JS] dispatch a progress event</td>
</tr>
<tr>
<td>onReceivedTouchStart [event, precomposed]</td>
<td>Receiving an apple touch icon</td>
<td>✓</td>
<td></td>
<td>[JS] dispatch a click event</td>
</tr>
<tr>
<td>onReceivedResource [url, userAgent, contentDisposition, mimetype, contentLength]</td>
<td>Downloading a file</td>
<td>✓</td>
<td></td>
<td>[HTML] &lt;iframe src=&quot;<a href="http://example.com">http://example.com</a>&quot; ...&gt;</td>
</tr>
</tbody>
</table>

26
3.4.1 Understanding Event Handler Triggering Constraints

The official Android documentation of events and event handlers is obscure and incomplete. We hence conduct a systematic study based on both reading documents about WebView on the web and analyzing real-world hybrid apps. The main study result is shown in Table 3.1. We find that 37 events are available for adversaries in WebView. The triggering code for each event is shown in the fifth column. Note that the DOM element `<iframe src=...>` can directly trigger two events, whose corresponding event handlers are `shouldOverrideUrlLoading()` and `shouldInterceptRequest()`, respectively. It depends on the attribute `src`'s content `s`. If `s`'s scheme is not `HTTP` and `HTTPS`, but customized, the former one is triggered. Otherwise, the latter one is triggered.

As reported in the third column, using HTML code, adversaries can trigger 15 event handlers, including popular event handlers `shouldOverrideUrlLoading()` and `shouldInterceptRequest()`. Note that, four of them require supports from the web server side to get appropriate HTTP response code. For instance, `onReceivedLoginRequest(webview, realm, account, args)` can be triggered by the combination of the HTML code `<iframe src="http://attacker.com/login">" and the HTTP response header "x-auto-login:realm=x&account=y&args=z", which is from the malicious server "attacker.com". `x`, `y`, and `z` are passed to `onReceivedLoginRequest()` as function parameters. As the above example shows, adversaries can pass data from the web context to the native context. In our study, we find that the parameters of 29 event handlers can be influenced by adversaries. More details are shown in the first column (i.e., the parameters between parentheses).

As reported in the fourth column, triggering event handlers are influenced by the level of web frames where the events occur. We find that events which occur in the main frame could trigger all event handlers, whereas the capability of events in subframes is limited. More specifically, three event handlers cannot be triggered by events that occur in the main frame. Let $E_i (i \geq 0)$ be the events that occur in the $ith$ level web frame and can be handled by event handlers, $E$ denotes all events available in the whole WebView space and $E_0$ denotes the events available in the main frame, they have the following relationship: $E_0 = E$ while $E_i \subseteq E (i > 0)$.

3.4.1.1 Event Triggering Constraints

The constraints for triggering events are mainly caused by the status of JavaScript. As reported in Table 3.1, almost 60% of the event handlers require JavaScript enabled to trigger.
3.4.1.2 Event Handler Triggering Constraints

The constraints for triggering event handlers are mainly from two aspects:

I: *The frame level*. Triggering three of the event handlers require their corresponding events to occur in the main frame. Adversaries must inject malicious code into the main frame, which is usually well protected. Also, it is easy for users to realize the injected web code, because it may reload web pages.

II: *Predetermined execution orders*. Several event handlers’ execution orders are predetermined in WebView, which also imposes constraints on triggering the event handlers. To understand these predetermined execution orders, we create an experimental app which registers all event handlers, and profile them when they are invoked. Then, the app loads fuzzing HTML/JavaScript code. We also apply static analysis to track the return values of all event handlers. If an event handler’s return value appears in a conditional statement, and later another event handler is called, a predetermined event order may exist. Finally, we confirm five predetermined execution orders:

1. `shouldInterceptRequest() → onLoadResource()`: The latter event handler is called only when the former event handler returns null.

2. `shouldOverrideKeyEvent() → onUnhandledKeyEvent()`: The latter event handler is only called when the former event handler returns false.

3. `onPageStarted() → ... → onPageFinished()`: When WebView starts loading a web page, `onPageStarted()` is called. When WebView finishes loading the page, `onPageFinished()` is called. During the process, other event handlers may be called as well, such as `onReceivedError()` and `shouldInterceptRequest()`.

4. `onPageStarted()` can be called multiple times before `onPageFinished()` is called. This happens when there are URL redirections in the web server side (i.e., 3xx HTTP response code). The number of times that `onPageStarted()` is called depends on the URL redirection number. Moreover, generally, `onPageFinished()` is only called once, no matter how many URL redirections there are. But if the last HTTP response code is 4xx, WebView may be redirected to show a page-not-found HTML, and then, `onPageFinished()` is called again.

5. `onGeolocationPermissionsShowPrompt() → onGeolocationPermissionsHidePrompt()` and `onShowCustomView() → onHideCustomView()`: When location permission is requested, or Full
Screen is entered, these events are called sequentially.

3.4.1.3 Adversaries’ Capability: Playing Gadgets

Adversaries can change program states by manipulating gadgets’ input and execution orders. More specifically, adversaries can pass data to web events, and then the data are passed to the corresponding event handlers as their function parameters. Adversaries can also trigger events and event handlers in arbitrary orders, even though there are constraints on triggering events and event handlers.

Gadgets’ Input. Adversaries may be able to control event handlers’ parameters. For example, shouldInterceptRequest()’s parameter (i.e., request) can be set as “https://attacker.com/img”, if adversaries use the HTML code “<iframe src="https://attacker.com/img"></iframe>” to trigger the event handler.

Gadgets’ Execution Orders. Consider two event handlers $eh_1$ and $eh_2$, there are two cases to analyze: (1) If $eh_1$ and $eh_2$ do not have any relationship, adversaries can call them in any order (i.e., $eh_1\rightarrow e_2$ and $eh_2\rightarrow eh_1$); (2) If $eh_1$ must be executed before $eh_2$, their relationship should be $t\rightarrow eh_1 \leftarrow c \rightarrow eh_2$, where $t$ is the trigger code to call $eh_1$ and $c$ is the pre-condition that must be satisfied to trigger $eh_2$. By repeating $t$ and make $c$ be satisfied, we may get the event handler sequence $(eh_1eh_2eh_1eh_2)$, which includes expected sequences (both $eh_1\rightarrow eh_2$ and $eh_2\rightarrow eh_1$).

3.4.2 Selective Symbolic Execution

To apply symbolic execution in event handlers, we address four challenges (with details in following subsections):

- **Path explosion**: To address this notorious problem, EOEDroid uses static analysis to provide heuristic information for path selection. However, static analysis may introduce false negatives to the heuristic information. To avoid it, we conservatively and safely apply static analysis on only a certain number of instructions that do not cause false negatives (Section 3.4.2.1).

- **Unsupported fork()**: In existing dynamic symbolic execution based approaches, fork() is frequently used to help systems traverse branches and keep the analysis environment clean. However, in Android, fork() is not supported. Instead, EOEDroid needs to sequentially traverse branches. However, different with desktop software, it is expensive to save and
restore states of Android apps. To fix the problem, we propose an analysis sandbox to handle the interaction with the external environment (Section 3.4.2.2).

- **Android Intent**: Intent is frequently used in event handlers, such as triggering a GUI event to open a GUI activity. However, it introduces semantic gap between Intent senders and receivers. Figure 3.4 shows an example that an intent message is delivered between two apps. The Intent message escapes from the Java context (i.e., DVM), enters the C/C++ context (i.e., Linux kernel), and finally returns to the Java context. This way raises challenges to track the Intent message in the Java context. When the receiver obtains the message, the associated symbolic information may be lost. To address the problem, we fill the gap between senders and receivers by synchronizing the symbol information in both sides (Section 3.4.2.3).

![Figure 3.4: Intent in Inter-Apps Communications](image)

- **Array-indexing class implicit flows**: In array-indexing type operations, if the index is symbolic, it is challenging to determine which element should be returned. The problem is known as "implicit flow". Similar problems also exist in other data structures such as Hashmap, Android Bundle, and Android share preference. In real world, this type of operations and data structures was frequently used in popular apps and ad libs, such as Google Ads.

To further demonstrate the problem, we use Hashmap as the example. As Figure 3.5 shows, in Java, Hashmap is implemented based on a bucket array with linked lists that are used to handle hashing collisions. Assume that the instruction \( v = M.get(k) \) is being executed, where \( M \) is the Hashmap object, \( k \) is the key and it is symbolized as ‘key’. In the function Hashmap.get(), the bucket index is firstly determined, which is \( k \)’s hashcode. Hence, the index is a symbolic expression built on \( key \). Then, an array-indexing operation is done to
obtain the associated linked-list. Since the index is symbolic, the operation introduces an implicit flow.

To mitigate the problem, we instrument $k$ to brute-forcely try all possibilities of keys (Section 3.4.2.4).

![Diagram](image.png)

Figure 3.5: The Internal Structure of HashMap

We implement selective symbolic execution by instrumenting the Android framework and Dalvik virtual machine (DVM). In Android frameworks, event handler functions and sensitive APIs (Section 3.2.5) are handled. In DVM, the mapping between variables and their corresponding symbolic expressions are managed through a global symbolic table. To support string operations, which are frequently faced in event handlers, the associated string APIs are modeled, including compare, append, replace, search, substring and split, and we use Z3-Str [35] to resolve string based path constraints.

3.4.2.1 Heuristic Generation

The heuristic information used in path selection includes the indication of whether a branch is interesting. To determine it, EOEDroid uses static analysis to scan a certain number (such as 100) of instructions in advance to check if a critical functionality is contained.

Due to the imprecision of static analysis, false negatives may be introduced. The determination result may be further influenced. To eliminate the concern, we also flag the following types of operations as *interesting*:

- **Field variables reading and writing**: This affects points-to and alias relationship.

- **Virtual function invocation**: Resolving this kind of invocations requires points-to information.
• **Java Reflection**: Due to the lack of real data, it is challenging for static analysis to solve this kind of problems.

• **Return Instruction**: In event handlers, the returned values of some event handlers (Section 3.4.1) are meaningful, such as `shouldOverrideUrlLoading()` and `shouldInterceptRequest()`. Take the former event handler as the example: If the event handler returns true, it means the app being analyzed handles the input. Otherwise, the Android system processes the input.

### 3.4.2.2 Analysis Sandbox

To keep the analysis environment clean, EOEDroid creates a sandbox environment to replace the real environment. All interactions with the external real environment is redirected to the sandbox environment. Based on the access direction, the interactions can be divided into two categories: writing and reading. For the writing operation, EOEDroid updates variables’ values in the sandbox instead of the real environment. For the reading operation, if the destination variable is written earlier, the corresponding value in the sandbox is retrieved and returned; otherwise, the value in the real environment is returned.

In this dissertation, we consider the interactions include accessing file system, global variables, and field variables whose scopes are bigger than the event handler function being analyzed. To implement them, necessary APIs and instructions are hooked and handled. For reading and writing files, the corresponding POSIX APIs (in libcore\io\Posix.java) are handled. However, it is challenging to maintain a file’s status, especially when the file is partially modified. To mitigate the problem, a backup file is created, and then all reading and writing operations are redirected to the backup file. For reading and writing global and field variables, the associated instructions (i.e., `iget/iput`, `aget/aput`, and `sget/sput`) [11] are handled. In practice, it is challenging to determine the scope of a field variable. To simplify the problem, all changes on the field variable are recorded. Please note that in the beginning of each round of test, all data and files saved in the sandbox are cleaned.

### 3.4.2.3 Intent Handler

To fill the symbolic information gap between Intent message senders and receivers, it is critical to restore symbolic information of the message in the receiver side. For this purpose, when the
Intent message is sent, EOEDroid temporally pauses the program by hooking the associated APIs (such as `startActivity(Intent)`), makes snapshot on the Intent object and its corresponding symbolic data, and also saves it. Then, when the receiver accepts and reads the message using associated APIs (such as `getIntent()`), the snapshot is read, and then the symbolic information is linked with the Intent object. Considering the sender and receiver may be not in the same app, such a snapshot is dumped to a public folder, which is allowed to be accessed by any app.

As variables’ absolute memory addresses are used to save their symbolic information in the snapshot, in the receiver side the restored symbolic information cannot be directly applied in the received Intent message, whose memory addresses are totally different from the sent message. To correct the differences, when the snapshot is made in the sender side, memory addresses are changed to relative addresses, based on the starting address of the sent message. Then, when the snapshot is read in the receiver side, memory addresses are changed back to the absolute addresses, based on the starting address of the received message.

Furthermore, to distinguish different Intent messages, each message is assigned a unique ID, which is also used as the corresponding snapshot’s name. To support it, a new integer field “IntentId” is added into the Intent Java class. Each time an Intent message is created, the field is automatically added by one.

### 3.4.2.4 Array-Indexing Type Implicit Flow

To mitigate the problem caused by this type of implicit flows, we brute-forcely convert the associated operation into multiple conditional statements. Array and other data structures are handled respectively as follows:

- **Array**: Assume the content of an array $A$ is $[e_0, e_1, e_2, ..., e_n]$, and in the operation $r = A[i]$, $i$ is symbolic. The operation can be converted to the following structure:

  ```java
  if (0 == i) r = e_0;
  else if (1 == i) r = e_1;
  ...;
  else if (n == i) r = e_n;
  ```

  Next, EOEDroid can handle the operation as regular conditional statements.

- **Hashmap, Android Bundle, and Android Share Preference**: Similar to array-indexing operations, hashmap type accessing can also be transformed to conditional statements. As-
sume that the following instruction is faced: \( r = \text{hasmap.get}(k) \). The keys of hashmap is \([k_0, k_1, k_2, \ldots, k_n]\). Hence, by instrumenting \( k \)'s real value in memory, the operation can also be converted to regular conditional statements.

\[
\begin{align*}
\text{if} & \ (k.\text{equals}(k_0)) \ k = k_0; \\
\text{else if} & \ (k.\text{equals}(k_1)) \ k = k_1; \\
\text{else if} & \ (k.\text{equals}(k_n)) \ k = k_n; \\
r & = \text{hasmap.get}(k);
\end{align*}
\]

To support the above operations, all keys in the hashmap object must be retrieved. However, it is challenging to do that in the low level layer (e.g., DVM). To fix the problem, the HashMap class is instrumented by adding a string array to record all keys. Thus, in the DVM, all keys can be retrieved by restoring the values of the added string array.

### 3.4.3 Program State Analysis

To discover how to reach the program state that leads to the execution of a critical functionality, we deal with the input and execution order of event handlers respectively.

#### 3.4.3.1 Event Handler Input Generation

Given an arbitrary interesting path, its input can be generated by handling its associated path constraints that are collected in the first phase. First, input constraints are extracted from the whole path constraints by filtering out the constraints of non-symbolic conditional statements. Second, the input can be generated by resolving the input constraints using an SMT solver (e.g., Z3-Str).

#### 3.4.3.2 Event Handler Execution Order Generation

Given a path that contains a critical functionality, the execution order of event handlers can be obtained by addressing the event handler dependency problem. The algorithm is shown in Algorithm 1. In the algorithm, three critical functions are required as input:

- **NS(eh, p, insn)**: Non-symbolic conditional statements can be extracted by going backward through \( p \) starting from \( insn \) and checking the operands of all faced conditional statements.

- **get_origin_variables(eh, p, insn, v)**: We define the origin variables as following. If in \( p \), \( v' \) can influence \( v \)'s value, \( v' \) is an origin variable of \( v \). Hence, to locate all \( v' \), we go backward through \( p \) starting from \( insn \), and apply backward data flow tracking on \( v \). If a variable is
found in the backward data flow and located in the external environment, the variable may be one of \( v \)’s origin variables.

- \textit{get\_origin\_values}(\textit{eh}, \textit{p}, \textit{O}, \textit{insn}, \textit{value}): To compute the expected values of origin variables, we re-run symbolic execution on \( p \) to construct \( v \)’s symbolic expression relying on origin variables. To this end, all origin variables in the set \( O \) are symbolized. Then, \( p \) is executed and analyzed by feeding \( eh \) with appropriate input. Next, when conditional statements are faced, the path constraint is constructed and saved. After that, when the instruction \( insn \) is faced, the analysis is finished. Finally, the values of origin variables can be generated by resolving the collected path constraints.

### 3.4.4 Exploit Code Generation

Algorithm 2 shows our algorithm to generate the exploit code. Two main functions (\textit{get\_web\_trigger\_code()} and \textit{get\_js()}) are required. The former function is implemented based on our study result (Table 3.1), and the latter function is provided by the sub-module “JavaScript code syntax analysis”.

#### 3.4.4.1 JavaScript Code Syntax Analysis

It is challenging to generate required JavaScript code as part of an event handler’s input. Because the JavaScript code is executed by associated WebView APIs (such as \textit{loadUrl()}) the values of these APIs’ parameters provide hints. Suppose the JavaScript code extracted from input is \( I \), and the JavaScript code that already exists in associated WebView APIs (such as hard code format) is \( J \). \( I + J \) have complete semantics.

To mitigate the problem, we assume that \( I \) is atomic, i.e., it is a leaf element in the AST (Abstract Syntax Tree) of \( I + J \). We can hence generate \( I \) based on its position in the AST. More specifically, when a WebView API that can execute JavaScript code (such as \textit{WebView.loadUrl()}) is executed, its parameter’s symbolic expression is dumped. Then, by replacing \( I \) with a specific concrete string (such as a randomized string), the concrete string of the parameter (i.e., \( I + J \)) is generated. Next, by applying a JavaScript interpreter engine (such as Mozilla Rhino 1.6) in \( I + J \), AST is generated. After that, \( I \)’s semantics can be understood by checking AST’s semantics and locating \( I \) in AST. Finally, concrete JavaScript code of \( I \) can be generated.
Algorithm 1 Event Order Generation

Input:
1: EH : all event handlers;
2: P(eh) : return all paths in the event handler eh;
3: NS(eh, p, insn) : return all non-symbolic conditional statements before the instruction insn in the path p of the event handler eh;
4: get_origin_variables(eh, p, insn, v) : return the variable v’s origin variables that influence v’s value;
5: get_origin_values(eh, p, insn, v, value, O) : return the required values for all origin variables that can assign value to v.

Output: the event order R

1: function GENERATE_EVENT_HANDLER_ORDER(eh, p, expect_insn)
2: for ns in NS(eh, p, expect_insn) do
3: c ← ns’s condition expression
4: v ← c’s value
5: r ← resolve_event_handler_dependency(eh, p, ns, c, v) ▷ Depending on which branch is taken, v is true or false.
6: if FAILURE == r then
7: return FAILURE
8: end if
9: end for
10: return SUCCESS
11: end function

12: function RESOLVE_EVENT_HANDLER_DEPENDENCY(eh, p, insn, variable, value)
13: O ← get_origin_variables(eh, p, insn, variable)
14: if O == φ then
15: R ← ∅ return FAILURE
16: end if
17: for o in O do
18: if o ∈ eh’s parameters then
19: R.add(<eh, p>)
20: end if
21: end for
22: for (o_i, v_i) in get_origin_values(eh, p, insn, value, O) do ▷ Rerun symbolic execution on the path p to compute each origin variable’s expected value
23: for eh’ in E do
24: for p’ in P(eh’) do
25: insn’ ← the instruction writing o_i
26: r ← resolve_event_handler_dependency(eh’, p’, insn’, o_i, v_i)
27: if FAILURE == r then
28: R ← ∅ return FAILURE
29: end if
30: end if
31: end for
32: end for
33: end for
34: return SUCCESS
35: end function

Algorithm 2 Exploit Code Generation

Input:
1: EO : the event handler execution order, which is the set of the pair <eh, p>;
2: get_input(eh, p) : return eh’s input that can guide the app to execute p;
3: get_web_trigger_code(eh, parameter): return web code that can trigger eh and pass parameter to eh
4: get_js(eh, p) : return required JavaScript code
5: Output: the exploit code X

1: function GENERATE_EXPLOIT_CODE(eh, p)
2: for <eh_h, p_h> in EO do
3: X += gen_js(eh_h, p_h)
4: input ← get_input(eh_h, (p_h))
5: X += gen_event_trigger_code(eh_h, input)
6: end for
7: end function
We use the code in Figure 3.3 to illustrate how this sub-module works. In the event handler `shouldOverrideUrlLoading()`, I is passed to `getId()` and executed to receive sensitive information. To automatically generate concrete JavaScript code of I, `loadUrl()`’s parameter is firstly dumped. Suppose the device ID is “1234”. The parameter’s symbolic expression is then ‘javascript: + Uri.<init>(InputUrl).getHost().split(“.”)[2] + ("1234")’. By replacing the symbolic data with a concrete string (such as “x”), a concrete example code of I + J may be ‘javascript:x(1234)’.

Next, AST (Figure 3.6) can be generated by applying Rhino in the JavaScript code “x(1234)”. By locating x in AST, we can find that x is a function name, and the function has only one string parameter. Hence, a JavaScript function (such as `steal_device_id()` in Listing 3.2) that satisfies the requirement can be defined in advance, and then the function name is passed to the event handler `shouldOverrideUrlLoading()`.

### 3.5 Evaluation

To evaluate EOEDroid, we implemented it on Android 4.3, and deployed it in a Nexus 10 smartphone. Given apps, we started the random UI exploration tool Android Monkey [36] to trigger as many WebView components as possible.

Note that it is challenging to automatically trigger a UI component. To mitigate the problem, we run Monkey to simulate users’ behaviors. Furthermore, we also use Monkey as the first-layer filter. The intuition is that if WebView is an important part of the app, it will be likely triggered in this way. Thus we reduce our workload by only considering the apps whose WebView components are successfully triggered in our dataset (Section 3.5.1).

Once a WebView complement is triggered, complete fuzzing code is injected to trigger all event handlers. More specifically, when WebView is going to connect to a web server, we start a crawler...
to check whether an HTTP link is involved in the connection. We limit the crawling depth in three levels. If there is an HTTP link, man-in-the-middle attacks is performed (Section 3.2.2). The proxy tool “mitmproxy” [37] is used to inject web event trigger (fuzzing) code, which is generated based on the study result (Section 3.4.1). Hence, once the injected code is loaded and executed in WebView, all event handlers are triggered, and then, EOEDroid is started to analyze them.

3.5.1 Dataset

In our evaluation, we collected apps as our evaluation dataset from two different app groups based on whether the WebView component could be triggered at run time. Both these two groups were collected from the Android official store Google Play. The first app group consists of 13,000 popular apps that we crawled from 26 categories, and extracted 500 most popular free apps for each category. The other app group contains 220 browser apps, which were collected by searching the key word ‘web browser’ in Google Play.

Finally, 3,652 apps were totally collected as our dataset, with 3,552 apps from the first app group and 212 apps from the second app group.

3.5.2 Findings

Our experiment casts light on the usage of event handlers in real-world hybrid apps. It also reveals interesting facts about EOE in hybrid apps.

3.5.2.1 Usage of Event Handlers

Figure 3.7 shows the distribution of the usage of top 20 event handlers. shouldOverrideUrlLoading() and onPageFinished() are the two most frequently used event handlers.

We also found most hybrid apps define their own event handlers. In our dataset (Section 3.5.1), 3,440 of 3,652 (94.2%) hybrid apps implemented their event handlers. It is clear that event handlers are in widespread use in real-world apps. Next we discuss the typical scenarios in which event handlers are used in apps.

- Access Control: Event handlers can be applied to perform access control on the communication to be accessed, and the content to be loaded in WebView. For instance, shouldInterceptRequest() can check the content requested by web code. If the content is not expected, the event handler can directly return null to reject the access.
**Customized URL Scheme**: Event handlers can be used to support customized URLs. For instance, the link “tel:xx” and “smsto:xx” can be supported to make a phone call and send a text message.

**Event Driven Authentication**: Using customized URL schemes, event handlers can also be applied to perform authentication. Consider that `shouldOverrideUrlLoading()` supports a customized URL scheme “sdk”. When the URL “sdk://auth_request” is received, the event handler redirects WebView to the authentication web site, while specifying the redirection URL as “sdk://auth_success”. Hence, when the URL “sdk://auth_success” is received by the event handler, the event handler can learn the authentication is successfully done.

### 3.5.2.2 EOE in Event Handlers

By applying EOEDroid on the 3,652 hybrid apps, we successfully identified 97 vulnerabilities in 58 hybrid apps, as briefly shown in Table 3.2. We also find several other interesting findings:

- **Distribution of vulnerable Event Handlers**: We found that most vulnerabilities (96/97) existed in the event handler `shouldOverrideUrlLoading()`. The remaining two vulnerabilities were found in `onCreateWindow()` and `onReceivedHttpAuthRequest()`.

- **Phishing**: We found the usage of the API `loadUrl()` to load new content in WebView likely introduced this type of vulnerabilities. It is mainly because developers wrongly assume the code loaded in WebView is trustable, and do not set up security checks before the sensitive API is called. In some apps, even though security checks were provided, these checks were incompetent to protect the critical functionalities and could be evaded. Take the following code as the example.
Adversaries could still hit the sensitive API by feeding the input ‘http://attacker.com/malicious/code?from=developer.com’.

```java
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    else if (url.contains("developer.com")) {
        view.loadUrl(url);
        return true;
    }
}
```

- **Cross-Frame DOM Manipulation**: As shown in Table 3.2, different from phishing, there were only a few cross-frame DOM manipulation vulnerabilities, even though `loadUrl()`’s parameter was totally controlled by adversaries. This is because that it is challenging to transfer the prefix string “javascript:” from the web code to the native code. Typically, in the web context, the prefix string “javascript:” is directly handled by JavaScript engine, rather than triggering any web events. However, using tricks it is still possible to deliver the prefix string. EOEDroid successfully discovered two vulnerable event handlers that could be leveraged to pass JavaScript code to the native context and execute the code. More details are discussed in our case studies in Section 3.5.3.1.

- **Sensitive Information Leakage**: In this category, EOEDroid successfully caught 26 vulnerable event handlers that could be utilized to steal Android ID. The further study showed that all of them were caused by an ad lib. The remaining 4 vulnerabilities were found in high profile browser apps. The first vulnerable event handler (from “com.webroot.xxx”) could be leveraged to leak the phone number to a public log file, which could be accessed by any app. The second vulnerable event handler (from “com.kiddoware.xxx”) could be triggered to leak IMEI. The third event handler (from “reactivephone.xxx”) could be exploited to steal GPS location information using the input in a specific format. More specifically, if the URL to be accessed contained the string “latitude,longitude”, the real GPS location data were retrieved to replace the string.
The last vulnerable event handler (from “com.mx.xxx”) was interesting, which contained a potential backdoor that could be used to steal sensitive information, such as IMEI. Although developers had attempted to close the backdoor, EOEDroid found that it was still possible for adversaries to leverage the backdoor by changing the program state through the manipulation of execution orders of gadgets. More details are shown in our case study in Section 3.5.3.2.

- **Local Resource Access**: One vulnerable app was found that it could allow adversaries to access local database. Even though this app checked the origin information of web code that was going to access the database, it could still be bypassed by containing the developer website name.

- **Intent Abuse**: One of the vulnerabilities was found in the event handler of the Korean Air app, which was allowed to send arbitrary intent message. Furthermore, the event handler also suffered from phishing attacks and cross-frame DOM manipulation.

Other ten vulnerabilities were found in browser apps. It was mainly because browser apps aimed to support the popular scheme “intent://”. However, these apps did not check the origin information, and specify the action or destination class, which might cause serious problems, as demonstrated by Wang et al. [14].

### 3.5.3 Case Studies

<table>
<thead>
<tr>
<th>App</th>
<th>Input Format</th>
<th>Table 3.3: The Input Format of the Two Vulnerable Apps Shown in Case 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.exsoul.xxx</td>
<td>“exsoul://id=[0-9]{8}&amp;url=”</td>
<td></td>
</tr>
<tr>
<td>com.fevdev.xxx</td>
<td>“intent://…fallback_url=”</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.5.3.1 Case 1: Cross-Frame DOM Manipulation

This section presents two vulnerable apps that suffer from cross-frame manipulation attacks. To transfer the prefix string “javascript:“, the input is crafted following the input format shown in Table 3.3. When the input is received and parsed by the event handler `shouldOverrideUrlLoading()`, the content $l$ of “url” and “fallback_url” is extracted and then fed into a sensitive API `loadUrl()`. Hence, if $l$ is in the format “javascript:...”, the JavaScript code can be then executed in the main frame.
3.5.3.2 Case 2 : Leveraging a Closed Backdoor

This high profile app has been downloaded more than 10 million times. The Listing 3.3 shows a code snippet of the vulnerable event handler. In this app, the variable flag (Line 1) is initially false. When the event handler shouldOverrideUrlLoading() is triggered, several conditional statements are determined relying on the flag (Line 12) and the URL. In Line 24, the URL is saved to a local variable, and then “%IMEI%” is replaced with real IMEI.

```java
flag = false;
pubic void onPageFinished(WebView view, String url) {
    flag = true;
    ... 
}
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    ... 
    url = url.toLowerCase();
    if (!flag)
...
else {
    if (url.startsWith("http://") || url.startsWith("https://")) ...
    else if (url.startsWith("file://")||url.startsWith("content://")) ...
    else if (url.startsWith("mx")) ...
    else {
        if (url.contains("app_name")) {
            String tmpstr = url;
            // read imei from shared preference
            String i = PreferenceManager.getDefaultSharedPreferences(this).getString("imei", "");
            tmpstr = tmpstr.replaceAll("%IMEI%", i)
            ...
            // send a Intent message containing tmpstr
            Intent intent = new ...
            intent.setData(Uri.parse(tmpstr));
            startActivity(intent)
            ...
```  

Listing 3.3: Code Snippet Extracted From the Example in Case 2

By applying EOEDroid on this app, the vulnerable event handler’s path constraints are collected, which are shown as follows.

(1) InputUrl.startsWith("http://") == 0
(2) InputUrl.startsWith("https://") == 0
(3) InputUrl.startsWith("file://") == 0
(4) InputUrl.startsWith("content://") == 0
(5) InputUrl.startsWith("mx") == 0
(6) InputUrl.contains("app_name") == 1
(7) flag == 1
(8) InputUrl.contains("%IMEI%") == 1

All constraints can be satisfied except (7). By addressing the event handler dependency problem on (7), the event handler execution order is generated: onPageFinished() → shouldOverrideUrl
However, due to the trigger constraint (Section 3.4.1), we found `onPageFinished()` was executed after `shouldOverrideUrlLoading()`. Hence, to generate the required execution order, the web page should be refreshed as follows.

(1) `<script> window.location.reload(true); </script>`

Then, the web code that can guide `shouldOverrideUrlLoading()` to execute the sensitive API `getDeviceId()` is shown as follows, if assuming FTP is supported by users’ phone.

(2) `<iframe src="ftp://attacker.com/app_name?imei=%IMEI%"/>`

### 3.5.4 Performance and Accuracy

The performance and accuracy of EOEDroid may be impacted by our symbolic execution implementation, where several heuristics are leveraged to mitigate the path explosion problem. Admitting that these heuristics may cause over approximation and/or inaccuracy to our analysis, they help us make a good tradeoff between performance and accuracy. In this section, we presented more evaluation details, and showed that our current system performance and accuracy were acceptable.

For each app, the average successful analysis time of EOEDroid is around 4.2 minutes, including 3.4 minutes for the event handler analysis. Considering our tool is designed to analyze apps offline, the overhead is acceptable.

We use false positives (FP) and false negatives (FN) to measure EOEDroid’s accuracy. We define a FP as that a non-vulnerable event handler is flagged as vulnerable, and a FN as that a vulnerable event handler is identified as non-vulnerable. More details are shown below:

- **False Positives**: We manually analyzed all vulnerable event handlers by running the exploit code generated by EOEDroid. Finally, we found that all vulnerabilities were successfully triggered, which indicated EOEDroid’s FP rate was low.

- **False Negatives**: To confirm false negatives, we randomly selected 200 apps from the hybrid apps that were flagged as non-vulnerable by EOEDroid. By carefully manually checking their event handlers, we found all apps were non-vulnerable except two apps. Our further study on these two apps showed that the main reason was that the SMT solver failed to resolve some path constraints
that contained multiple regular expressions and string split operations. This still represents a low
FN rate for EOEDroid.

3.6 EOE Countermeasure Discussion

The key to counter EOE is that apps should only allow trustable web code to access critical
functionalities in event handlers. To achieve this, apps should first fully use HTTPS in all commu-
nications, which will effectively reduce the attack surface. Second, when a critical functionality
is called through an event handler, the frame level and origin information of web code should be
carefully checked.

The newest version of Android provides a new setting that only allows web code downloaded
over HTTPS to access `shouldOverrideUrlLoading()`, and also includes more information in the
event handler’s parameters, such as the frame level and origin information of web code. Hence,
we strongly recommend developers port their apps to the new version, and leverage these security
information in their development.

3.7 System Limitations and Future Work

EOEDroid is not perfect. First, currently we simply use Monkey to trigger WebView. Exploring
all possible UI components is a difficult issue, though orthogonal to this research. Second, in
EOEDroid, we do not solve all implicit flow problems, instead only focus on array-indexing type
operations, which are frequently used in event handlers. Finally, we do not handle all native code in
Android, instead only model important native code such as `system.arraycopy()`. In future work, we
plan to explore solutions in these directions to improve EOEDroid.

3.8 Summary

In this chapter, we present our study on web events. We find in contrast to regular browsers, web
events are handled in a totally different way (i.e., the event handler bridge) in WebView. However,
there are no protection on the event handler bridge. Any web content can freely access sensitive
functionalities inside event handlers. In this chapter, we demonstrate this unauthorized access cause
serious consequences.
4. STUDYING AND PROTECTING HYBRID POSTMESSAGES

4.1 Overview

Motivation Cross-origin communication using the HTML5 postMessage facility [8] has been a popular and often necessary technique on the web platform. It relaxes the restrictions enforced by the well-known same origin policy (SOP) security model [6] by allowing bidirectional messaging between mutually distrusting web frames or windows. With the increasing amalgamation of the web and mobile platforms, postMessage has also found utility on WebView and hybrid apps.

In addition to cross-origin communication, the hybrid mobile app model introduces the necessity for cross-platform communication between the web platform and the mobile platform. Not only do hybrid apps need to communicate between different origins loaded in a WebView, they must also facilitate communication between those origins and the native layer (e.g., the Android Java code). While hybrid apps can already utilize web-mobile bridges (such as the JavaScript Bridge) [9] for cross-platform execution, cross-platform messaging in the form of HTML5 postMessage is not available.

Android 6.0 partially addresses this shortcoming by providing a new cross-platform API called postWebMessage(). However, this API is plagued by the notorious Android fragmentation problem [22] and does not scale well. Moreover, it is limited to unidirectional communication from native to web but does not support communication from web to native. In our empirical study on a set of popular hybrid apps, we found postWebMessage() was rarely used in practice.

As a result, developers have resorted to customizing postMessage in hybrid apps using ad-hoc methods such as web-mobile bridges (see Figure 4.1). In general, this customization treats the native context as a new different-origin frame. This results in “hybrid postMessage”, which provides both native-to-web ($N \rightarrow W$) and web-to-native ($W \rightarrow N$) messaging.

Unfortunately, while hybrid postMessage provides easy and convenient cross-platform communication, it also opens a door for adversaries through code injection attacks (such as web or network...
Figure 4.1: Overview of Regular and Hybrid postMessage

Figure 4.2: Sending Messages Through Regular and Hybrid postMessage

Figure 4.3: Receiving Messages Through Regular and Hybrid postMessage
attacks shown in Figure 4.1) to launch denial-of-service (DoS) attacks, steal sensitive information, silently access local hardware (such as the microphone), and perform other nefarious actions. The security problem is rooted in the loss of the origin information when messages move across the web and native layers. More specifically, the origin information of the message sender (source) and message receiver (target) is either not respected or totally lost. There are two main reasons:

1) Hybrid postMessage may not provide any interface to allow the message sender to specify the **target origin**, which is critical in the regular HTML5 postMessage to control the message receiver;

2) Hybrid postMessage may not provide the **source origin** of a received message, which means it is impossible for the message receiver to validate the message. This adds a new layer to the known security problem of client-side validation (CSV) in the web platform [19] [18] [20]. For convenience, we term the novel security issue caused by hybrid postMessage “**Origin Stripping Vulnerability**” (OSV).

Figures 4.2-4.3 illustrate that OSV may compromise the confidentiality and integrity of cross-platform communication. Consider that adversaries inject malicious code into WebView through web or network attacks. The malicious code may leverage hybrid postMessage to passively receive and monitor messages that contain sensitive information, or actively send messages to arbitrary message receivers to access their internal functionalities or data.

In Figure 4.2-a, Alice sends a message to Bob through the regular postMessage. The message contains the message content (“How are you doing?”), and the target origin (Bob), which determines that only Bob can receive the message. However, hybrid postMessage breaks this convention by stripping the target origin (Figure 4.2-b). As a result, Mallory, an adversary who runs malicious code in another web frame can receive and read the message. If the message carries sensitive information, Mallory can easily violate the confidentiality of Alice and Bob’s communication. In Figure 4.3-a, Bob is receiving a message from Alice. When the message arrives, Bob can validate that the source origin of the message is Alice. However, hybrid postMessage loses the source origin information (Figure 4.3-b), which means that it is impossible for Bob to conduct validation. Therefore, Mallory may send a message (“What’s your password?”) to Bob and access its confidential data.

Although the detailed implementation guideline and security model for postMessage are established in HTML5 [8], it is challenging for developers to implement hybrid postMessage conforming to it. The main obstacle is the gap between the web and native platforms. Web-mobile bridges may
be applied to fill the gap. However, as shown in prior work [9] [10] [13], these bridges are often the cause of security vulnerabilities, because any code loaded in WebView may freely access them.

For example, we found hybrid postMessage was implemented in the popular “Facebook React Native” framework using the JavaScript Bridge. As shown in Listing 4.1, the crucial JavaScript method `window.postMessage()` is rewritten to allow all messages to be sent to the native frame. However, due to the intrinsic weakness of the JavaScript Bridge, the native frame cannot distinguish the identity of the message senders, or even safely obtain the source origin.

```
WebView.loadUrl("javascript:
"window.originalPostMessage = window.postMessage," +
"window.postMessage = function(data) {
  // The source origin is lost.
  // Only data is transferred through a JavaScript Bridge.
  __REACT_WEB_VIEW_BRIDGE.postMessage(String(data));
}"
"

Listing 4.1: Implementing $W \rightarrow N$ in Facebook React Native
```

Existing defense solutions, such as NoFrak [9], Draco [10], MobileIFC [15], WIREframe [16], and HybridGuard [17], were designed to provide protection for WebView and web-mobile bridges by either extending SOP to the native layer, or enforcing security policies to offer access control. However, they are circumscribed to prevent OSV for several reasons. First, most existing defense solutions can only protect $W \rightarrow N$, but not $N \rightarrow W$. Only WIREframe can offer protection in two directions. However, unfortunately, its security policies enforced in $N \rightarrow W$ may be under the control of adversaries. Second, existing defense solutions are coarse-grained, and may have high false negatives. Their provided protection is usually performed based on the origins of web frames, and thus it is difficult for them to limit the behaviors of the embedded JavaScript code.

Moreover, existing defense solutions may be hindered by the blend of OSV and CSV vulnerabilities. Consider a scenario in Figure 4.4 which we found in a real-world advertisement library. In the web platform, a nested third-party iframe can send messages to the main frame, where a message handler receives the messages but does not validate their source origins (i.e., CSV vulnerability). It then forwards the received messages to the native frame through hybrid postMessage. After that, the defense solutions are enforced to protect $W \rightarrow N$. They attempt to obtain the message sender’s origin to apply their policies. However, they can only obtain the main frame’s origin, rather than the real message sender’s origin (i.e., the third-party frame’s).

CSV detection and defense solutions [19] [18] [20] may be applied to mitigate the above threat.
However, their performance may also be limited. They rely on the analysis or detection of source origins of received messages. The messages received by the message handler of the main frame include not only messages (“$M_1$”) from the third-party frame, but also messages (“$M_2$”) from the native frame. They may protect $M_1$, but not $M_2$, because the source origin of $M_2$ may not be provided in hybrid postMessage.

We first conduct the systematic study on hybrid postMessage and identify the novel security issue “OSV”. Second, to evaluate the prevalence and presence of hybrid postMessage and OSV in Android hybrid apps, we design a lightweight detection tool, called OSV-Hunter, that can help developers and analysts identify hybrid postMessage and discover potential OSVs. Different from existing detection tools [13, 38], which fall short of filling the web-mobile gap and tracking origins, OSV-Hunter automatically discovers message senders and receivers, and analyzes the semantics of the link between them.

Third, we evaluate OSV-Hunter using a set of popular apps. We found 74 apps implemented hybrid postMessage, and all these apps suffered from OSV, which may be exploited by adversaries to perform denial of service (DoS), local critical hardware device access (such as real-time microphone monitoring), data race, internal data manipulation, and so on. Several popular frameworks and libraries suffer from OSV, such as Facebook React Native and Google cloud print. Several high-profile apps are also impacted, such as Adobe Reader and WPS office. In addition to the Android platform, OSV also impacts other platforms (like iOS), since the hybrid postMessage APIs of vulnerable frameworks (such as Facebook React Native) are also available in these platforms.

We have reported all our findings to the Android security team, and the relevant framework, library, or app developers. We are actively helping them fix the discovered OSV problem. The Facebook security team has confirmed our findings in the React Native development framework, and they also admitted that it was difficult to eliminate the security problem caused by OSV in their current implementation. Instead, they explicitly added a security warning in their development.
Lastly, motivated by the above difficulty faced by developers to eliminate OSV, we design and implement a set of new hybrid postMessage APIs in the newest WebView, called OSV-Free. Our evaluation shows that OSV-Free is secure and fast, and it is generic and resilient to the notorious Android fragmentation problem. We also demonstrate that OSV-Free is easy to use, by applying OSV-Free to harden the complex “Facebook React Native” framework. OSV-Free is open source, and its source code and more implementation and evaluation details are available online: http://success.cse.tamu.edu/lab/osv-free.php.

4.2 The OSV Problem Definition

We define OSV based on the possible violation on postMessage’s security model (or design guideline) [8], which is defined as follows. We assume \( SF \) and \( RF \) are the frames which a message sender and its corresponding message receiver belong to respectively. The security model can be defined using the following two rules:

- **Rule I**: When a message is being sent, its target origin \( T_{\text{origin}} \) should satisfy that 1) \( T_{\text{origin}} \) is specified or implied; 2) \( T_{\text{origin}} = R_{\text{origin}} \) or \( T_{\text{origin}} = \ast \).

- **Rule II**: When a message is being received, its source origin \( S_{\text{origin}} \) should meet that 1) \( S_{\text{origin}} \) is defined; 2) \( S_{\text{origin}} = S_{\text{origin}} \); 3) \( S_{\text{origin}} \) is unique for \( SF \).

Hence, if the above two rules are not followed in hybrid postMessage, OSV may exist. For convenience, we define four sub-vulnerabilities (i.e., \( V_1 \) to \( V_4 \)) based on the violation of the above two rules in two directions, as shown Table 4.1.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Native → Web</th>
<th>Web → Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violated Rule</td>
<td>Rule I</td>
<td>Rule II</td>
</tr>
<tr>
<td>Sub-Vulnerability Type</td>
<td>( V_1 )</td>
<td>( V_2 )</td>
</tr>
</tbody>
</table>

**Table 4.1: Definitions of Four Sub-Types of OSV**

The four OSV sub-vulnerabilities disclose more attack patterns than those discussed in Section 1.3. For example, consider a scenario in Figure 4.5. Alice and Mallory are web frames, while
Bob is a native frame. Bob sends messages to Alice through hybrid postMessage. Due to $V_2$, the source origin of the native frame may not be provided or not unique. Mallory may be able to forge a message with the same source origin, by creating a nested controllable iframe that has the same origin, and then sending a crafted message from the new iframe to Alice using the typical web postMessage. When Alice receives the message, Alice notices that the source origin is the same as the native frame’s. As a result, Alice treats Mallory as Bob and allows Mallory to access the internal functionalities. If Alice carries critical functionalities or data, serious consequences may be caused.

To prevent $V_2$, it is important to ensure the uniqueness of the source origin of the native frame. However, even if the source origin is unique, it is hard to manage and may still introduce security issues. For example, to receive messages from the native frame, Alice may need to relax its validation logic for all incoming messages, which may cause CSV. In our evaluation (Section 4.4), we show such problems exist in real-world apps.

4.3 OSV-Hunter Design and Implementation

4.3.1 Design Observations

OSV-Hunter is designed to identify apps with actual hybrid postMessage implementations, and vet such implementations against OSV in a lightweight and generic way, based on several key insights and observations:

- *The JavaScript method window.postMessage() should be a message sender of hybrid postMessage:*
“window.postMessage()” may be 1) directly called in web frames, or 2) indirectly invoked in the native frame through WebView JavaScript code loading APIs (such as WebView.loadUrl()). For example, the following Java code sends native data (i.e., content) from the native frame to the main web frame:

```java
WebView.loadUrl("javascript:window.postMessage('" + content + ", " + '*')");
```

In both cases above, “window.postMessage()” should be a communication launcher (message sender). To discover its corresponding message receiver, its parameter, especially the message content c, should be tracked. If c appears in a function f of the opposite frame, f is likely a message receiver.

To implement it, a special and unique string ID, such as “PM_Case1_<Random Number>” for the first case and “PM_Case2_<Random Number>” for the second case, is injected into c and tracked. More specifically, in the native frame, all native function invocations should be checked to verify if their parameters contain ID. If ID is found, there should be a link between window.postMessage() and the firstly found native function. For the second case, all message handlers of web frames should be monitored. Once ID appears in the message handlers of a web frame, there should also be a link from the native function that executes window.postMessage() through WebView.loadUrl() to the message handlers of the web frame.

- **A message handler of a web frame may be a message proxy, or receiver:** It is possible for a message handler to 1) receive messages from the native frame (i.e., N→W), or 2) forward messages received from other web frames to the native frame (i.e., W→N). The above possibilities can be verified respectively. For the first possibility, the value of the parameter of the message handler should be monitored to check if ID exists. For the second possibility, similar with how window.postMessage() is handled, the received message content of the message handler should be tracked. For this purpose, if no ID exists in the received message content, a new ID, such as “MH_ForwadingMessage_<Random Number>”, should be injected into the received message content. When the message content is forwarded, if the ID appears in a native function in the native frame, the native function is likely a message receiver. Hence, there may be a link between the message handler of the web frame and the native function of the native frame.

- **The APIs (such as web-mobile bridges) that provide cross-platform functionalities are likely**
utilized to implement hybrid postMessage: For example, apps may execute JavaScript code to trigger a message event using the JavaScript execution APIs (like `WebView.loadUrl()`). Hence, the parameters of these APIs should be carefully handled. Additionally, `WebView.postMessage()` should also be monitored, since it can be used for N→W messaging.

4.3.2 Design Details

Guided by these observations, we designed two main phases in OSV-Hunter containing a number of sub-modules, as shown in Figure 4.6. In Phase#1, “hybrid postMessage Identification” fills the semantic gap between the native and web frames, and identifies the implementation of hybrid postMessage. In Phase#2, “Message Origin Analysis” collects all delivered messages between message senders and receivers, and performs origin analysis to determine the existence of OSV.

More specifically, given a hybrid app, a fuzzing module “Tester” is first started to 1) trigger as many WebView components as possible, and 2) attempt to trigger message senders of both the native and web frames. When a WebView component appears, the loaded HTML/JavaScript code is analyzed and instrumented to discover potential message senders and receivers in web frames. It is achieved by the modules “HTML/JS Analysis” and “HTML/JS Instrumentation”. To monitor all messages cross the native frame, the native code is instrumented by the module “Native Code Instrumentation”. Then, by collecting and analyzing the information generated by above modules, message senders and receivers can be identified and linked together, which is done by the module “Source & Target Link Generation”. Finally, the “Message Content Collection” module dumps all content of delivered messages, which are further analyzed in “Message Origin Analysis” to determine the existence of OSV.

We next describe the design details of each sub-module.

4.3.2.1 Hybrid postMessage Identification

In this section, we presented the design details of each sub-module:

- **Tester**: To trigger WebView and run native code (for triggering message senders in the native frame), we use a random UI explorer “Monkey” to simulate users’ behaviors [36]. Once WebView is started, network activities may occur. Then, the pre-defined JavaScript fuzzing code is injected into network traffic based on our threat model, which is done using the popular proxy tool
“mitmproxy” [37]. Please note that in order to perform network attacks, network links are crawled to check if a HTTP link can be navigated. For convenience, we limit the crawl depth as three.

The above injected JavaScript fuzzing code is designed to drive the test on $W \rightarrow N$. Usually, the JavaScript methods that send messages (e.g., `window.postMessage()`) are called in all kinds of environments. It is implemented mainly based on existing work, such as the work of Schwenk et al. [40].

Please note that even when a WebView component is started, Monkey is still kept running. It is because this is helpful to trigger as much native code as possible, and thus, message senders in the native frame may be triggered.

- **HTML/JS Analysis and Instrumentation**: When HTML is going to be loaded in WebView, the HTML content is analyzed and instrumented as follows. First, the first page of the HTML code and all JavaScript code are cached in local storage for further instrumentation. Please note that JavaScript code will be handled by JS Analysis and JS Instrumentation later. Then, all important remote links in HTML are converted to local links, such as the link specified by the “src” attribute of the element `<script>`. So that the local instrumented content can be loaded in run-time, instead. To analyze and instrument the content of nested frames, an extra WebView event handler implementation of `shouldInterceptRequest()` is imported to handle the nested frame loading event, and control the content of nested frames.

JavaScript code is analyzed and instrumented as follows. First, message senders (such as `window.postMessage()`) are identified and handled by inserting extra instructions to print necessary information (like the origin of the web frame that the message sender belongs to), and instrumenting the method parameters, such as inserting ID if ID does not exist.

Then, message handlers are processed. To hook a message handler method $f$, a wrapper function $f'$, which has the same function prototype with $f$, is defined to replace $f$. In $f'$, all necessary information is printed, such as the web frame’s origin and the method parameters, and then, $f$ is called and fed with $f'$’s parameters. In this way, the original semantic of the web code is kept. To track the message content received by $f'$, ID is injected.

- **Native Code Instrumentation**: Native code is instrumented to discover all message sending and
receiving activities. To discover a message receiver of $W \to N$, all native functions’ parameters are checked, which is done by instrumenting the run-time interpreter in Android ART (i.e., $DoCall()$ in the file “interpreter_common.cc”). If a parameter is a string, its low-level object $StringObject$ is retrieved for further analysis, such as converting it back to a normal string, and checking if ID exists.

To discover the message sender of $N \to W$, critical APIs (such as $WebView.loadUrl()$ and $WebView.postMessage()$) are monitored, which is done by instrumenting the Android framework code to record the parameters of these APIs. Please note that if the parameters of $WebView.loadUrl()$ are JavaScript code, the JavaScript code will be analyzed by the sub-module $JS$ Analysis and $Instrumentation$. If $postWebMessage()$ is called, the message content to be sent is also instrumented by inserting ID.

- **Message Source and Target Link Generation:** Guided by the insight and observation (Section 4.3), message senders and receivers in both native and web frames can be identified. First, all log information that is generated by $HTML/JS$ Analysis and $Instrumentation$, and Native Code Instrumentation is collected. Then, the log is filtered using the special format of ID. Finally, message senders and receivers can be linked together by matching ID. Since each ID is unique, the established links are also unique.

4.3.2.2 Message Origin Analysis

- **Message Content Collection:** To determine the existence of OSV, the content of all delivered messages are fully dumped and collected. In the native frame, the content of all related low level objects (e.g., $StringObject$) are dumped. In the web frames, the content of all JavaScript variables is printed. If a variable is an object, all its fields (including inherited fields), and the corresponding values are logged. All other critical logs are also gathered, such as the ones containing origin information of message senders and receivers.

- **Vulnerability Determination:** OSV can be determined based on the definitions of the four sub-vulnerabilities (Section 4.2). More specifically, $V_1$ and $V_4$ can be automatically determined by checking if the origin information is contained in relevant APIs or delivered messages using the
information collected by the sub-module “Message Content Collection”. However, for $V_2$ and $V_3$, it is challenging to analyze the origin information, since the native frame does not have an explicit origin. Hence, manual efforts may be needed in this phase.

4.3.3 Implementation

We implement OSV-Hunter by instrumenting the Android source code (the 6.0 version). All modules are built from scratch, except HTML/JavaScript analysis and instrumentation. The HTML analysis and instrumentation module is built based on JSoup 1.10.3, and the JavaScript analysis and instrumentation module relies on Mozilla Rhino 1.7.7. JSoup and Rhino are written in Java, and added into WebView as libraries. Please note that Rhino is very powerful, but in OSV-Hunter, we only statically use it to generate and manipulate AST (Abstract Syntax Tree) of target JavaScript code, and convert AST back to new JavaScript code.

4.4 Study of Hybrid postMessage and OSV

4.4.1 Data Set

To build an appropriate data set for the evaluation, we crawled 17K most popular free apps from 32 categories (top 540 apps for each category) in Google Play in July 2017. However, not all apps should be analyzed. For example, some apps do not even use WebView.

Therefore, to reduce the workload, we establish two qualifications to narrow down our data set. The first one is that apps must contain at least one WebView instance. Thus, we use the keyword “WebView” on apps’ disassembled code to statically filter apps.

The other qualification is that apps should contain postMessage-related code. To avoid potential false negatives, both regular and hybrid postMessage should be included. For this purpose, we use the background knowledge to establish our static filter. An expected app should contain postMessage-related keywords such as: 1) “postMessage”, which is used to send messages; 2) “WebMessage”, which is frequently contained in official APIs, such as “WebView.postMessage()”; 3) “onmessage”, which is the global message handler; 4) “addEventListener(“message””, which is used to register message handlers.

As a result, 1,104 apps remain as our data set.
4.4.2 Results

In our study, we deployed OSV-Hunter in Nexus 5 to identify apps that contain actual hybrid postMessage implementations. Each app was tested for 10 minutes. Finally, we identified 74 apps that implemented hybrid postMessage and we also found that all these apps were vulnerable.

The results are summarized in Table 4.2. Several popular third-party frameworks or libraries (like Facebook React Native, and Google cloud print) suffer from OSV, and may cause serious consequences, such as remote real-time microphone monitoring, permanent data race, internal data manipulation, denial of service (DoS) and so on. Furthermore, several high-profile apps are impacted. For example, the Google cloud print service in Adobe Reader and WPS office may suffer from DoS attacks due to the OSV.

As shown in Table 4.2, both $N \rightarrow W$ and $W \rightarrow N$ are demanded and implemented by developers. For $N \rightarrow W$, it is supported in the React Native framework, the EclipseSource app, and the WebView official API `WebView.postMessage()`. All the implementations except `WebView.postMessage()` suffer from $V_1$, since the target origin of the message to be sent cannot be specified. All the implementations, including `WebView.postMessage()`, may be impacted by $V_2$, as the source origin is not well provided in the message receiver. More specifically, in the React Native framework, the source origin of $N \rightarrow W$ is “undefined”. It is because a customized data structure is designed to carry the delivered message. In the data structure, the “data” field is set to contain the message content. However, another important field “origin” is not defined. Hence, when a message receiver reads the source origin of a received message, “undefined” is obtained.

Although we did not find a good counter-example to prove the origin “undefined” is wrong for

<table>
<thead>
<tr>
<th>Vulnerability Name (App or Framework)</th>
<th>Impacted Apps / Total Apps</th>
<th>Example App</th>
<th>Vulnerability Type</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook React Native</td>
<td>43/43</td>
<td>com.adb.xxx</td>
<td>Native $\rightarrow$ Web $\rightarrow$ Native</td>
<td>Monitoring Audio, Data Race, Internal Critical Data Manipulation, ...</td>
</tr>
<tr>
<td>Google Print</td>
<td>30/30</td>
<td>com.adb.xxx</td>
<td>Native $\rightarrow$ Web $\rightarrow$ Native</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>Eclipse Source</td>
<td>1/1</td>
<td>com.eclipse.xxx</td>
<td>Native $\rightarrow$ Web $\rightarrow$ Native</td>
<td>Sending a message with a source origin not belonging to itself</td>
</tr>
<tr>
<td>WebView’s postWebMessage()</td>
<td>0/0</td>
<td></td>
<td>Native $\rightarrow$ Web $\rightarrow$ Native</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: The Evaluation Result
the native frame (such as “undefined” may be not unique), “undefined” is meaningless and hard to manage. As discussed in Section 4.2, such meaningless origins may cause more security issues, such as CSV. A similar problem is also found in `WebView.postMessage()`, which provides a meaningless origin (empty string) as the source origin. It is because in the native layer, the internal implementation of `postWebMessage()` does not explicitly define the origin of the native frame, and NULL is used at default. Correspondingly, in the web space, an empty string is treated as the source origin.

Different from the above implementations, the EclipseSource app provides the source origin. However, the origin may not be correct. It is because in this app, the JavaScript method `parent.postMessage()` is hijacked by a JavaScript Bridge, where the origin of the top frame is always used as the message source origin, even when a message is sent from an iframe.

For $W \rightarrow N$, it is implemented in all developers’ hybrid `postMessage` implementations. This suggests $W \rightarrow N$ is highly demanded, and thus the official API `WebView.postMessage()` that provides the simple functionality does not meet the requirement.

However, all $W \rightarrow N$ implementations are also impacted by OSV, especially the sub-vulnerability $V_4$. Note that $V_3$ is not flagged even though the required origin is not transferred. It is because although in $W \rightarrow N$ the target origin cannot be specified, it is implied in the message-sending methods themselves. More specifically, to implement $W \rightarrow N$, developers rewrite the JavaScript method “`window.postMessage()`” to send a message to the native frame at default. Hence, if the native frame is unique, the target origin information should be implied in the APIs themselves, since the native frame is the sole destination. In fact, the native frame is unique. “`window.open()`” may create a new native frame, but it does not influence the original native frame’s uniqueness. It is because the new native frame is totally independent of the original native frame, and web frames can only communicate with their corresponding native frames.

$V_4$ exists in all implementations. All source origins are lost during message delivery. Hence, if malicious code is injected into WebView, the malicious code can freely access the internal functionalities inside the message receiver of the native frame. Section 4.4.4 demonstrates this sub-vulnerability may introduce serious consequences.
4.4.3 Findings

From our study results, we have the following findings:

- **Developers wrongly assume the content loaded in WebView is trustable:** This wrong assumption is reflected in developers’ implementations. For instance, in $N \rightarrow W$, their implementations usually do not provide an interface to specify the target origin. No matter what origin is loaded in the target web frame, the message will be delivered. In $W \rightarrow N$, when the native frame receives a message, the source origin of the message is not provided. This indicates that the content loaded in WebView is fully trusted, which may cause serious consequences.

- **The requirement of a feasible hybrid postMessage implementation may be urgent:** Regular postMessage is still very popular in hybrid mobile apps. However, compared with regular postMessage, a feasible hybrid postMessage implementation is more preferred. For instance, in many apps, $W \rightarrow N$ is implemented by rewriting the JavaScript method `window.postMessage()`, which breaks the regular postMessage functionality.

- **In all web frames, only the main web frame usually has the capability to communicate with the native frame, but some main web frames are treated as message proxies during message delivery:** Within our data set, we found 73/74 (98.6%) apps only allow the main web frame to exchange data with the native frame, and 30/74 (40.5%) apps leverage the main web frame as proxies.

- **The blended vulnerabilities of CSV and OSV exist in real world apps:** 30 apps use the main web frame as message proxies, where both CSV and OSV exist. The blended vulnerabilities may result in that existing WebView defense solutions may be fooled.

- **The official hybrid postMessage APIs are rarely used in practice:** Within our whole dataset, no apps use the official WebView APIs. Compared with developers’ implementations, the functionality provided by `WebView.postMessage()` is too simple.

- **The communication “$W \rightarrow N$” is usually implemented relying on JavaScript Bridge:** JavaScript Bridge opens bridges linking web code with native code. However, as JavaScript Bridge usually does not carry any origin information, OSV is likely caused. Although there are several solutions proposed to protect JavaScript Bridge, all are limited in their ability to prevent OSV.
Figure 4.7: Hybrid postMessage in Facebook React Native

4.4.4 Case Studies

4.4.4.1 Facebook React Native

Facebook React Native is a third-party development framework that allows developers to develop mobile apps purely in JavaScript. It supports several popular mobile platforms (like Android and iOS). Thus, the OSV vulnerability impacts all the supported platforms.

The architecture of the React Native framework is shown in Figure 4.7. In run-time, the running environment is first created. Developers’ JavaScript code “DJ” is parsed and executed by the embedded generic and powerful JavaScript engine “JavaScriptCore”. Through JavaScriptCore, DJ can interact with Android, such as creating native UI components, and handling UI events.

WebView (i.e., customized WebView in Figure 4.7) is also available in the React Native framework. To enable it, it is required for DJ to create a WebView object O as the reference. Listing 4.2 illustrates how to create a WebView object in DJ (Line 9), and let WebView to show a remote web page (Line 13).

```
// A message handler
handleMessage(e) {
  // The message content is saved in e.nativeEvent.data.
  // However, the source origin is lost.
  this.webview.postMessage("[native] received a message : " + e.nativeEvent.data);
}
// Configure UI layout
render() {
  return (<WebView // Create a WebView component 'O'

    // Enable JavaScript
    javaScriptEnabled={true}

    // Load a remote web page in WebView
    source={{uri: "https://developer.com"}}

    // Register a message handler
    onMessage={this.handleMessage}

```

60
In the React Native framework, hybrid postMessage is implemented to allow the communication between $O$ and the JavaScript code loaded in the native WebView component (for convenience, we denote the latter JavaScript code as “$WJ$”). For this purpose, two APIs are added in $O$: 1) `WebView.postMessage()` (Line 5 of Listing 4.2), which sends a message from $O$ to the main web frame of $WJ$; and 2) `WebView.onMessage()` (Line 15 and Lines 2-6 of Listing 4.2), which receives messages from the main web frame of $WJ$.

As discussed in Section 4.4.2, the hybrid postMessage implementation of the React Native framework suffers from OSV. More details are presented as follows.

To support hybrid postMessage, the React Native framework customizes Android WebView, where the origin information is not carefully handled. More specifically, as shown in “Customized WebView” of Figure 4.7, when a message is sent from $WJ$, it first enters the native context (i.e., “Native Customization”) through a pre-imported JavaScript Bridge, where the origin information is lost. Then, the message is delivered to the embedded JavaScript engine, and further forwarded to $O$.

The key implementation is shown in Listing 4.1. In Customized WebView, the JavaScript method `window.postMessage()` is rewritten. So that when `window.postMessage()` is called in $WJ$, the message is redirected to a pre-defined native function in the JavaScript Bridge “__REACT_WEB_VIEW_BRIDGE”. However, during the message delivery, the source origin information is lost.

```
Listing 4.2: Example Code of Creating a WebView Object in $DJ$

```{1}

```
Listing 4.3: Sending Messages to the Main Frame Through WebView.loadUrl() in the Native Context

```

To implement sending a message in the opposite direction, the code shown in Listing 4.3 is used. The message content to be sent is wrapped in a message event (Lines 3-6), and then is dispatched to message handlers in the main web frame (Line 12). Since the message origin is not defined in the
event wrapper, “undefined” appears as the source origin. More importantly, the implementation cannot ensure the code is executed in the correct context (e.g., the target origin may not be right).

Because of the OSV problem, adversaries may be able to send messages to message receivers to access the internal functionalities, or play as message receivers to monitor sensitive information contained in messages. com.altvr.xxx and com.giantfood.xxx are two good examples to demonstrate the problems:

- **Case#1 com.altvr.xxx**: It is designed for VR (Virtual Reality) device management. Users can create events (such as party, concert, and conference) and let others join in them. In addition, even though there are no VR devices, the app can still launch 2-D mode, which is available for most phones.

```javascript
window.postMessage(
     {'
        "method": "enterSpaceForceVR",
        "args": {
            "Url": "<event_url>",
            '
        }, ...
     }
)
```

**Listing 4.4: Example Attack Code to Let Apps Forcefully Join Any Events**

By leveraging OSV, malicious code injected into WebView can freely access the functionality inside the message receiver of O (i.e., `WebView.onMessage()`). As the example attack code (Listing 4.4) shows, adversaries can call the method “enterSpaceForceVR” (Line 3) to let the app silently and forcibly join any events specified by adversaries (i.e., “Url” in Line 5). If the microphone is enabled, adversaries may be able to remotely monitor the microphone.

Hence, a feasible attack scenario for silently monitoring the microphone is that an attacker first logs in developers’ website to create an event, and gets a URL of the created event. Then, the attacker joins the event to wait for victims in advance. After that, the attacker injects crafted malicious code into the victim’s WebView through an embedded third-party JavaScript library. Next, the malicious code triggers hybrid postMessage and calls the “enterSpaceForceVR” method with the pre-obtained event URL as the parameter. After that, the app silently joins in the event controlled by the attacker. Finally, the attacker may start to monitor the victim’s microphone.

Furthermore, the above attack code may also cause data race. When the app is opened, the app usually takes a long time for initialization, especially when the microphone is enabled. At that
period, if the attack code shown in Listing 4.4 is injected and executed, a data race occurs. In our test, the data race can be stably triggered. When a third-party JavaScript lib is fetched by the app’s WebView, adversaries can immediately inject and run attack code. Then, the data race can be triggered. In addition, the influence of the data race is continuous, and can only be avoided by totally cleaning user data, or re-installing the app.

The cause of data race is that once the microphone is enabled, a flag object will be initialized when the app is opened. Before the flag object’s initialization, if the attack code is executed, an exception will be triggered and the app will be crashed.

In the above two attacks, the functionalities inside the message receiver of O can be fully leveraged. It is because due to OSV, the React Native framework does not provide any source origin information for validation.

The implementation of the app’s message receiver is shown in Listing 4.5. When a message is received, the message content is retrieved and parsed (Line 5). Then, the message receiver executes an arbitrary method whose name and arguments are determined by the fields “method” and “args” of the received message (Lines 9). Finally, the execution result “r” is returned through WebView.postMessage() (Line 13).

```javascript
1 // e is a WebView object in O
2 // Registering a message handler
3 e.onMessage = function(t) {
4    // Reading message content to a
5    var a = JSON.parse(t.nativeEvent.data);
6    ...
7    // Executing an arbitrary method in the WebView object e
8    r = e[a.method](a.args);
9    ...
10   // Returning the execution result to WJ
11   e.refs.wv.postMessage(JSON.stringify({..., value: r, ...}));
12 },
```

Listing 4.5: Code Snippet of onMessage() in Case1

- **Case#2 com.giantfood.xxx**: It is a food shopping management app. The operation on users’ cart (i.e., the shopping list) relies on data exchange over hybrid postMessage. In WJ → O, the main frame of WJ can send a command to ask for corresponding actions, such as opening and editing cart, and adding and removing items to or from the cart.

Hence, a feasible attack scenario is that an attacker injects malicious code through an HTTP link,
and then, sends messages through $WJ \rightarrow O$ to manipulate the app’s internal data.

The implementation of the message receiver of $O$ is shown in Listing 4.6. When a message is received, its content is directly parsed and dispatched to the corresponding event handler. Hence, if the content of the transferred message is equal to the values in “SHOPPING_LIST”, all internal functionalities can be accessed.

```
1 // The message receiver in O 'WebView.onMessage()'
2 key: "onMessage",
3 value: function(e) {
4 // Dispatch events based on the message content
5 // However, the message' source origin is not provided for validation
6 switch ((e.nativeEvent.data)) {
7 case SHOPPING_LIST.OPEN:
8 // Dispatch the event
9 (0, N.tagEvent)(SHOPPING_LIST.OPEN);
10 break;
11 case SHOPPING_LIST.EDIT: ...
12 ...
```

Listing 4.6: Code Snippet of onMessage() in Case 2

### 4.4.4.2 Google Cloud Print

The Google cloud print library is designed to provide the cloud print service. It is very popular, and available in many high-profile documentation management apps. The library is usually started by an inter-component communication (i.e., Intent) message that carries the details of the document to be printed (such as file URI and type). Then, it opens a WebView component to load a remote print web page. As shown in Listing 4.7, when the web page is fully loaded (Line 1), a message handler is registered in the native context (Line 4). The message handler works as the message proxy to forward all received messages to the native layer (Lines 7-9). It is done by calling a JavaScript Bridge (Line 8).

```
public void onPageFinished(WebView view, String url) {
    webView.loadUrl("javascript:" +
    "window.addEventListener(" +
    "message", " +
    "window." + JS_INTERFACE + ".onPostMessage(evt.data)" +
    ")");
}
```

Listing 4.7: The Source Code of Registering a Message Handler in Google Print

Please note that although a JavaScript Bridge is used in the message handler of the main web frame, we still count the JavaScript Bridge as part of the implementation of hybrid postMessage.
It is because in this scenario, the native function ("onPostMessage()") of the JavaScript Bridge is the essential message receiver that handles the received message content. It is also reflected in its implementation, which is shown in Listing 4.8. In the native function, the message content is handled and parsed. If it is equal to a constant value, which is saved in the variable “CLOSE_POST_MESSAGE_NAME”, the service will be finished.

```
public void onPostMessage(String message) {
    // CLOSE_POST_MESSAGE_NAME is a constant string
    if (message.startsWith(CLOSE_POST_MESSAGE_NAME)) {
        finish();
    }
}
```

Listing 4.8: Source Code of the Message Handler in Google Print

The above implementation of $W \rightarrow N$ suffers from $V_4$, since the source origin is lost. As a result, DoS may be caused, considering the following situations: 1) based on our URL crawler, the web page loaded in WebView contains an HTTP link, which may be leveraged to inject malicious code; 2) adversaries can leverage hybrid postMessage to send a special message to the native frame to stop the service. If the content of the sent message is equal to the value of the variable “CLOSE_POST_MESSAGE_NAME”, DoS may be caused.

In addition, the message handler of the main frame is also a message proxy. However, CSV exists, which indicates that the scenario about the blended attacks on OSV and CSV is feasible (Figure 4.4).

4.5 The Mitigation Solution: OSV-Free APIs

4.5.1 Goals

Motivated by our study result, we aim to design safe hybrid postMessage APIs. The new APIs should achieve the following goals:

- **Meeting the development requirements**: The new APIs should provide both $N \rightarrow W$ and $W \rightarrow N$ functionalities.
- **Secure**: The APIs should not be affected by OSV.

- **Fast**: The APIs should only introduce low overhead.

- **Easy to use**: The APIs should be easily applied and integrated.

- **Generic**: The APIs should be resilient to the notorious Android fragmentation problem, and support as many devices as possible.

### 4.5.2 Overview

Guided by the above goals, we design the OSV-Free APIs. To avoid potential vulnerabilities, such as $V_2$, we explicitly define the origin of the native frame as “nativeframe”. To the best of our knowledge, the origin is meaningful and unique. Please note that the origin is configurable. If an error is found in the origin, the origin can be changed by developers or updated by users.

Similar to existing hybrid postMessage implementations (Section 4.4.3), we also only allow the main web frame to communicate with the native frame. Moreover, to avoid the weakness of existing security solutions, the APIs offer fine-grained origin information and rich hints for building the whole picture of the message delivery, which is helpful to let developers be aware of the blended attacks on OSV and CSV.

As a result, we propose three new hybrid postMessage APIs, called OSV-Free, to allow the secure, fast and generic messaging between the native frame and the main web frame. The APIs are listed in Table 4.3, and more design details are discussed as follows.

In the native frame, the new API `postMessageToMainFrame()` is proposed to allow the native frame to send messages to the main web frame. Since the API can specify the target origin and ensure only the target origin can receive messages, the sub-type vulnerability $V_1$ is eliminated. Correspondingly, in the main web frame, the message handlers can receive messages from the native frame as normal. Since the meaningful and unique source origin “nativeframe” is provided, $V_2$ is also eliminated.

In the main web frame, the new JavaScript method `postMessageToNativeFrame()` is created. Since the native frame is the sole destination, the target origin is already implied in the API itself, and thus $V_3$ is eliminated. In the native frame, to receive messages from the main web frame, a callback function is registered in advance through the API `receiveMessageFromMainFrame()`.
Then, when a message arrives, the callback function is called to handle it with multiple level origin information, so that it can conduct the fine-grained validation. Therefore, $V_4$ is also eliminated.

```java
public class Callback {
    public void onMessage(
        String frameOrigin,
        String scriptOrigin,
        boolean isProxyInvolved,
        String data);
}
```

Listing 4.9: The Prototype of onMessage

Listing 4.9 shows the prototype of the native callback function "onMessage". When a message is received by the callback function, three levels of origin information is provided so that the callback function can perform validation in a fine-grained way, and also obtain hints about the whole picture of the message delivery process. More specifically, the first provided origin “frameOrigin” indicates the origin of main web frame; the second origin “scriptOrigin” provides the origin of the embedded script, where the JavaScript method that sends the message is located; the third variable flag “isProxyInvolved” indicates whether the main web frame is forwarding a message as a proxy. If the flag is true, the scenario similar to what is shown in Figure 4.4 is faced. Hence, developers should carefully handle this situation.

Furthermore, OSV-Free also brings benefits to existing defense solutions for CSV ("$D_1$") and defense solutions for WebView ("$D_2$"). More specifically, OSV-Free makes $D_1$ effective again, since it provides required source origins. OSV-Free also makes up the deficiency of $D_2$ by providing multiple level origin information. Thus, $D_2$ can also offer fine-grained security enforcement and also be aware of the blended attacks on CSV and OSV.

### 4.5.3 Design and Implementation

The key observation behind OSV-Free is that in Android 5+, the declaration and implementation of WebView’s interfaces are separated. The implementation is placed in a standalone library, which is self-managed and self-updated. Hence, we mainly implement OSV-Free by instrumenting the above library, which brings benefits of easy upgrade and minimal modification on the Android source code.

In Android, users can select a browser provider as the library. Currently, Chromium [41] is the default provider. Roughly, Chromium consists of three modules: 1) content, which links Android WebView with the render module together; 2) render, which is responsible to handle rendering
tasks and interact with the JavaScript engine V8; 3) V8, which is a open-source JavaScript engine developed by Google.

OSV-Free’s design is shown in Figure 4.8. OSV-Free mainly consists of two parts: \textit{OSV-Free WebView} and \textit{Customized Chromium Provider}. OSV-Free WebView is a WebView wrapper that declares the native APIs \textit{postMessageToMainFrame()} and \textit{receiveMessageFromMainFrame()}, while Customized Chromium Provider provides the essential implementations of the above two native APIs. For the remaining JavaScript method \textit{postMessageToNativeFrame()}, Customized Chromium Provider can automatically enable it in the main web frame, when a callback function is registered through the native API \textit{receiveMessageFromMainFrame()}. Please note that \textit{OSV-Free WebView} should be integrated into vulnerable apps to replace the original WebView.

To implement OSV-Free, Chromium’s content and render modules are instrumented for each provided API as follows.

\subsection*{4.5.3.1 postMessageToMainFrame()}

This API is implemented by reusing existing methods. When the API is called, the customized content module is started, and then an internal API, called \textit{postMessageToFrame()}, is invoked to handle the whole task of the $N \rightarrow W$ message.

\subsection*{4.5.3.2 receiveMessageFromMainFrame() and postMessageToNativeFrame()}

\textit{receiveMessageFromMainFrame()} is implemented by instrumenting the content and render modules. When the API is called, the content module is entered, where the API’s parameter is cached, parsed, and checked to make sure the format is correct and its internal callback function is not empty. Then, a message is sent to the render module to notify that a callback function is being
registered. After that, the render module reads the context of V8, and binds a pre-defined callback function \( f \) to V8 as “\( \text{postMessageToNativeFrame()} \)”.

In run-time, when \( \text{postMessageToNativeFrame()} \) is called in the main web frame, \( f \) follows. Then, in \( f \), multiple level origin information is collected. The origin of the main web frame “frameOrigin” is obtained by identifying the mainframe object in the frame tree and retrieving the last-loaded URL from the mainframe object. It can be done by calling “\( \text{frame_tree()->GetMainFrame()->last_committed_url().GetOrigin().spec()} \)”.

The origin of the nested script “ScriptOrigin” can be retrieved from the last node of the frame stack (i.e., \( \text{v8::StackTrace::CurrentStackTrace()} \)). The flag “isProxyInvolved” is configured by checking if a message handler is called, which is done by analyzing the above frame stack. Currently, only the global message handler “onmessage” is supported. We leave supporting other message handlers as our future work.

Later, the render module packs all above origin information together with the message content and sends them to the content module. Finally, developers’ callback function “Callback.onMessage()” is called with multiple level origin information and the message content.

### 4.5.4 Evaluation

In this section, we present our evaluation result of OSV-Free on its performance, effectiveness, and compatibility. In the end, we also demonstrate that OSV-Free is easy to use.

#### 4.5.4.1 Performance

To evaluate OSV-Free’s performance, we develop a simple app to call the OSV-Free APIs. We found that OSV-Free was fast, and only used ~2 milliseconds. The details are shown in Table 4.4.

More specifically, we record the starting and ending time of the API execution, and then compute the time difference as the cost. However, we found it was challenging to record the time in two different platforms. To mitigate the problem, we select the method “Date.getTime()”, which is available in both web and native platforms, and also record the time using the same standard. The method returns the milliseconds since midnight 01 January 1970 UTC.

#### 4.5.4.2 Effectiveness

To check OSV-Free’s effectiveness, we use OSV-Free to patch two vulnerable frameworks: the Facebook React Native framework and the Google Print lib. We found that the vulnerabilities could
Table 4.4: The Performance of OSV-Free APIs

<table>
<thead>
<tr>
<th>Target Item</th>
<th>APIs</th>
<th>Average Cost Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The official API (N→W)</td>
<td>postWebMessage()</td>
<td>2.63</td>
</tr>
<tr>
<td>OSV-Free N→W</td>
<td>postMessageToMainFrame()</td>
<td>2.23</td>
</tr>
<tr>
<td>OSV-Free W→N</td>
<td>postMessageToNativeFrame → receiveMessageFromMainFrame()</td>
<td>2.08</td>
</tr>
</tbody>
</table>

be eliminated. In N→W, only the specified target origin can receive the message. When a message is received, its source origin is the native frame’s origin. In W→N, the target origin is implied in the function postMessageToNativeFrame(), while the source origin of the received message provides rich and correct origins.

4.5.4.3 Compatibility

To confirm OSV-Free’s compatibility, we installed and successfully verified OSV-Free APIs in several popular Android versions (5.0+). These tested versions collectively occupy ~80% distribution of the Android market [23].

4.5.4.4 Case Study: Patching the Facebook React Native Framework

To demonstrate OSV-Free is easy to use, we apply OSV-Free to patch the Facebook React Native framework (version 46). We found only a few minutes were used in the process. Our patching code is mainly located in the class ReactWebViewManager. More details are shown as follows.

First, we import the OSV-Free WebView class into the React Native framework. To make it effective, we make the framework’s own customized WebView (i.e., ReactWebView) inherit OSV-Free WebView.

Then, the communication “W→N” is enhanced. Initially, it is implemented based on a JavaScript Bridge, which is enabled by calling two Java methods setMessagingEnabled() and linkBridge(). Instead, in its enhanced implementation, our API postMessageToNativeFrame() is used. To enable postMessageToNativeFrame(), in the above two Java methods, the Java method receiveMessageFromMainFrame() is called instead. Please note that a callback function is pre-defined as the parameter of receiveMessageFromMainFrame() to receive messages from web code. Once a message is received, the received message content and multiple-level source origin information are sent to the JavaScript engine JavaScriptCore (by calling onMessage()), and finally forwarded to
developers’ JavaScript code.

Lastly, the communication “N→W” is also improved. It is done by instrumenting the native method `receiveCommand()`. When a command “COMMAND_POST_MESSAGE” is received for sending a message from the native frame to the main web frame, `postMessageToMainFrame()` is used instead.

4.6 Discussion

4.6.1 OSV-Hunter’s goal

Although some hybrid postMessage APIs are implemented based on JavaScript Bridge, OSV-Hunter is not designed to analyze JavaScript Bridge. Instead, it is used to vet hybrid postMessage against OSV.

4.6.2 OSV-Hunter’s weakness

As a dynamic test tool, OSV-Hunter may have false negatives. For example, OSV-Hunter uses the random test tool “Monkey” to trigger WebView. However, some apps’ WebView can only be shown when preconditions are satisfied. For example, users must finish login, or a pdf file must exist in local storage in advance. To mitigate the problem, we assume all the preconditions are satisfied before our test.

4.6.3 Other ways to defend against $V_4$

Developers may retrieve the origin of the main frame through other ways, such as the native API `WebView.getUrl()`, which provide the URL for the current page. However, the API may fail and return NULL [42]. Developers may also maintain the status of current URL using event handlers [42]. However, this approach may also fail, since event handlers may not be successfully triggered [43].

4.7 Summary

In this chapter, we study hybrid postMessage. We show in the context of WebView, regular postMessage does not meet the requirements. Thus, mobile app developers extend it as hybrid postMessage, which allows cross-platform communication. However, this extended postMessage cannot follow the security model defined for regular postMessage, which causes serious security issues. We develop novel detection and defense solutions to detect and defend against the security
issues.
5. STUDYING AND PROTECTING IFRAMES/POPUPS*

5.1 Overview

5.1.1 Motivation

In the web platform, iframes/popups are frequently used, but also often the root cause of several critical security issues (e.g., frame hijacking [3] and clickjacking [4, 5]). In past years, in regular browsers, these security issues have been well studied, and a variety of mature iframe/popup protection solutions (e.g., Same Origin Policy (SOP) [6], HTML5 iframe sandbox [7], and navigation policies [3]) have been deployed.

5.1.2 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 serious security concerns. For example, WebView enables several programming APIs, such as Settings (Figure 1.1), to help developers customize iframe/popup behaviors. When these APIs are used, it is unclear whether existing iframe/popup protection solutions are still effective.

Furthermore, WebView UI is designed in a simple style (Figure 5.1) 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 bars, 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, serious consequences may be caused (e.g., phishing attacks).

Consider the scenario shown in Figure 5.2 and 5.3. The Huntington banking app (one million+ downloads) uses WebView to help users reset passwords (Figure 5.2-a,b). Inside WebView, the main frame contains an iframe for isolatedly loading untrusted third-party tracking content (Figure 5.3).

Table 5.1: A Summary of Differential Context Vulnerabilities (DCV)

<table>
<thead>
<tr>
<th>Critical Features &amp; Behaviors</th>
<th>Different Contexts</th>
<th>Attacks</th>
<th>Explanations</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Browsers</td>
<td>WebView</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main-Frame Creation</td>
<td>Address Bar</td>
<td>Java APIs</td>
<td>Origin Hiding Attack</td>
<td>Special common origins (e.g., null) Of Main-Frame</td>
</tr>
<tr>
<td>Management of new popups</td>
<td>Table Bar</td>
<td>Android Frameworks</td>
<td>WUI overlap attack</td>
<td>No protection on the WUI rendering sequence</td>
</tr>
<tr>
<td>Main-Frame Navigation</td>
<td>Address Bar</td>
<td>Java APIs</td>
<td>Traditional navigation based attack</td>
<td>Harmful conflict between WebView Customizations and web APIs</td>
</tr>
</tbody>
</table>

However, if the untrusted web content inside the iframe obtains the ability of stealthily redirecting the main frame to a fake website (Figure 5.2-c), serious security risks are posed. For example, users’ personal and bank account information may be stolen, and further financial losses may also be caused.

5.1.3 Differential Context Vulnerabilities (DCV)

Motivated by above security concerns, we conduct the first study on the security of iframe popup behaviors in the context of Android WebView. Please note that 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., settings APIs in WebView), and security policies (e.g., SOP and navigation policies).

As a consequence, our study uncovers a novel class of vulnerabilities and design flaws in WebView. To demonstrate the security implications and impacts of the vulnerabilities, we develop several corresponding concrete attacks. More specifically, we find that these vulnerabilities are rooted in the inconsistencies between differential contexts of regular browsers and WebView. The inconsistencies are summarized in Table 5.1. Several critical we features and behaviors (e.g., main-frame creation, popup creation, main-frame navigation) are involved. They are harmless or even safe in the context of regular browsers, but become risky and dangerous in the context of WebView. We illustrate that by triggering and leveraging these differential features and behaviors,
an untrusted iframe/popup inside WebView can easily obtain unexpected and risky privileges and abilities:

- **Origin-Hiding**: hiding the origin when 1) breaking the integrity of web messaging (i.e., postMessage) [8], which allows the communication between mutually distrusted web frames; and 2) secretly accessing web-mobile bridges [9], which link the web layer and the mobile or native layer (e.g., Java for Android) together (Figure 1.1);

  Existing work has shown that postMessage's message receivers [18, 19] and web-mobile bridges [9, 10, 44] often carry sensitive functionalities, which can be further stealthily accessed by the untrusted iframe/popup. As a result, sensitive information (e.g., GPS location) may be stolen, and important hardware (e.g., camera and microphone) may be unauthorizedly accessed.

- **WebView UI Redressing**: performing phishing attacks by overlapping the foremost benign WUI with an untrusted WUI;

- **(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, but also the solutions proposed for Android UI and WebView. We find that these solutions are ineffective to defend against associated attacks:

  - For origin-hiding attacks, existing defense solutions for postMessage [3, 18–20] and web-mobile bridges [9, 10, 15–17] 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.

  - For WUI redressing attacks, they are similar to Android UI redressing attacks [45–47]. However, the associated Android UI protection solutions (e.g., [48, 49]) 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.

  - 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 5.3. For convenience, especially considering the root reason of this new type of vulnerabilities (i.e., differential contexts), we refer to the vulnerabilities as *Differential Context Vulnerabilities* or DCV, and the associated attacks as DCV attacks.

### 5.1.4 DCV-Hunter & Findings

We next 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 DCV. Then, by applying DCV-Hunter on a number of most popular apps, we show that DCV are prevalent. More specifically, we find 30.4% of 11,341 hybrid apps are potentially vulnerable, including 9,770 potentially vulnerable WebView instances and 18,459 potential vulnerabilities. Up to now, the potentially impacted apps have been downloaded more than 23 billion times in total. Furthermore, our evaluation shows DCV-Hunter is scalable and effective, and has relatively low false positives (~1%).

We also manually verify that many high-profile apps are vulnerable (a list of video demos...
of our attacks can be found online [50]), 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 libs, such as Facebook Mobile Browser and Facebook React Native, are also vulnerable and influence thousands 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 DCV due to several limitations. One major limitation is that their implementation is often error-prone. For example, Facebook Messenger (Figure 5.4, one billion+ downloads) is equipped with the library “Facebook Mobile Browser” to handle URLs contained in messages (e.g., SMS). The browser lib implements its own address bar (Figure 5.4-b) to reflect the change of web content (Figure 5.4-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 5.4-d). Due to the inclusion of the vulnerable lib, many high-profile apps are impacted, such as Facebook and Instagram. In addition to the vulnerable lib, we find this design flaw exists in many other popular apps that are not equipped with that lib, 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.
5.1.5 DCV Mitigation

DCV are not caused by programming mistakes. Instead, they are rooted in the inconsistencies between regular browsers and WebView in term of the UI and programming features, which make several critical and frequently used web features and behaviors risky. It is difficult and impossible for developers to eliminate the DCV security issues, especially considering the limitations of WebView (Section 5.3.6). To mitigate DCV, 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 lib, and provide a list of third-party 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.

5.2 Threat Model

In this dissertation, we mainly focus on the hybrid app whose WebView contains an untrusted sub-frame. In our threat model, we assume its native 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 [34]. A possible scenario example is that 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 arbitrary 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 [51, 52]. 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 [53, 54] and tracking [55–59] services often causes serious security concerns. More than that, as figured out by existing work [60, 61], 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., [51]), 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 in Facebook to lure users to access them. When these websites are opened in the corresponding vulnerable apps (e.g., Facebook or Facebook Messenger), the domains controlled by attackers are accessed. Attackers obtain chances to inject malicious code and launch attacks.

Furthermore, as discussed in Section 2.4, considering the security of the popup behavior is one of our research objectives, we also assume the popup-creation ability of an iframe/popup is enabled in its sandbox attribute.
5.3 Differential Context Vulnerabilities

In this section, we mainly focus on DCV, 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 DCV.

5.3.1 Study Overview

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

- **The application of common origins**: WebView content initialization APIs may create the main frame with common origins, such as “file://” and “null”. For example, the invocation
  
  ```java
  WebView.loadurl('file:///android_asset/index.html')
  ```
  
can load a local file with the origin “file://”, while WebView.loadData() and WebView.loadDataWithURL() 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 2.4). 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...
untrusted sub-frames can still leverage the “null” origin to make several nefarious actions (Section 5.3.2).

- **Concise WebView UI design**: WebView’s UI design causes security risks that untrusted iframes/pop-ups 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 manages multiple WUIs, and 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 5.3.6). As a result, the crucial ability of manipulating the WUI rendering order is exposed (Section 5.3.3.1). 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 5.3.3.2).

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 2.4), 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 5.3.4.1).

Note that the above navigation can be disabled by iframe sandbox (Section 2.4). 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 5.5.2).

- **WebView programming features**: 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 5.3.4.2).

### 5.3.2 Origin Hiding Attacks

As introduced in Section 5.3.1, 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 5.3.2.1) and stealthily accessing web-mobile bridges (Section 5.3.2.2).

#### 5.3.2.1 Attacking Web Messaging

Figure 5.5 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 $F_{\text{nested}}$ with the “null” origin, and let $F_{\text{nested}}$ send a fake message to the victim sub-frame, the victim sub-frame may be fooled.

As shown in Listing 5.1, 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.

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

Listing 5.1: 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 [3, 18–20], because it is challenging for them to distinguish between the main frame and $F_{nested}$.

### 5.3.2.2 Accessing Web-Mobile Bridges

As shown in Figure 5.6, the security risks are also posed that untrusted iframes/popups can also secretly access web-mobile bridges by leveraging the “null” origin (Listing 5.2), 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” [9] and “Draco” [10] are enforced respectively. NoFrak extends SOP to the native layer of a third-party development framework, while Draco implements the access control in WebView. In the first POC app, we use the newest version of the popular third-party framework “Apache Cordova” and instrument the plugin manager to implement NoFrak. In the second POC app, we implement Draco’s prototype system which is implemented in the WebView lib [10]. In both POC apps, we find that untrusted accesses by DCV attacks on web-mobile bridges, especially JavaScript bridges, cannot be prevented.

```javascript
// 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,...<script>alert(\'I am the main frame\')...');
document.body.appendChild(ifrm);
```

Listing 5.2: Accessing the Event Handler onJsAlert() in the Untrusted Iframe/Popup
5.3.3 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 5.7-a), and the WUI closure attack (Figure 5.7-b). We next describe them in detail.

5.3.3.1 WebView UI Overlap Attack

Listing 5.3 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 5.3.6).

```java
public boolean onCreateWindow(WebView view, boolean gesture) {
    // Creating a new WebView UI
    WebView myNewWebView = new WebView(getActivity());
    // Initializing the new WebView UI
    view.addView(myNewWebView);
    // Providing the new WebView UI to Android
    ...}
```

Listing 5.3: Vulnerable onCreateWindow()

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 5.4, the overlap attack can be easily set up in practice.

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

Listing 5.4: Exploit Code of the WUI Overlap Attack and the Privileged Navigation Attack (Table 5.1)

We note that the key API name “addView” also appears in existing work on Android UI redressing attacks such as [47]. 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 dissertation, “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.

### 5.3.3.2 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 5.3.1). As shown in Figure 5.7-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 5.5) 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 5.6).

```
// 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 5.5: Vulnerable onTouchWindow()
We note that 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.

5.3.4 Main-Frame Navigation Attacks

5.3.4.1 Traditional Navigation Attack

Untrusted iframes/popups can leverage traditional navigation policies (Section 2.4) to launch phishing attacks (e.g., using the code in Listing 5.7 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.

5.3.4.2 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 5.4. Note that this code is also used for launching the WUI overlap attack. As
shown in Table 5.1 (the fourth column), when popup-creation is disabled (by default), the code may launch the navigation attack. Otherwise, the WUI redressing attack may be available.

5.3.5 Advantages of DCV Attacks

Compared to existing Android attacks (such as Trojan attacks [30]), DCV attacks do not require declaring permissions, or carrying payload. Compared to other WebView-based attacks (e.g., [9, 28, 31, 32]), 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 [62], DCV attacks do not need to control the native code, and craft the placement of multiple WebView components in Activity layout XML.

In addition, DCV can be leveraged to boost other attacks. For example, event-oriented attacks [44] 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 DCV, such as the privileged navigation attack (Section 5.3.4.2).

5.3.6 Root Causes of DCV

DCV is rooted in the inconsistencies between WebView and regular browsers in terms of UI and programming features (5.3.1). 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, 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 on\texttt{CreateWindow()}, other event handlers such as on\texttt{CloseWindow()} face similar problems.

Another event handler \texttt{shouldOverrideUrlLoading(view,request)} 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, \texttt{shouldOverrideUrlLoading()} often has its own implementation problems in origin validation. For example, our empirical study shows some hybrid apps do not even perform any check, and some of them only check the domain of the URL but ignore the scheme (e.g., “HTTP” or “HTTPS”).

5.4 DCV-Hunter

![Figure 5.8: The Overview of DCV-Hunter](image)

There are several tools for analyzing hybrid apps [13, 38, 44], however, it is challenging to directly apply these tools to detect DCV. On the one hand, existing static analysis tools are not designed for the analysis of iframe/popup behavior (e.g., [13, 38]), and they are often coarse-grained (e.g. [2]). 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., [44]) have high false negatives, as it is very difficult to trigger a WebView instance at runtime. For example, as shown in Figure 5.4, 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 5.8, 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 exist. Finally, DCV-Hunter determines if the given app is potentially vulnerable or not.

### 5.4.1 Complete Call Graph Construction

We build DCV-Hunter on top of FlowDroid [63]. The original FlowDroid is not suitable for analyzing WebView related function invocations to generate accurate call graphs because of the missing of type information and semantics related to WebView. To mitigate this issue, we proposed an app-level online-patching approach so that when all native code is translated to intermediate representatives, DCV-Hunter patches all instructions “I” that contain WebView initialization methods (e.g., “findViewById()”) by providing corresponding type information.

### 5.4.2 WebView Context Reconstruction

In this phase, DCV-Hunter first 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 [2]. 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 [64] 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 5.2. 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 5.4, in Facebook Messenger, when a link is clicked, an Intent message that includes the link is sent out to an activity (Android UI) to start WebView and show that link.

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLs</td>
<td>WebView content loading APIs</td>
</tr>
<tr>
<td>Settings</td>
<td>WebView Setting APIs</td>
</tr>
<tr>
<td>Event Handlers</td>
<td>WebView content loading APIs</td>
</tr>
<tr>
<td></td>
<td>WebView Setting config APIs</td>
</tr>
<tr>
<td></td>
<td>WebView content loading APIs</td>
</tr>
<tr>
<td></td>
<td>WebView Setting config APIs</td>
</tr>
<tr>
<td></td>
<td>Event handler registration APIs</td>
</tr>
</tbody>
</table>

**Table 5.2: Source and Sink APIs**

5.4.3 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()” through path constraint analysis to determine whether extracted URLs are approved.

5.4.3.1 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 2.4), 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” 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., [65]). 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 [66] to automatically collect the webpages (the mobile version) that can be navigated to from the URL within three depth levels. For each collected webpage, its sub-frame is checked based on our threat model (Section 5.2).

  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.

### 5.4.3.2 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 [67], when untrusted iframes/popups are opened or created, the event handler is triggered, and should return false (Note that returning true is usually used for denying the link or other purposes [44]).

Hence, our solution is that 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 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) $\text{parameter}.\text{scheme} == \text{"HTTP"}$
or 2) $\text{extracted}\_\text{URL}.\text{domain} == \text{parameter}.\text{domain}$.

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 [68]) to solve all constraints. If path constraints can
be satisfied, it indicates that the extracted URL should be approved. Please note that to implement
the solution, we also model several frequently used Java classes (e.g., WebResourceRequest, URL,
and String) to support the related operations.

5.4.4 Vulnerability Analysis

To determine each vulnerability, DCV-Hunter checks its conditions respectively (shown in the
fourth column in Table 5.1).

- **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 corre-
  sponding event handlers onCreateWindow() or onCloseWindow() are vulnerable or not. This is
done by checking the existence of the sensitive APIs listed in Table 5.3. Based on the analysis of
the design flaws of these event handlers (Section 5.3.6), 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.
Table 5.3: APIs for the Analysis of WUI Redressing Problems

<table>
<thead>
<tr>
<th>Attacks</th>
<th>Sensitive APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlap</td>
<td>ViewGroup.addView()</td>
</tr>
<tr>
<td></td>
<td>ViewGroup.RemoveView()</td>
</tr>
<tr>
<td>Closure</td>
<td>WebView.setVisibility()</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

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.

5.5 Security Impact Assessment

To assess DCV’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, 3,448 apps (30.4%) were potentially vulnerable, including 9,770 potentially vulnerable WebView instances and 18,459 potential vulnerabilities (Table 5.4). This indicates DCV widely impact real-world apps.

We evaluated the accuracy of DCV-Hunter by measuring its false positives. We randomly selected 400 apps from all the apps flagged as “potentially vulnerable” by DCV-Hunter, and manually checked them (see more details in Section 5.5.1). 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 5.4.2), some unrelated URLs were accounted, due to the imprecise taint analysis (i.e., overtaint). For the remaining two apps, “URL Approval Analysis” (Section 5.4.3.2) 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 two computing cores and 16GB memory. Our time cost showed that each process needed 74 seconds for each app. Thus, the total average time cost was 0.74 second, which indicated DCV-Hunter was fast and scalable.
Table 5.4: Potential Vulnerability Details

<table>
<thead>
<tr>
<th>Potential Attacks</th>
<th>Impacted WebView</th>
<th>Impacted Apps</th>
<th>Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin-Hiding</td>
<td>324</td>
<td>264</td>
<td>2.3 Billion</td>
</tr>
<tr>
<td>WUI Overlap</td>
<td>1,619</td>
<td>408</td>
<td>6.1 Billion</td>
</tr>
<tr>
<td>WUI Closure</td>
<td>639</td>
<td>95</td>
<td>0.3 Billion</td>
</tr>
<tr>
<td>Traditional Navigation</td>
<td>9,769</td>
<td>3,447</td>
<td>23 Billion</td>
</tr>
<tr>
<td>Privileged Navigation</td>
<td>6,108</td>
<td>2,369</td>
<td>16.5 Billion</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9,770</strong></td>
<td><strong>3,448</strong></td>
<td><strong>23 Billion</strong></td>
</tr>
</tbody>
</table>

5.5.1 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 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 (e.g., hardly locating target iframes/popups, and easily causing HTML/JavaScript errors). Instead, we leverage Chrome’s USB debug interfaces. 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.

5.5.2 Findings

Based on our assessment results, we get several interesting findings:

- Many high-profile apps are impacted by DCV: DCV widely exist in hybrid apps. Up to now, the potentially vulnerable apps have been downloaded more than 23 billion times (the fourth column of Table 5.4). Furthermore, these also include many manually verified popular apps (some examples are shown in Table 5.5) such as Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, U.S. Bank.
Table 5.5: Summary of Example (Manually Verified) Vulnerable Apps/Libs
(* 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.)

<table>
<thead>
<tr>
<th>Apps/Libs</th>
<th>Possible Attack Scenarios</th>
<th>Vulnerabilities</th>
<th>Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Main-Frame</td>
<td>Untrusted Sub-frame</td>
<td>OH</td>
</tr>
<tr>
<td>Facebook</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Instagram</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Facebook Messenger</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kakao Talk</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Google News</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Skype</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WeChat</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Yelp</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kayak</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Uber</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ESPN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>McDonald’s</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Samsung Mobile Print</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lastpass</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dashlane</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1password</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>U.S. bank</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Huntingdon bank</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chime mobile bank</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Facebook Mobile Browser Lib</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Facebook React Native Lib</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Almost all categories of apps are affected**: Figure 5.9 shows the related distribution data. The green 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 DCV are common.

We observe that some categories are more subject to DCV attacks than others, such as news, weather, dating, and events. 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 rendering strategy improves user experience. However, as described in Section 5.3.6, 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 5.4, 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.

5.5.3 Case Studies

We have successfully manually launched DCV attacks in many popular apps (some examples are shown in Table 5.5). Readers can find also several video demos at [50] (the website is anonymized). In this section, we mainly present two example apps (Skype and Kayak) for case studies.

5.5.3.1 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 5.10. 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 5.10-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 5.10-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 5.10-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 navigation event occurs. Hence, when the phishing attack occurs, the victim user may hardly be aware of it.

5.5.3.2 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 5.11, 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 5.11-a), such as the AA flight, and then clicks the "View" button to get more details (Figure 5.11-b).

Next, a customized WebView instance is triggered to show more flight details from “aa.com” (Figure 5.11-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 5.11-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.

5.5.4 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, 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.

- **Implementation Errors**: Some apps/libs, such as "Facebook Mobile Browser", use very small fonts to show origins (Figure 5.4). This mitigates the above length limitation problem. As Figure 5.4-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 the letter “O” → 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 5.4-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 lib, 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 [50]). This indicates the blended attack is stealthy. In practice, the blended attack can be easily launched by using the code shown in Listing 5.8.

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

Listing 5.8: Exploit Code of Blended Attacks

### 5.6 Vulnerability Mitigation

#### 5.6.1 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 5.3.6). 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 first-party URL list in a configuration file inside their apps (located in the app folder “assets”), the untrusted requests can be denied at default.

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 also 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 5.5.4). 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 origins.

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.

5.6.2 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 lib 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 [69]).

5.7 Discussion

5.7.1 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 good complementary to our work, and we leave this as our future work.

5.7.2 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.

5.8 Summary

In this chapter, we study iframe/popup behaviors in the context of WebView. Different from regular browsers, WebView enables unique UI and programming features that can heavily impact iframe/popup behaviors and break the integrity of existing defense solutions. We demonstrate remote attackers can utilize (trigger) iframe/popup related behaviors to obtain unexpected and dangerous privileges. To detect and mitigate the security issues, we present novel vulnerability detection and mitigation solutions.
6. RELATED WORK

6.1 Android WebView Security


Several static analysis based approaches [13, 38] were proposed to vet hybrid apps. For example, Chin et al. [26] statically analyzed WebView vulnerabilities that result in illegal authorization and file-based attacks. Yang et al. [13] and Hassanshahi et al. [38] proposed static analysis tools to vet hybrid apps armed with web-mobile bridges. However, they were limited to analyze all the vulnerabilities discussed in this dissertation.

6.2 WebView Related Defenses

Several defense approaches, such as NoFrak [9], MobileIFC [15], and Draco [10], are proposed to extend SOP to local resources, or provide access control on event handlers in the native layer. Other defense approaches, such as WIREframe [16] and HybridGuard [17], provided policy enforcement in WebView to protect app-web bridges. However, all of them were not suitable to prevent the attacks proposed in this dissertation. For example, there are difficulties in applying existing approaches to prevent the EOE attacks. First, Draco requires the root permission to replace WebView’s internal native library, and MobileIFC and NoFrak also require the recompilation of hybrid apps with their own customized hybrid frameworks. Second, they are implemented by instrumenting WebView or third-party hybrid frameworks. Hence, they may have to keep doing extra more work in porting their systems into newest versions. Third, the defense level totally depends on how well the security policies are written by developers. Finally, they performed access control based on the web frame’s origin information.
In addition, many solutions [48, 49] are also designed to mitigate the Android UI deception problems [45–47]. However, as discussed in Section 5.3.3, they cannot monitor the state change of WebView UI, and circumscribed to prevent WUI redressing attacks.

6.3 Event Handler Security

Luo et al. [28] discussed that event handlers may be used by malware to hijack and sniff web events. However, compared with EOE, this type of attacks is more difficult to launch, because adversaries have to control the native code in user devices, such as registering their own native event handlers in WebView. Chen et al. [72] and Mutchler et al. [2] discovered the event handler feature may cause sensitive data leakage (such as the authentication URL) in Oauth. Georgiev et al. [9] and Tuncay et al. [10] discussed the possibilities that adversaries may leverage the event handler feature to access native code. In contrast, we systematically study all types of feasible web event oriented attacks, including the attacks that are carried out by leveraging both one single web event and stitching multiple web events together to influence the program state.

Compared with existing attacks on Webview, EOE is more feasible and practical. Chin et al. [26] analyzed Webview vulnerabilities that result in excess authorization and file-based cross-zone scripting attacks. Wu et al. [70] discussed file leakage problems caused by file:// and content:// schemes in webview. However, these two kinds of attacks are limited in the Android new versions, which provide better protections on directly accessing local files.

6.4 Symbolic Execution

In past years, symbolic execution has made big progress. Several static approaches (such as Intellidroid [73] and TriggerScope [74]) were proposed to vet Android apps using symbolic execution. However, these static approaches may have both higher false positives and negatives in the context faced in this paper. First, static analysis has to address points-to and alias problems. Second, due to the lack of real data, it is challenging to resolve Java Reflection and Intent. Finally, it is difficult to address the array indexing type implicit flows. In real world, this type of implicit flows is frequently used in popular apps and ad libs, such as Google Ads.

Many dynamic approaches were also implemented based on symbolic execution. For example, DART [75] and CUTE [76] applied concolic execution to automatically test software. EXE [77] and KLEE [78] used symbolic execution to find bugs. IntScope [79] employed symbolic execution
to detect integer overflow problems. SAGE [80] was designed for Windows to apply symbolic execution to vet the operating system. S2E [81] proposed the selective symbolic execution to improve the performance. Driller [82] used selective symbolic execution to guide fuzzing, and the result showed the combination was very effective. Existing dynamic approaches may have low false positives. However, it is challenging for them to generate the event sequences required for triggering a found vulnerability.

Several symbolic execution based approaches were also designed to handle implicit flows. For instance, DTA++ [83] used symbolic execution to solve control flow problem (i.e., implicit flows), while Spandex [84] implemented symbolic execution in Android to vet apps about password usage. However, these two systems fall short of handling Android specifications (such as Android Intent) and array indexing type implicit flows.

### 6.5 Iframe/Popup Security

In web apps, iframes/popups are often the cause of security issues, such as frame hijacking [3], clickjacking [4], and double-click clickjacking [5]. 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 [4] solution can prevent being framed. In this work, we mainly focus on the exploration of the abilities of untrusted iframes/popups. As shown in Section 5.3, existing solutions are circumscribed to prevent DCV attacks.

### 6.6 postMessage Security

In past years, several detection and defense solutions for regular postMessage were proposed. However, all of them are incompetent to detect or defend against OSV. Barth et al. [3] conducted a systematic study of the frame isolation and communication, and enhanced postMessage. However, it could not prevent postMessage from being misused, and also did not support hybrid postMessage. Saxena et al. [18] highlighted the client-side validation vulnerability (CSV) in postMessage and proposed the detection tool “FLAX”. Weissbacher et al. [20] applied the dynamic invariant detection technique in defending against CSV. Son et al. [19] conducted a systematic study of CSV on a large number of popular websites, and also proposed novel defense solutions to defend against CSV. Guan et al. discovered DangerNeighbor attacks on postMessage, and designed a deployable defense
solution. However, they were only available to vet or protect the message receivers of $N \rightarrow W$, and could not eliminate OSV by making up the lost origins. Furthermore, since the source origin is not always provided due to $V_2$, their effectiveness may be impacted.
7. LESSONS LEARNED AND BEST WEBVIEW INTEGRITY PRACTICE

As discussed in Chapter 6, WebView security attracts more and more attention. They are already many detection and defense solutions proposed. However, unfortunately, it is still challenging for them to discover and mitigate the security issues we present in this dissertation. There are several reasons:

- **Many popular web features and behaviors are not considered and covered**: Today, the web platform has made significant progresses. A large number of web features and behaviors are available. However, many of them are complicated and sophisticated, which introduce challenges to build a complete protection solution. In this dissertation, we aim to explore sensitive web behaviors, which bring benefits to current solutions. Based on our study results, existing solutions can be enhanced.

- **Event handlers would play more important roles in the protection of WebView**: As presented in this dissertation (e.g., Section 5.3.6), many web features and behaviors can be controlled and limited by the WebView event handler system. Although the event handler system has made great progresses, there are still several weaknesses. In particular, many event handlers failed to provide more detailed information for developers, such that developers can do appropriate validation to prevent malicious behaviors. The expected information includes
  
  - Who is triggering the web event? The information may include the origin and the frame information (indicating if the main frame is making the action) of the actor.
  
  - What action current event is actually making? For example, when a new popup is created, the url being loaded by the new popup (being created) should be provided to the event handler “onCreateWindow()”. Hence, developers can validate the url to ensure only trusted web content can be popped up.

Moreover, based on our study, we find when WebView is integrated, there are several ways for developers to reduce the security risks:
• **Reducing or narrowing down the attack surface**: To launch attacks, remote attackers usually need to inject untrusted code into WebView through web or network attacks. If the code injection surface is eliminated, it is hard for remote attackers to continue attacks. Hence, it is better for developers to carefully control the content loaded in WebView. This can be mainly done through two event handlers:

  - `shouldOverrideUrlLoading()`: This is helpful to control the content loaded in sub-frames.
  - `shouldInterceptRequest()`: This is useful to limit the loading of third-party resources.

• **Doing validation before providing sensitive functionalities**: Some of event handlers provide rich origin information. Developers should well utilize this information and ensure only trusted web code can access sensitive functionalities.

• **Limiting capabilities of sub-frames**: As demonstrated in Chapter 5, the iframe sandbox mechanism is an effective defense solution to limit capabilities of sub-frames. When third-party web content is inevitably involved, the corresponding iframe sandbox should be assigned to disable critical capabilities, such as popup-creation and main-frame navigation.

• **Providing necessary UI elements**: The inconsistencies between regular browsers and WebView cause serious security issues. It is still benefited by mitigating the inconsistencies, such as providing address bar. However, the mistakes discovered in Chapter 5.5.4 should be avoided.

• **Using WebView alternatives**: WebView provides rich UI and programming features. However, for many developers, these features are not required. In this case, the WebView alternatives can applied. These alternatives mainly include

  - **Regular browsers**: Developers can select to open trusted content in WebView, and open untrusted content in regular browsers. This solution can be implemented by leveraging the event handler “`shouldOverrideUrlLoading()`”.

  - **Custom tab**: This is powered by Chrome/Chromium, and safer than WebView.

• **Globally disabling JavaScript**: Many attacks are launched by JavaScript code. When JavaScript is not needed, it is better to globally disable it, which can prevent many attacks. This can be done by using WebView setting APIs.
Nowadays, mobile app developers are enjoying the benefits of the amalgamation of web and mobile techniques. This is achieved through the application of WebView, which can be easily integrated to show web pages and run JavaScript code inside mobile apps. In past few years, WebView security has made significant processes. However, the abilities of untrusted web code in WebView are not well studied. In this dissertation, we fill the gap by conducting the systematic security study on three web behaviors, and several serious security issues:

- **Event-Oriented Exploits (EOE):** We explore the possibilities of involving the event handler features in potential real-world attacks. We further thoroughly study all web events, native event handlers and their triggering constraints. Based on our findings, we present EOEDroid, a novel system that can automatically detect and verify EOE vulnerabilities by generating exploit code. We evaluate EOEDroid using a large number of apps and find several critical vulnerabilities.

- **Differential Context Vulnerabilities (DCV):** 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. We conduct the systematic study of iframe/popup behaviors, and identify several fundamental design flaws and vulnerabilities, named differential context vulnerabilities (DCV). We find that by exploiting DCV, an untrusted iframe/popup becomes very dangerous in Android WebView. We have designed a novel detection technique, DCV-Hunter, to assess DCV security impacts on real-world apps. Our measurement on a large number of popular apps shows that DCV are prevalent. We have also presented a multi-level protection solution to mitigate DCV, which is shown to be scalable and effective.

- **Origin Stripping Vulnerability (OSV):** We conduct the first systematic study on hybrid postMessage in Android apps and identify a new type of vulnerabilities called Origin Stripping Vulnerability (OSV). To measure the prevalence and presence of OSV, we design a lightweight vulnerability detection tool, called OSV-Hunter. Our evaluation on a set of
popular apps demonstrates that OSV is widespread in existing hybrid postMessage implementations. Guided by the evaluation results, we design three safe hybrid postMessage APIs, called OSV-Free, to eliminate potential OSVs in hybrid apps. We show that OSV-Free meets the development requirements: it is secure, fast, and generic.

In the future, we plan to study the following directions:

- Implementing a safer WebView. WebView enables several great unique UI and programming features. However, its design flaws make preventing existing and newly found attacks extremely difficult. We plan to implement a new WebView with safer design and implementation (e.g., safer event handler systems).

- Studying more critical web features and behaviors. In this dissertation, we study three interesting web behaviors, which are safe in regular browsers and become dangerous in WebView. Inspired by this direction, there may be more security issues on other web behaviors. Therefore, in the future, we plan to explore more web features and behaviors (e.g., third-party web content), discover more security issues, and better enhance the security of WebView.
REFERENCES


[33] “Calling fork() from jni code.” [https://groups.google.com/forum/#!protect\kern-.1667em\relaxtopic/android-platform/80jr--A-9bU](https://groups.google.com/forum/#!protect\kern-.1667em\relaxtopic/android-platform/80jr--A-9bU).


