TU WIEN Informatics

Security Analysis of WebViews in Cross-Plattform Mobile Frameworks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Paul Florian Sattlegger, BSc
Matrikelnummer 11810278

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr. techn. Martina Lindorfer, BSc
Zweitbetreuung: Senior Scientist Dott. ric. Marco Squarcina
Mitwirkung: Jakob Bleier, BSc BSc MSc

Signiert von: Paul Florian Sattlegger
Datum: 25.09.2023 12:25:35
A TRUST
Dieses Dokument ist digital signiert!
Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.
Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.a-trust.at/pdf

Signiert von: Martina Lindorfer
Datum: 25.09.2023 11:48:45
A TRUST
Dieses Dokument ist digital signiert!
Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.
Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.a-trust.at/pdf

Wien, 24. September 2023

Paul Florian Sattlegger　　Martina Lindorfer

TU WIEN Informatics

Security Analysis of WebViews in Cross-Plattform Mobile Frameworks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Paul Florian Sattlegger, BSc
Registration Number 11810278

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr. techn. Martina Lindorfer, BSc
Second advisor: Senior Scientist Dott. ric. Marco Squarcina
Assistance: Jakob Bleier, BSc BSc MSc

Vienna, September 24, 2023

Paul Florian Sattlegger | Martina Lindorfer

Erklärung zur Verfassung der Arbeit

Paul Florian Sattlegger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. September 2023

Paul Florian Sattlegger

Acknowledgements

First and foremost, I want to thank my supervisors, Professor Martina Lindorfer, Marco Squarcina, and Jakob Bleier, for their support and guidance throughout my thesis. Their expertise, responsiveness, and commitment provided a solid foundation for my research and encouraged me at vital stages of this journey.

I would also like to thank my parents for their constant support throughout my studies. Their encouragement and faith in me have been unwavering, and I owe much of my success to their belief in my abilities.

Abstract

WebViews play a critical role in integrating web content into mobile apps. However, balancing their smooth integration with security has been a challenge. More recently, modern cross-platform mobile frameworks (XMFs) such as React Native and Flutter have emerged as popular tools for streamlining cross-platform development. Despite their growing adoption, the security implications of WebViews in these frameworks remain largely unexplored. This thesis fills this gap by systematically investigating the integration and vulnerabilities of WebViews within XMFs.

Throughout this thesis, we describe five vulnerabilities associated with using WebViews that pose threats ranging from information leakage to phishing attacks. These vulnerabilities include (a) surreptitious access to methods injected via the JavaScript interface, (b) lack of effective navigation restrictions, (c) insecure HTTP authentication, (d) unauthorized navigation within iframes, and (e) insecure defaults. Our analysis of 3,411 apps shows that many XMF-based apps are likely to be affected due to their use of features found to be vulnerable. While determining the exact number of vulnerable apps is challenging due to the novelty of modern XMFs and the corresponding lack of advanced analysis tools, these findings highlight the urgent need for improved security auditing and provide valuable insights for developers, security analysts, and the broader mobile app industry.

Kurzfassung

WebViews spielen eine zentrale Rolle bei der Integration von Webinhalten in mobile Anwendungen. Diese Integration hat bisher zu zahlreichen schwerwiegenden Sicherheitslücken geführt. In jüngster Zeit haben sich moderne cross-platform mobile frameworks (XMFs) wie React Native und Flutter als beliebte Werkzeuge zur Beschleunigung der plattformübergreifenden Entwicklung etabliert. Trotz ihrer zunehmenden Verbreitung sind die Sicherheitsrisiken von WebViews in diesen Frameworks noch weitgehend unerforscht. Die vorliegende Arbeit schließt diese Forschungslücke durch eine systematische Untersuchung der Integration von WebViews in XMFs und den damit verbundenen Sicherheitsrisiken.

In dieser Arbeit werden fünf Schwachstellen beschrieben, die mit der Verwendung von WebViews zusammenhängen und Bedrohungen darstellen, die den unbefugten Zugriff auf Informationen und Phishing-Angriffe ermöglichen. Zu diesen Schwachstellen gehören (a) unerlaubter Zugriff auf Methoden, die über die JavaScript-Schnittstelle bereitgestellt werden, (b) das Ausbleiben von wirksamen Navigationsbeschränkungen, (c) unsichere HTTP-Authentifizierung, (d) unbeschränkte Navigation innerhalb von iframes und (e) unsichere Standardeinstellungen. Unsere Analyse von 3.411 Anwendungen, die auf XMFs basieren, zeigt, dass viele dieser Anwendungen betroffen sind, da sie Funktionen verwenden, die mit diesen Sicherheitslücken in Zusammenhang stehen. Aufgrund der Neuartigkeit moderner XMFs und des entsprechenden Mangels an ausgereiften Analysewerkzeugen ist es schwierig die genaue Anzahl anfälliger Apps zu bestimmen. Dennoch unterstreichen diese Ergebnisse den dringenden Bedarf an verbesserten Sicherheitsprüfungen und liefern wertvolle Erkenntnisse für Entwickler und Sicherheitsanalysten im engeren und für die gesamte Mobile-Apps-Branche im weiteren Sinne.

Contents

CHAPTER 1

Introduction

WebViews are a convenient way to integrate web content into mobile apps seamlessly. They allow developers to render websites and run JavaScript without requiring users to switch apps. To enable deep integration, WebViews can be customized by adding communication interfaces to the underlying app or by using event handler APIs [Luo+11]. Historically, this customization has proven difficult because integrating web content into apps involves carefully balancing two conflicting goals. On the one hand, web content should integrate well with the app. On the other hand, potentially untrusted web content should be isolated from the underlying app to minimize security risks. The problem is further evidenced by the fact that security experts continue to present new attack vectors involving WebViews despite more than a decade of intensive research [GJS14; HMM17; Ras+16; Yan+17; Yan+18; YHG18; YHG19; Zha+22]. The importance of studying the security and privacy aspects of WebViews stems from the severe consequences they pose and their widespread use. The implications of inadequate isolation of WebViews range from (a) surreptitious access to the hardware such as the microphone and camera [Yan+18], (b) leakage of sensitive information, including authentication tokens [GJS14; Yan+18; YHG18], (c) launching of phishing attacks [GJS14; HMM17; YHG18; YHG19], (d) to the installation of malware on smartphones [Zha+22]. In addition, the topic has recently made headlines worldwide after an independent report found that ByteDance and Meta use WebViews to manipulate websites [Kra22]. Even though there is no evidence of harmful intentions, the presence of such practices confirms the importance of a more thorough investigation.

1.1 Aim of the Thesis

Motivated by the security concerns described above, the primary objective of this thesis was to comprehensively investigate the security of WebViews in the context of cross-platform mobile frameworks (XMFs). In essence, XMF is a new term used to describe frameworks that let developers build apps for various mobile platforms with a single codebase. Traditionally, such frameworks relied on web technologies, and apps using them were commonly referred to as *hybrid apps* [Mal+15; MS21; Yan+17; Yan+18]. However, since XMFs are not restricted to web technologies, we use this new term to to cover all such frameworks.

XMFs work by abstracting the intricacies of the underlying platforms away from developers. We investigated the presence of novel vulnerabilities caused by these abstractions and quantified their prevalence. This fresh look at the problem allowed us to examine specific implementation details of these frameworks and find a large number of potentially vulnerable apps, as apps inherit the vulnerabilities of the frameworks they use. Below we describe the specific objectives formulated as research questions:

[**RQ1**] How are WebViews integrated into XMFs? What bridging mechanisms are used by the most popular frameworks?

[**RQ2**] What vulnerabilities arise from the integration of WebViews in XMFs? Do these bridging mechanisms allow attackers to escalate privileges beyond the web context?

[**RQ3**] How many apps are affected, and what are the consequences of the vulnerabilities?

Although the purpose of XMF is to provide a single codebase for all platforms, we decided to focus on the Android implementation of XMFs because the XMFs are already extensive in their terms, and iOS, for example, would be well beyond the scope of this work.

Considering that XMFs account for a good third of mobile apps [Lio22], our research represents a significant contribution towards further safeguarding the entire app ecosystem.

1.2 Methodology

To answer the research questions described in Section 1.1, we divided our thesis into the following three phases:

(a) **Exploring Implementation Approaches of XMFs.** We first identified the most popular XMFs by adoption rate and then performed a *manual analysis* of their documentation and source code. In addition, we created small test apps for each of the XMFs to support the overall understanding. The aim was to get a comprehensive overview of each XMF's implementation approach and to understand if and how native functionality is interwoven with web content.

(b) **Assessing Vulnerabilities in WebView Plugins of XMFs.** Based on the comprehensive overview, we tried to identify fundamental weaknesses in each approach, with attention to the abstractions provided by XMFs concerning WebViews. The developer can customize the WebView component of Android via several classes. We examined the differences in the customizations using our small test apps and the corresponding source code. The differences in the implementation are essentially what the framework provides out of the box and where bugs can be propagated to all apps based on it. For each vulnerability we found, we briefly described the vulnerability along with a threat model and a proof of concept (PoC) for validation.

(c) **Evaluating the Prevalence of Found Vulnerabilities.** To determine the extent to which the vulnerabilities we found are widespread, we compiled a dataset of apps that rely on XMFs and then checked whether and how often certain vulnerable features are used to give an estimate of the number of affected apps. We used *static program analysis* for this, as previous work has already shown that a static approach is necessary because WebViews often require a specific state that is difficult to create (and reproduce) dynamically [YHG19]. In addition, dynamic approaches are difficult to scale, which is particularly problematic when there are a large number of apps to be reviewed.

1.3 Contribution

We summarize the contributions of this thesis:

(a) We conducted the first systematic study of vulnerabilities in *modern* XMFs, namely React Native and Flutter, focusing on vulnerabilities in the WebView component. We found that while modern frameworks simplify the development process, they also increase the risk of attack and pose significant challenges for security analysis due to the lack of tool support resulting from their novelty.

(b) Regarding the security vulnerabilities, we were able to find that: (i) React Native and Flutter both use a known vulnerable pattern, and changes to the Android APIs increased the ease of attacks. (ii) React Native and Flutter's WebView components are less secure, as navigations are not handled by the OS but are managed internally. (iii) React Native implements an insecure HTTP authentication mechanism. (iv) Flutter does not restrict navigation within iframes. (v) React Native and Flutter use less secure defaults.

(c) We semi-automatically analyzed 50,831 apps. We confirmed that 3,411 apps with a total of over 24 billion downloads are based on XMFs, including high-profile apps such as Microsoft Office or Skype. We also found that up to 87.53 % of apps built with XMFs use WebViews, demonstrating the importance of their study. We further identified 415 apps that are potentially affected by one of the vulnerabilities we found.

1.4 Structure of the Thesis

The rest of the thesis is organized in the following way: First, in Chapter 3, we examine how the three most popular XMFs use WebViews. We then analyze how the integration of WebViews introduces vulnerabilities and describe five unique vulnerabilities in Chapter 4. Afterward, we discuss the impact of the vulnerabilities we found by quantifying the apps that use vulnerable features in Chapter 5. Finally, we conclude this thesis in Chapter 6 with a summary and outline of possible future work.

CHAPTER 2

Preliminaries and Related Work

In this chapter, we provide the necessary knowledge to understand the rest of the thesis and review the state-of-the-art research in web security and mobile system security related to vulnerabilities in WebViews and XMFs. In Section 2.1, we briefly explain the basic features of the WebView component. In Section 2.2, we present the tools that we used throughout the thesis. In Section 2.3, we address the studies that have explored the security, or lack thereof, in WebViews. In Section 2.4, we give a comprehensive overview of work concerning the security of XMFs.

2.1 Background on WebView

WebViews allow developers to display web pages within ordinary native apps, i.e., apps written in Java or Kotlin that can take full advantage of the capabilities of the device and its OS. They eliminate the need for users to switch context from the app to a separate browser, providing a more seamless user experience. In addition, the component can be deeply integrated into an app by allowing developers to customize its UI layout and behavior to fit into the native app. This makes WebViews a pragmatic choice, despite drawbacks such as moderate performance and limitations in accessing device-specific features.

Integrating a WebView into an Android app is as simple as instantiating the `WebView` class and calling the `loadUrl` method, specifying the URL we want to load (see Listing 2.1).

A critical aspect of using WebView, especially from a security perspective, is its ability to facilitate bi-directional communication between the native app and the web page loaded within the WebView [Luo+11]. In what follows, we discuss both directions, using small snippets to illustrate the idea.

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.loadUrl("http://www.example.com");
```

Listing 2.1: Loading `http://www.example.com` in a WebView

2.1.1 Invoke Java From JavaScript

Using the `addJavascriptInterface` method, developers can inject a reference to a Java class into the WebView's JavaScript context, allowing the web page to call back to Java.

```
class JsObject {
  @JavascriptInterface
  public String toString() { return "injectedObject"; }
}
myWebView.getSettings().setJavaScriptEnabled(true);
myWebView.addJavascriptInterface(new JsObject(), "injectedObject");
myWebView.loadData("html", "text/html", null);
myWebView.loadUrl("javascript:alert(injectedObject.toString())");
```

Listing 2.2: Invoking Java code from a WebView's JavaScript [And23d]

Listing 2.2 demonstrates the process of injecting an instance of `JsObject` into the context of `myWebView`. The injection is done by calling the `addJavascriptInterface` method, which injects the instance using the supplied name `"injectedObject"`. We then trigger a JavaScript alert dialog that displays the string returned by the `toString` method of `JsObject`, illustrating the successful communication of JavaScript to Java code.

2.1.2 Invoke JavaScript From Java

The standard way to communicate from the native app to the web page is by injecting JavaScript code from Java [Luo+11]. This is achieved either by calling the `loadUrl` method with a `javascript:` prefix or by using the `evaluateJavascript` method.

```
webView.loadUrl("javascript:alert('injected')");
webView.evaluateJavascript("alert('injected')", null);
```

Listing 2.3: Interacting with a WebView from Java code

Listing 2.3 demonstrates the two ways to execute JavaScript in an Android WebView. The `loadUrl` and `evaluateJavascript` methods both trigger a JavaScript alert dialog, illustrating the communication of Java to JavaScript code.

2.2 Background on Tools

This section is intended to give a brief overview of all the tools used throughout the thesis. For further information, we refer to the project websites.

2.2.1 Android Studio

Android Studio is the official integrated development environment (IDE) for Android, maintained by Google [And23a]. Based on IntelliJ IDEA, Android Studio provides a full-featured code editor, a fast and feature-rich emulator, and a powerful debugger, making it a valuable tool not only for developers but also for security researchers.

We used Android Studio extensively throughout our research process. First, we used it to examine the source code of the frameworks we were studying. Subsequently, Android Studio served as our development environment for creating small apps using these frameworks. It was also key in developing, testing, and debugging our PoCs. Finally, Android Studio was the platform of choice for executing and evaluating our attacks in real-world apps.

2.2.2 AndroGuard

AndroGuard is an open-source Python tool designed for reverse engineering and analyzing Android apps [Ant23]. It provides a set of powerful features for extracting information from Android Package (APK) files, such as decompilation, bytecode analysis, and vulnerability detection.

We used AndroGuard's capabilities to automate the scanning of thousands of apps, specifically targeting the detection of certain classes and strings. We then used these findings to compile comprehensive statistics that shed light on the prevalence of these elements within the analyzed apps.

2.2.3 JADX

JADX is an open-source tool used for decompiling Android apps directly into Java source code [Sky23].

We turned to JADX for manual inspection of the app source code. However, it is important to note that while JADX excels at analyzing compiled Java code, its utility diminishes when applied to apps developed using React Native and Flutter. This limitation is because both React Native and Flutter use their own programming languages, making JADX less suitable for these types of apps.

2.2.4 Jupyter Notebooks

Jupyter Notebooks are interactive documents that blend code, text, and visualizations [Jup23]. A key feature of Jupyter Notebooks is the ability to run code cells in real time, making it easy to see immediate data analysis results. Jupyter provided a convenient interface for seamless interaction with Pandas, facilitating rapid iteration of our research results.

2.2.5 Pandas

Pandas is a widely used Python library for data manipulation and analysis [Pan23]. We used Pandas to work effortlessly with the data generated by AndroGuard.

2.3 Vulnerabilities in WebViews

In line with the frequent use of WebViews, there is a large body of work dealing with vulnerabilities involving WebViews.

Zhang et al. [Zha+22] examined so-called *app-in-app ecosystems* for WebView vulnerabilities. App-in-app ecosystems essentially provide a platform for lightweight WebView-based apps, i.e., the host apps act much like a browser (or OS). These ecosystems suffer from poor isolation of individual apps caused by inadequate *identity checks*. As a result, an attacker can maliciously call the host apps' privileged APIs from a vulnerable or unprivileged guest app. Like our work, they examined a specific family of apps for vulnerabilities and discovered novel vulnerabilities. We have adopted their approach but examined XMFs instead.

Yang, Huang, and Gu [YHG19] described two novel attacks against apps that use WebViews. The first attack is a *clickjacking* attack named *WebView UI Redressing Attack*, enabled by WebViews that cover the entire screen and insecure handling of popups. The second attack, referred to as *Main-Frame Navigation Attack*, involves navigating to the top-level frame from an untrusted subframe. Both attacks require a vulnerable app to load an untrusted sub-frame (e.g., through a person-in-the-middle (PITM) attack or by controlling an included subdomain). They then conducted an extensive analysis and found many potentially vulnerable apps, including high-profile apps such as Facebook, Instagram, and Uber. Our work is related in that we found that XMFs allows navigation by default, and developers rarely use the framework's abilities to restrict navigation. Both of these features allow attacks similar to the *WebView UI Redressing Attack* described.

Yang et al. [Yan+18] investigated how custom *postMessage-like* APIs for communication between WebView and native app code can be exploited. Specifically, they highlight the security implications of *origin stripping* implemented in several apps, allowing attackers to access native functions from untrusted WebViews, enabling activities like eavesdropping on the microphone or data manipulation. Unlike Yang et al., we examined not only the postMessage-like APIs provided by XMFs, but also looked at other customizations XMFs makes to the platform's WebView. These include changes to navigation behavior, HTTP authentication, and security-critical defaults. Additionally, we used static analysis instead of dynamic analysis to achieve a higher code coverage [YHG19].

Hu et al. [Hu+18] conducted a comprehensive empirical investigation into the bugs caused by apps using WebViews. They analyzed 124 popular open-source Android apps and identified six distinct categories of bugs. The primary effects of these bugs were performance degradation, UI inconsistencies, and app crashes. In addition, they introduced an automated testing methodology, ωDroid, which they used to uncover similar but previously undetected bugs in an additional 146 app. In contrast to our work, their study centered on general errors in WebView-enabled apps rather than on security issues.

Mutchler et al. [Mut+15] conducted a large-scale study of 998,286 mobile apps to quantify the prevalence of security vulnerabilities related to WebViews. In doing so, they picked up on weaknesses previously studied by Luo et al. [Luo+11] and included them in their analysis pipeline. They found that 28 % of apps contained at least one security vulnerability and listed mitigations that involve changes to the Android APIs to reduce the frequency and severity of these vulnerabilities.

Chin and Wagner [CW14] explored two attacks on the WebView implementation on Android. The first attack allows a web attacker to invoke Android app code. The second attack allows exposing the devices' file system to an attacker. They also built a tool to detect and characterize the prevalence of these vulnerabilities and found 67 vulnerable apps.

Wang et al. [Wan+13] described various attacks on mobile apps based on mobile platform-specific communication channels such as intents, URL schemes, and utility classes for communicating with WebView. All attacks rely on the apps not knowing the origin of a particular action, or in other words, the lack of origin-based protection usually provided by a browser. They found exploits in high-profile apps such as Facebook and Dropbox for each category. Furthermore, they designed and implemented a protection mechanism for mediating the root cause at the operating system level.

Neugschwandtner, Lindorfer, and Platzer [NLP13] explored the actual exploitation of the attack vectors identified by Luo et al. [Luo+11]. They examined two potential threat scenarios: server compromise and traffic compromise, both targeting WebView. They also presented case studies of successfully exploited apps. For example, they identified the *Nature Wallpaper* app, with over 500,000 installs, as vulnerable and were able to gain access to persistent storage and download (and execute) further malicious content. In addition, they conducted a large-scale evaluation of 287,512 Android apps to determine the prevalence of vulnerable WebView apps.

Luo et al. [Luo+11] were the first to thoroughly investigate attacks on the WebView component of Android. They outlined that WebView allows bypassing the sandbox generally provided by a browser and thus provides many opportunities for attacks.

Reference	Scope
[Zha+22]	App-in-app ecosystems
[YHG19]	WebView-enabled apps using frames and popups
[Yan+18]	WebView-enabled apps using postMessage-like API
[YHG18]	WebView-enabled apps using web events
[Hu+18]	Open-source WebView-enabled apps
[Mut+15]	WebView-enabled apps
[CW14]	WebView-enabled apps (using `file://` scheme)
[Wan+13]	Intent, scheme and web accessing utility-enabled apps
[NLP13]	WebView-enabled apps
[Luo+11]	WebView-enabled apps

Table 2.1: Overview of prior research on WebView vulnerabilities

Additionally, they discuss attacks involving code injection into WebViews and attacks utilizing event handlers. Although the paper is now dated, much of the current research is still building on the foundations it laid.

Table 2.1 summarizes the vulnerabilities examined in this section, including the specific references and the scope – namely, the categories of apps or mechanisms under investigation.

2.4 Cross-Platform (Mobile) Frameworks

Despite the increasing popularity of XMFs in mobile development [Lio22], the research on their vulnerabilities has been limited. As a result of papers published as recently as 2022, it has become apparent that there may have been a failure to apply the lessons learned from previously discovered attacks to other related mobile platforms. For example, Zhang et al. [Zha+22] discovered that so-called app-in-app ecosystems are vulnerable to an attack similar to one described four years earlier by Yang et al. [Yan+18]. Furthermore, Beer et al. [Bee+22] has recently shown that modern XMFs, specifically React Native and Unity, are vulnerable to an access control vulnerability. In the following, we summarize work that concerns cross-platform (mobile) frameworks.

Xiao et al. [Xia+22] investigated cross-platform (desktop) apps for arbitrary code execution (ACE) vulnerabilities. They collected ten vulnerabilities from bug bounties, CVE databases, and industry studies. As a result, they identified *custom communication interfaces* between the rendering and system-side contexts as particularly insecure. In addition, they generated restrictive API profiles (similar to AppArmor [Imm23], a Linux kernel security module) as a defensive solution using a combination of static and dynamic analysis. Similar to their work, we are also investigating cross-platform apps, but for mobile phones rather than desktops. The mobile platform is currently dominant and, as such, is a more compelling target for threats than the traditional desktop environment [Sta23].

Reference	Scope	Desktop	Mobile
[Xia+22]	Electron-based Apps	●	-
[Zha+21]	In-App Browsing Interfaces	-	●
[MS21]	Apache Cordova-based Apps	-	●
[GJS14]	PhoneGap	-	●

Table 2.2: Overview of prior research on Cross-Platform (Mobile) Frameworks

However, since Android has significantly more restrictions (e.g., app sandbox), we believe this platform is also more challenging to exploit. Additionally, we also investigated the communication interfaces that the frameworks provide.

Zhang et al. [Zha+21] studied so-called *in-app browsing interfaces*, i.e., the interfaces apps provide when opening (external) web APIs. To this end, they collected and analyzed 25 high-profile mobile apps and performed a systematic analysis throughout interacting with these browsing interfaces. Their results show various deficiencies which make it impossible for the user to detect, e.g., insecure schemes or phishing attacks. In comparison, our work focuses more primarily on technical flaws, i.e., errors in the implementation, than errors in the use of apps with embedded browser instances. Nevertheless, the studies are related because they examine the same topic and aim to find vulnerabilities, albeit with a different methodology and goal.

Mohanty and Sridhar [MS21] examined Apache Cordova-based IoT companion apps for pre-existing security issues (e.g., web vulnerabilities, outdated software development kits (SDKs), insecure WebView usage). To this end, they developed a vulnerability assessment framework that identified nine vulnerabilities in 102 hybrid apps. Our work covers Apache Cordova and the now more popular Flutter and React Native frameworks. In addition, we found new vulnerabilities rather than relying solely on existing ones.

Georgiev, Jana, and Shmatikov [GJS14] demonstrated that hybrid app frameworks, in particular, but not limited to Apache Cordova (formerly PhoneGap), do not correctly compose the same-origin policy (SOP) and local access control policy. As a result, foreign-origin web content embedded in hybrid frameworks can gain access to native device resources. Web content can access native content in several ways: (a) through globally exposed objects via `addJavascriptInterface`, (b) via event handlers for functions, and (c) via page redirection. They call the resulting vulnerabilities *fracking* attacks and additionally propose a defense mechanism called *NoFrak*. In contrast, we did not focus on hybrid app frameworks but on XMFs.

Table 2.2 provides an overview of related work in cross-platform (mobile) frameworks. We indicate the scope, i.e., the type of apps or mechanisms studied, and whether the study focuses on mobile or desktop apps.

CHAPTER 3

Cross-Platform Mobile Frameworks

In this chapter, we explore the domain of XMFs, focusing in particular on their approaches for integrating web content into native apps. In Section 3.1, we provide a concise introduction to XMFs and explain the reasoning behind our selection of frameworks for further exploration. From Section 3.2 to Section 3.4 we then break down the architecture of each of the chosen frameworks and analyze how they approach the task of embedding web content in a native environment. Lastly, in Section 3.5, we present a summary of our findings, comparing the differences and similarities between the investigated frameworks.

3.1 Term Definition and Key Frameworks

XMFs provide a consistent way to build software that runs seamlessly across multiple platforms. They reduce developers' time, resources, and technical differences associated with creating different versions of apps for each platform, such as Android and iOS, by allowing developers to *write once, run anywhere*. While previous work has often used the term *hybrid app* to refer to these kinds of frameworks, this term is not entirely appropriate for describing *modern* XMF frameworks like React Native and Flutter. The term hybrid app has traditionally been used to describe apps that are built using web technologies such as HTML, CSS, and JavaScript, and then encapsulated within a native app container to provide native functionality. However, XMFs may not rely on web technologies at all, representing a broader category of frameworks. Therefore, we choose to use the more inclusive term XMF in this thesis to accurately reflect the range of frameworks under discussion. There are many XMFs available. We focused our investigation on the three most popular and widely used frameworks: Apache Cordova [The23a], React Native [Met23c], and Flutter [Goo23a].

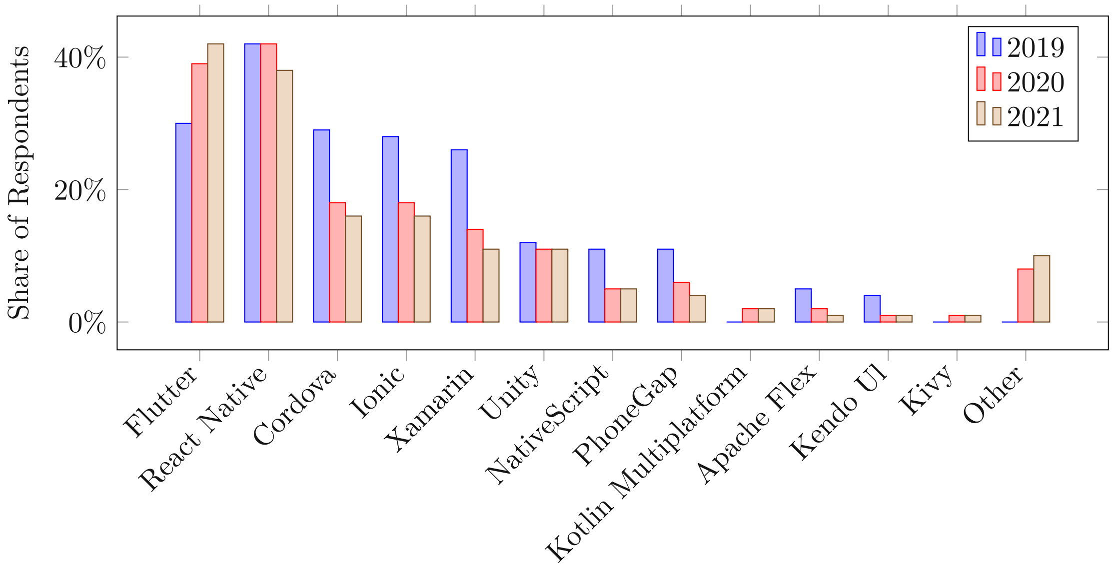


Figure 3.1: XMFs usage by software developers worldwide from 2019 to 2021 [Lio22]

Figure 3.1 illustrates the popularity of XMFs based on a survey conducted in collaboration with JetBrains [Lio22]. The y-axis represents the number of developers using each framework, while the x-axis provides a temporal comparison for the years 2019, 2020, and 2021. The data indicate that Flutter has gained significant popularity in recent years, while React Native has maintained a relatively stable level of popularity. Other frameworks, including Cordova, have seen a decline in popularity.

3.2 Apache Cordova

Apache Cordova is a popular open-source framework for building mobile apps using standard web technologies such as HTML, CSS, and JavaScript. Cordova apps are essentially web apps that are wrapped in a native container, allowing them to access device-specific functionality.

Figure 3.2 provides an architectural view of a Cordova app. A Cordova app consists of mainly three components: App code, plugins, and a WebView. The app code contains the business logic of the app, implemented as an ordinary web page using web technologies. Plugins play a crucial role in Cordova apps by providing access to native functionality such as the camera, file system, or contacts. These plugins act as a thin wrapper for platform-specific code, bridging the app code to the underlying native functions.

A Cordova app utilizes a WebView as its runtime environment. The WebView serves two primary purposes: (a) It provides a bridge between the part of the plugin that is implemented as web code and the part of the plugin that is implemented using platform-specific code, and (b) it handles the rendering and execution of the app code

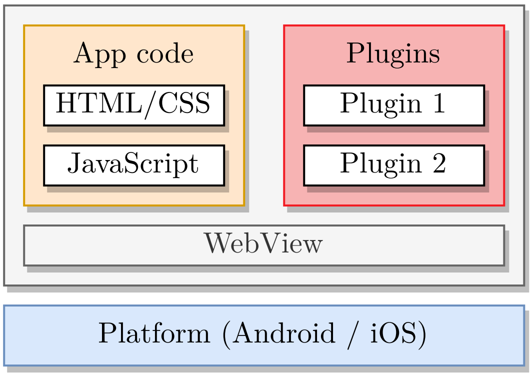


Figure 3.2: Architecture of an Apache Cordova-based app [The23b]

itself.

It remains to be clarified how the part of the plugins *inside* the WebView communicates with the part *outside* the WebView. Given the lack of documentation, we created a small Android app and included the *Battery Status* plugin [The23c] for demonstration. The Battery Status plugin provides an implementation of the Battery Status API [KLK22].

We then examine the functioning using the debugger in Android Studio: The process begins with the web app starting the initialization of the plugin by executing the `exec` method. This method is exposed via `addJavascriptInterface` and is thus callable from JavaScript [And23d]. Following that, the plugin's `execute` method is called, which registers a *broadcast receiver* so that the web app is notified when the battery level changes. Upon battery level change, the Android platform broadcasts an ACTION_BATTERY_CHANGED intent. This intent is received by the plugin, which then notifies the WebView. How the web app receives this event depends on the supported bridging mechanism. Cordova supports four bridging mechanisms namely: (a) POLLING, (b) ONLINE_EVENT, (c) LOAD_URL, (d) EVAL_BRIDGE. With POLLING, the web app polls for pending messages by repeatedly calling a native method exposed via `addJavascriptInterface`. ONLINE_EVENT works just like POLLING but additionally tells JavaScript when to poll for messages. LOAD_URL injects a JavaScript snippet into the WebView by loading a URL prefixed with `javascript:`; EVAL_BRIDGE injects the snippet into the web app via the `evaluateJavascript` method.

3.3 React Native

React Native is another popular open-source framework for building mobile apps. The framework combines the *React framework* with access to native platform features, as the name suggests [Met23c]. While React Native and Apache Cordova share the use of JavaScript for app development, the former uses a platform-independent C++ rendering engine called Fabric and a custom JavaScript engine called Hermes instead of

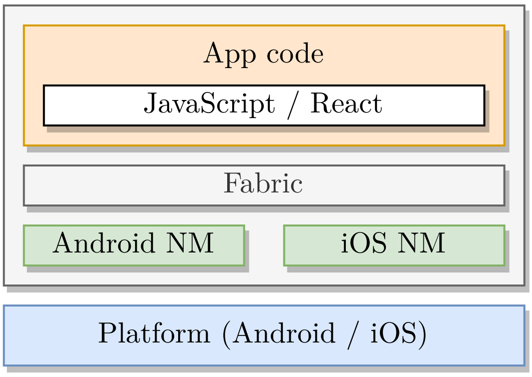


Figure 3.3: Architecture of an React Native-based app [Met23b]

a WebView [Met23b]. At runtime, React Native creates the corresponding platform-specific views and components. Hence, React Native apps look, feel, and perform like any other app because React Native components are based on the same views as the platform [Met23a].

However, React Native also provides the ability to display arbitrary web pages within a WebView. This can be done using the react-native-webview plugin [Rea23b]. To enable deep integration, the plugin allows bi-directional communication between the app code written in React and the web page rendered in the WebView [Rea23a]:

(a) To send messages from the WebView to React Native, the `onMessage` attribute must be set when declaring a `<WebView/>` element. Subsequently, communication can take place from the WebView via the `ReactNativeWebView.postMessage` method with the callback set via the `onMessage` attribute (see Listing 3.1).

(b) To send messages from the WebView to React Native, the `injectedJavaScript` can be used. For this purpose, a `MessageEvent` can be used to dispatch the message, and inside the WebView an `EventListener` is used to receive the message (see Listing 3.2).

3.4 Flutter

Unlike React Native and Apache Cordova, Flutter apps are not written using web technologies, but in the Dart programming language and run in the Dart VM on the target platform [Goo23a].

Figure 3.4 provides an architectural view of a Flutter app [Goo23b]. Starting from the platform, i.e., Android or iOS, Flutter abstracts all the peculiarities of the platform with the help of a platform-specific *embedder*. The embedder then provides a platform-agnostic foundation for the *engine*, where all low-level primitives are implemented. The engine

```
function App(): JSX.Element {
  return (<WebView
      source={{ uri: "https://www.example.com/" }}
      onMessage={(event) => {
        console.log(event.nativeEvent.data);
        // "Hello from JavaScript!"
      }}/>)
}
```

(a) App code in React Native

```
window.ReactNativeWebView.postMessage("Hello from JavaScript!")
```

(b) Web page inside the WebView

Listing 3.1: Web → Native communication in React Native

```
function App(): JSX.Element {
  const fun = `
  (() => document.dispatchEvent(
    new MessageEvent("message", "Hello from React Native!")
  ))();
  true;`;
  return (<WebView source={{uri: "https://www.example.com/"}}
  ↪ injectedJavaScript={fun}/>)
}
```

(a) App code in React Native

```
window.addEventListener("message", (event) => {
  console.log(event.data)
})
```

(b) Web page inside the WebView

Listing 3.2: Web ← Native communication in React Native

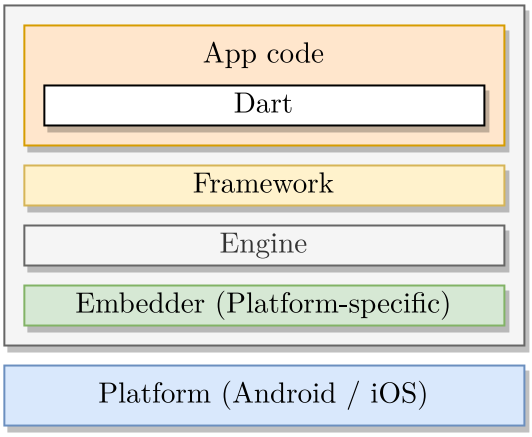


Figure 3.4: Architecture of an Flutter-based app [Goo23b]

is responsible for the implementation of Flutter's core API, plugin architecture, Dart runtime, file, and network I/O. A Flutter app is developed using the framework, which provides all the primitives for app developers. Building on the framework, the business logic itself is implemented in the programming language Dart.

Flutter supports the integration of web content via WebViews, comparable to React Native. There are two widely used packages for this purpose: webview_flutter [Goo23d] (3k stars) and flutter_inappwebview [Lor23] (2.5k stars). Similar to React Native both of these plugins support two-way communication between web content and native functionality. For the flutter_webview plugin, we briefly show here how this communication can be implemented in each direction:

(a) To send messages from the WebView to Flutter, a `JavaScriptChannel` has to be attached to the `WebViewController`. This channel is assigned a name, which can be used to reference it within the web page. By doing so, a global JavaScript object is attached to every frame, allowing the use of the `postMessage` method to send messages to Flutter (see Listing 3.3).

(b) To send messages from Flutter to WebView, the `runJavaScript` method can be employed. Similarly to the example in React Native, we use a `MessageEvent` to dispatch a message and an `EventListener` to process the message (see Listing 3.4).

```
controller = WebViewController()
  ..loadRequest(Uri.parse('https://www.example.com/'))
  ..setJavaScriptMode(JavaScriptMode.unrestricted)
  ..addJavaScriptChannel(
    'Channel',
    onMessageReceived: (JavaScriptMessage message) {
        debugPrint(message.message);
        // "Hello from JavaScript!"
    },
  );
```

(a) App code in Flutter

```
window.Channel.postMessage("Hello from JavaScript!")
```

(b) Web page inside the WebView

Listing 3.3: Web → Native communication in Flutter

```
controller = WebViewController()
  ..loadRequest(Uri.parse('https://www.example.com/'))
  ..setJavaScriptMode(JavaScriptMode.unrestricted)
  ..runJavaScript("""(() =>
    document.dispatchEvent(new MessageEvent("message", "Hello
    ↪  from React Native!")))()
    """);
```

(a) App code in Flutter

```
window.addEventListener("message", () => {
  console.log(event.data)
})
```

(b) Web page inside the WebView

Listing 3.4: Web ← Native communication in Flutter

	Cordova	React Native	Flutter
Language	HTML, CSS, JavaScript	JavaScript	Dart
UI Approach	WebView	Native Components	Custom Rendering Engine
WebView	Native	Plugin	Plugin
Web → Native	`addJavascriptInterface`	`postMessage`	`addJavaScriptChannel`
Web ← Native	`evaluateJavascript`	`injectedJavaScript`	`runJavaScript`

Table 3.1: Comparison between Cordova, React Native and Flutter

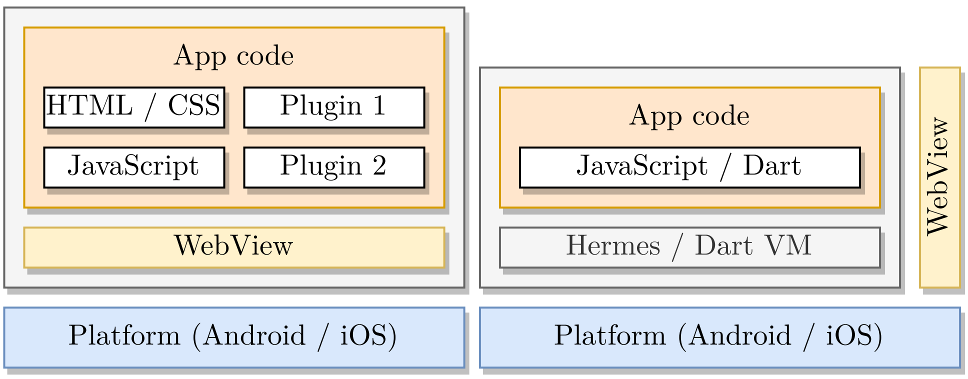
3.5 Summary

In this chapter, we have examined the underlying structure of the various XMFs. For reference, the main differences are summarized in Table 3.1. Apache Cordova is the only framework where apps are implemented on top of HTML, CSS, and JavaScript. React Native, on the other hand, is based on the React framework, so it still uses JavaScript, but no CSS or HTML. Flutter is based on its programming language Dart and is technically completely independent of web technologies. All three frameworks take a different approach to the UI: Cordova leaves rendering to the WebView implementation, React Native translates React Native components into regular Android or iOS views, and Flutter relies on its built-in rendering engine. Displaying web pages in WebViews is supported in all three frameworks: Cordova allows developers to use the WebView of the app itself, and React Native and Flutter have WebView components that are implemented through plugins. Concludingly, the frameworks differ significantly in their level of abstraction: Apache Cordova has a relatively thin abstraction layer, the app is simply rendered in a WebView, and the native functions are mapped into the WebView itself. React Native takes a more sophisticated approach to mapping app logic written with the React framework to native, platform-specific APIs. Flutter even renders the UI itself instead of relying on platform primitives. Regarding our first research question, we can conclude that modern XMFs, i.e., Flutter and React Native, unlike Apache Cordova, are not based on WebViews, but only allow their integration with the help of plugins. However, both frameworks still provide the ability to integrate web content with native content via communication interfaces.

CHAPTER 4

Vulnerability Analysis

In this chapter, we explore the vulnerabilities that arise from the integration of WebViews in XMFs. We begin our discussion by characterizing the attack surface along with the key frameworks under consideration in Section 4.1. In Section 4.2, we go through the customizations XMFs make to the platform's WebView. Then, from Section 4.3 to Section 4.7, we discuss each vulnerability, providing a concise overview, a threat model, and a relevant proof of concept for full understanding. Finally, we briefly outline the responsible disclosure process in Section 4.8, and summarize our analysis and findings in Section 4.9.



(a) Apache Cordova

(b) React Native, and Flutter

Figure 4.1: Comparison of the architecture of Cordova with that of React Native, and Flutter

4.1 Attack Surface

Based on the different approaches of how XMFs integrate WebViews, we discuss a diverse attack surface below. Unlike Cordova, React Native and Flutter are not inherently built on top of a WebView, but rather allow WebViews to be embedded, much like ordinary native apps. Figure 4.1 illustrates this distinction. With Cordova, the app code runs inside a WebView, and plugins that expose native functionality are mapped into the WebView. In contrast, React Native and Flutter execute the app code inside their runtime (i.e., Hermes or Dart VM). React Native and Flutter developers can include a WebView for web-based functionality they want to include in their app. Thus, the attack surface of React Native and Flutter apps is mainly identical to that of native apps. However, it is essential to note that all interactions with the underlying WebView are not in the platform's language (e.g., Java or Swift) but in JavaScript or Dart. This becomes critical when exploiting vulnerabilities, as inherent properties of a language can either open the door to further exploitation or make it impossible. For example, React Native uses JavaScript, which is prone to prototype pollution because it is dynamic, whereas Java is not, being statically typed. However, unlike JavaScript and Dart, Java allows reflection, which has already led to ACE in connection with WebViews [Wit13]. Furthermore, developers are limited to using the primitives (including their limitations) provided by the framework for communication and not building on those provided by the platform itself.

We will focus on React Native and Flutter for the remainder of this chapter, as Cordova is fundamentally different from React Native and Flutter in its architecture and attack surface, and Cordova has already been covered in detail by related work [AS17; GJS14; MS21].

4.2 Customization of the WebView

As introduced in Chapter 3, React Native and Flutter provide app developers with a platform-independent abstraction to integrate WebViews into their apps. To this end, the framework developers can customize the platform's WebView to their needs.

In this section, we systematically examine the customization of the platform WebView component in both React Native and Flutter, intending to find weaknesses stemming from the framework's code.

4.2.1 Methodology

We took a systematic approach, starting with creating minimal apps in each framework. These apps had no declared configurations and displayed only the link: `https://example.com`. Our next step was to analyze any changes or extensions to the code or behavior of the platform that came from the frameworks to identify potential vulnerabilities inherent in the framework's code.

```
Surface(modifier = Modifier.fillMaxSize()) {
  AndroidView(
    factory = { context ->
      WebView(context)
    },
    update = { webView ->
      // Load a web page
      webView.loadUrl("https://www.example.com")
    }
  )
}
```

(a) Android

```
<WebView source={{ uri: "https://www.example.com" }}/>
```

(b) React Native

```
@override
void initState() {
  final WebViewController controller = WebViewController();

  controller.loadRequest(Uri.parse("https://www.example.com"))
  _controller = controller;
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: WebViewWidget(controller: _controller),
  );
}
```

(c) Flutter

Listing 4.1: WebView component declaration in Android, React Native, and Flutter

Name	Version	Date
Apache Cordova	11.1.0	January 16, 2023
Flutter	3.10.0-7.0.pre.3	April 18, 2023
React Native	0.71.7	January 12, 2023
react-native-webview	12.0.2	April 4, 2023
webview_flutter	4.2.0	April 22, 2023

Table 4.1: Version numbers of XMFs, SDKs and plugins used for the vulnerability analysis

Our analysis methods included both static and dynamic analysis. For static analysis, we examined the source code of the WebView plugins react-native-webview [Rea23b] and webview_flutter [Goo23d]. For dynamic analysis, we used the debugger in Android Studio. This combination gave us a comprehensive view of the app's source code and run-time behavior. We paid particular attention to the customization of WebViews on Android, which is primarily handled through three distinct classes:

(a) `WebSettings` [And23c],

(b) `WebViewClient` [And23e], and

(c) `WebChromeClient` [And23b].

These classes, which offer different functionalities, are potential sources of vulnerabilities and represent the *exclusive* means for XMFs to modify the Android's WebView. Furthermore, developers can give web content access to the internal code of an app via the `addJavascriptInterface` [And23d] method. We have systematically examined all the methods of the above classes, using both the API documentation and the minimal apps, as well as any use of `addJavascriptInterface`.

Listing 4.1 shows WebView component declarations in Android and each framework. We only specify the URL to be navigated without further configurations in all three cases.

Table 4.1 shows the respective versions of the build tools, SDKs, and plugins used for our analysis. The Android device ran on an Android 13.0 Google APIs system image.

4.2.2 WebSettings

`WebSettings` [And23c] is a class that provides several configuration options for a WebView. For example, enable JavaScript execution, allow third-party cookies, or specify the user agent. Each WebView has its own `WebSettings` instance, which can be obtained by calling `webview.settings`.

However, the `WebSettings` instance returned is based on Java Native Interface (JNI), i.e., the only way to extract the settings from the object is to invoke each getter manually.

```
webView.settings.javaClass.methods
  .filter { it.name.startsWith("get") }
  .associate { method -> method.name to kotlin.runCatching { method.invoke(webView.settings) } }
  .toSortedMap()
```

Listing 4.2: Snippet to extract the values of a `WebSettings` [And23c] instance

This is because the attributes have been mangled and are thus no longer human-readable. We developed a script utilizing reflection to streamline calling all getters, as shown in Listing 4.2.

Table 4.2 illustrates the variations in the default settings of `WebSettings`. Any values that differ in at least one implementation are emphasized. Here, we provide a brief overview of these differences, and we refer to Section 4.7 for a comprehensive security assessment:

(a) `AcceptThirdPartyCookies` and `JavaScriptEnabled` are enabled in React Native, unlike the default setting and Flutter's.

(b) `AllowContentAccess` and `JavaScriptCanOpenWindowsAutomatically` are enabled in Flutter, unlike the default setting and Flutter's.

(c) `BuiltInZoomControls`, `DomStorageEnabled`, `LoadWithOverviewMode` and `useWideViewPort` are enabled in React Native and Flutter, unlike the default setting.

(d) `DisplayZoomControls` is enabled in React Native and Flutter, unlike the default setting.

	Android	React Native	Flutter
AcceptThirdPartyCookies	-	●	-
AllowContentAccess	●	-	●
AllowFileAccess	-	-	-
AllowFileAccessFromFileURLs	-	-	-
AllowUniversalAccessFromFileURLs	-	-	-
BlockNetworkImage	-	-	-
BlockNetworkLoads	-	-	-
BuiltInZoomControls	-	●	●
CacheMode		LOAD_DEFAULT	
CursiveFontFamily		cursive	
DatabaseEnabled	-	-	-
DatabasePath		""	
DefaultFixedFontSize		13	
DefaultFontSize		16	
DefaultTextEncodingName		UTF-8	
DefaultUserAgent		null	
DefaultZoom		MEDIUM	
DisabledActionModeMenuItems		0	
DisplayZoomControls	●	-	-
DomStorageEnabled	-	●	●
FantasyFontFamily		fantasy	
FixedFontFamily		monospace	
ForceDark		FORCE_DARK_AUTO	
JavaScriptCanOpenWindowsAutomatically	-	-	●
JavaScriptEnabled	-	●	-
LayoutAlgorithm		NARROW_COLUMNS	
LightTouchEnabled	-	-	-
LoadsImagesAutomatically	-	-	-
LoadWithOverviewMode	-	●	●
MediaPlaybackRequiresUserGesture	-	-	-
MinimumFontSize		8	
MinimumLogicalFontSize		8	
MixedContentMode		MIXED_CONTENT_NEVER_ALLOW	
NavDump	-	-	-
OffscreenPreRaster	-	-	-
PluginsEnabled	-	-	-
PluginsPath		""	
PluginState		OFF	
SafeBrowsingEnabled	-	-	-
SansSerifFontFamily		sans-serif	
SaveFormData	-	-	-
SavePassword	-	-	-
SerifFontFamily		serif	
StandardFontFamily		sans-serif	
TextSize		NORMAL	
TextZoom		100	
UseDoubleTree	-	-	-
UserAgent		0	
UserAgentString		""	
UseWebViewBackgroundForOverscrollBackground	-	-	-
UseWideViewPort	-	●	●
VideoOverlayForEmbeddedEncryptedVideoEnabled	-	-	-

Table 4.2: Default settings of WebView in Android, React Native, and Flutter derived from a `WebSettings` instance (- disabled, ● enabled)

4.2.3 WebViewClient

`WebViewClient` [And23e] serves as an interface to intercept *navigational* events triggered by a WebView [CW14]. These events encompass scenarios such as URL loading, error encounters, and changes to the visited history. To this end, it provides overridable methods to manipulate the default behavior. By default, WebView does not use a `WebViewClient`. Consequently, WebView does not independently manage navigation; instead, this responsibility is delegated to the OS. For example, when a user interacts with a HTTP URL within a WebView, the navigation request is passed to the OS, typically resulting in the opening of a web browser. However, React Native and Flutter both provide a `WebViewClient` instance, which means they have control over the navigation behavior of the WebView.

Table 4.3 presents a comparison of the behavior of the implementations. In the table, ϟ is used to denote instances where the request is canceled, **-** is used to denote a no operation, ◀ indicates that the method is not overridden but merely inherited, and ⥀ signifies that a method has been overridden, but only to update some internal state. Again, we briefly overview the differences and refer to Section 4.7 for a comprehensive security assessment.

When examining the table, some differences become apparent. Specifically, React Native: (a) uses the `onPageStarted` callback to inject JavaScript before the contents of the page, (b) provides credentials for a potential HTTP authentication request, and (c) inhibits crashing the app if the render process terminates. On the other hand, Flutter makes hardly any changes to the default behavior.

However, the callback for `shouldOverrideUrlLoading` behaves differently in all three cases:

(a) By default, the decision is left to the Activity Manager.

(b) For React Native, the URL is loaded within the WebView if it is on the allow list that corresponds to `['http://*', 'https://*']` by default.

(c) For Flutter, any URL is allowed within the WebView, but the `onPageStarted` callback can be used to restrict navigation to certain pages.

	Android	React Native	Flutter
`doUpdateVisitedHistory`	-	◀	⟲
`onFormResubmission`	ϟ	◀	◀
`onLoadResource`	-	◀	◀
`onPageCommitVisible`	-	◀	◀
`onPageFinished`	-	⟲	⟲
`onPageStarted`	-	Inject JavaScript snippet provided in the WebView declaration before content	Ask Navigation Delegate for permission
`onReceivedClientCertRequest`	ϟ	◀	◀
`onReceivedError`	-	⟲	⟲
`onReceivedHttpAuthRequest`	ϟ	Use the credential provided in the WebView declaration, otherwise ignore	◀
`onReceivedHttpError`	-	⟲	◀
`onReceivedLoginRequest`	-	◀	◀
`onReceivedSslError`	ϟ	⟲	◀
`onRenderProcessGone`	Crash the host app	Do not crash the host app	◀
`onSafeBrowsingHit`	Show interstitial page	◀	◀
`onScaleChanged`	-	◀	◀
`onTooManyRedirects`	-	◀	◀
`onUnhandledKeyEvent`	Notify the host app	◀	⟲
`shouldInterceptRequest`	-	◀	◀
`shouldOverrideKeyEvent`	-	◀	⟲
`shouldOverrideUrlLoading`	Ask the Activity Manager to choose the proper handler for the URL	Load the URL if it is on the allow list otherwise, issue an intent	Load the URL

Table 4.3: Behavior of `WebViewClient` callbacks in Android, React Native, and Flutter derived manually from source code (ϟ cancel request, - ignore, ◀ inherit, ⟲ update state)

	Android	React Native	Flutter
`onCloseWindow`	-	◄	-
`onConsoleMessage`	-	-	◄
`onCreateWindow`	-	-	Create window if the URL is secure, otherwise cancel
`onGeolocationPermissionsHidePrompt`	-	Request the geolocation permission	◄
`onHideCustomView`	-	◄	◄
`onJsAlert`	-	◄	◄
`onJsConfirm`	-	◄	◄
`onJsBeforeUnload`	-	◄	◄
`onJsPrompt`	-	◄	◄
`onJsTimeout`	-	◄	◄
`onPermissionRequest`	-	Grant permission request if the app is already granted, otherwise request permission	-
`onPermissionRequestCanceled`	-	◄	◄
`onProgressChanged`	-	-	-
`onReceivedIcon`	-	◄	◄
`onReceivedTitle`	-	◄	◄
`onReceivedTouchIconUrl`	-	◄	◄
`onRequestFocus`	-	◄	◄
`onShowCustomView`	-	◄	◄
`onShowFileChooser`	-	Show file chooser	-

Table 4.4: Behavior of `WebChromeClient` callbacks in React Native's, and Flutter's WebView derived from source code (- ignore, ◄ inherit)

4.2.4 WebChromeClient

`WebChromeClient` [And23b] provides developers an interface for managing events that are not handled by `WebViewClient` (see Section 4.2.3). However, the events are not related to the navigation of the WebView but rather the *surroundings* of the WebView. For example, JavaScript dialogs, website icons, titles, and the progress of a page loading within the WebView. The Android WebView does not use a `WebChromeClient` by default, but Flutter and React Native do.

Table 4.4 shows the different behavior of the implementations. In the table, - signifies that a method has been overridden, but only to update some internal state, and ◄ indicates that the method is not overridden but merely inherited. Again, we briefly overview the differences and refer to Section 4.7 for a comprehensive security assessment. When examining the table, some differences become apparent:

(a) React Native automatically grants permission requests if the user already granted permission to the app.

(b) React Native allows uploading files with a `<input type="file"/>` element, while Android's and Flutter's WebView do not.

4.2.5 JavaScript Interface

The JavaScript interface allows developers to give web content access to the app's internal code. To this end, developers can provide a handle via `addJavascriptInterface` to be used by JavaScript in loaded pages.

React Native and Flutter implement their communication interfaces from the WebView to the app, described in-depth in Chapter 3, via the JavaScript interface. In both cases, the JavaScript interface is only attached when a communication interface is enabled.

4.3 Origin Validation

This section investigates an attack vector initially identified by Yang et al. [Yan+18]. Interestingly, this particular attack vector also applies to XMFs's bridging functions described in Chapter 3. The attack is made possible by the fact that XMFs use the JavaScript interface to implement a mechanism that mimics the postMessage API known from the web, but *without any origin information*. However, origin checking is one of the cornerstones on the web to ensure secure communication between frames [MDN23b]. Given that browsers implement cross-window communication more securely, we consider this vulnerability to be a design flaw.

4.3.1 Threat Model

We assume a benign app implemented using XMFs, which uses the communication interfaces outlined in Chapter 3. We also assume that the attacker can hijack a frame contained within a benign page displayed in the app. This could be done in several ways:

(a) The attacker can trick the user into rendering an attacker-controlled frame. This can be done either through third-party content or by directing the user to a malicious page, for example by redirecting from a benign page or by deep linking. This attacker is usually referred to as a *gadget attacker* [BJM09].

(b) The attacker can use PITM attacks to take control of insecure connections and inject a malicious script into a benign frame.

4.3.2 Technical Details

Figure 4.2 illustrates all the flaws that lead to the vulnerability. Once the app has an interface to communicate between the WebView and the app enabled, a reference of the handler becomes accessible to *all* frames regardless of their origin, i.e., this may include malicious frames. Next, if the WebView communicates back to the app, the handler code cannot identify which origin has sent the message, i.e., it must process the request regardless. Consequently, within our threat model, an attacker can exploit this situation to elevate their privileges to the level of the handlers' functionality.

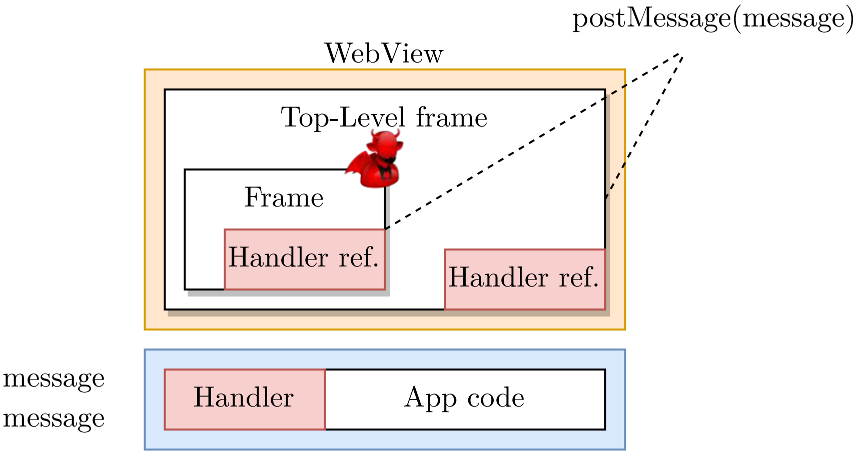


Figure 4.2: postMessage-like APIs in React Native and Flutter

```
window.addEventListener("message", (event) => {
    if (event.origin !== "https://example.com")
      return;
    ...
  }
)
```

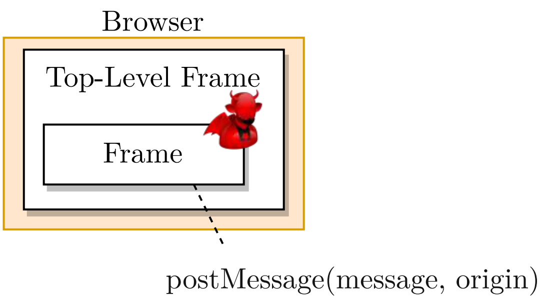


Figure 4.3: postMessage API on the web

Figure 4.3 shows the analogous process in a browser and makes the shortcomings more apparent: A correct implementation of the message handler in the top-level frame requires verifying the message's origin. The message handler returns if the origin does not match the permitted origin.

Contrary to the findings of Yang et al. [Yan+18] study, we found that communication is now possible from all frames, not just the top-level frame. This is due to recent changes made by Android to the API of the JavaScript interface. Previously, only the top-level frame had a reference to communicate with the app recently, these capabilities were extended. This change significantly increases the attack surface. The comparison between the 2018 and 2023 API documentation, which clearly illustrates these changes, is shown in Table 4.5.

On a side note, we would like to add that Yang et al. [Yan+18] did report the vulnerability to Facebook in 2018. However, Facebook did not fix the vulnerability but only add a security warning in their developer documentation [Met23d]. Furthermore, the

Snapshot	
November 21, 2018 [Way18]	June 1, 2023 [Way23]
The object is injected into the JavaScript context of the *main frame*	The object is injected into *all frames* of the web page, including all the iframes

Table 4.5: Changes to the Android API of WebView regarding `addJavascriptInterface`

security alert has been removed from newer React Native releases, even though the issue remains [Rea23c].

4.3.3 Proof of Concept

We give a PoC of this vulnerability in React Native: In Listing 4.3a a page *alice* that has another page *mallory* embedded via an iframe on a different origin is given. Furthermore, a handler is defined to expose the `ReactNativeWebView.postMessage` to *all* frames. In Listing 4.3b, a message is sent to this handler to redirect the WebView to a crafted web page, for example, to launch a phishing attack. Because the message handler in Listing 4.3c cannot distinguish whether the message comes from a trusted origin or not but assumes the former, it processes the message regardless. In conclusion, it is possible to launch a phishing attack in this example successfully, and because WebViews often do not have a proper UI, it is difficult to detect it as such [YHG19]. Worse, such a pattern could lead to a universal cross-site scripting (UXSS) vulnerability. That is, an cross-site scripting (XSS) vulnerability that works on sites that are not vulnerable themselves. To do this, the attacker sets the variable source to, for example, `javascript:alert(1)`. The JavaScript snippet is then executed in the context of the top-level frame.

4.4 Navigational Behavior

While studying the integration of WebViews in Section 4.2 we noticed that the WebView component in both XMFs React Native and Flutter enable navigation by default. That is the WebView will navigate to the page while remaining in the app when clicking on links. This behavior is different from the default, where the decision is left to the `ActivityManager`, i.e., `http://` URLs would be opened by the default browser. For example, assume the user is tricked into clicking an `<a href="https://example.com"></a>` element. In other words, the WebView is not used like a *view*, but more like a browser, and since WebViews, unlike browsers, do not provide a proper UI, we consider this (default) behavior a design flaw. Also, note that this problem does not exist in Android's WebView because navigation is not allowed by default, so the page being viewed is controlled by the developer alone.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>alice</title>
</head>
<body>
  <div name="origin"/>
  <iframe src="http://10.0.2.2:8000/"/>
</body>
<script>
    // trusted
    origin.textContent = location.origin
</script>
</html>
```

(a) Example of the trusted website *alice* embedding an untrusted iframe *mallory* on a different origin

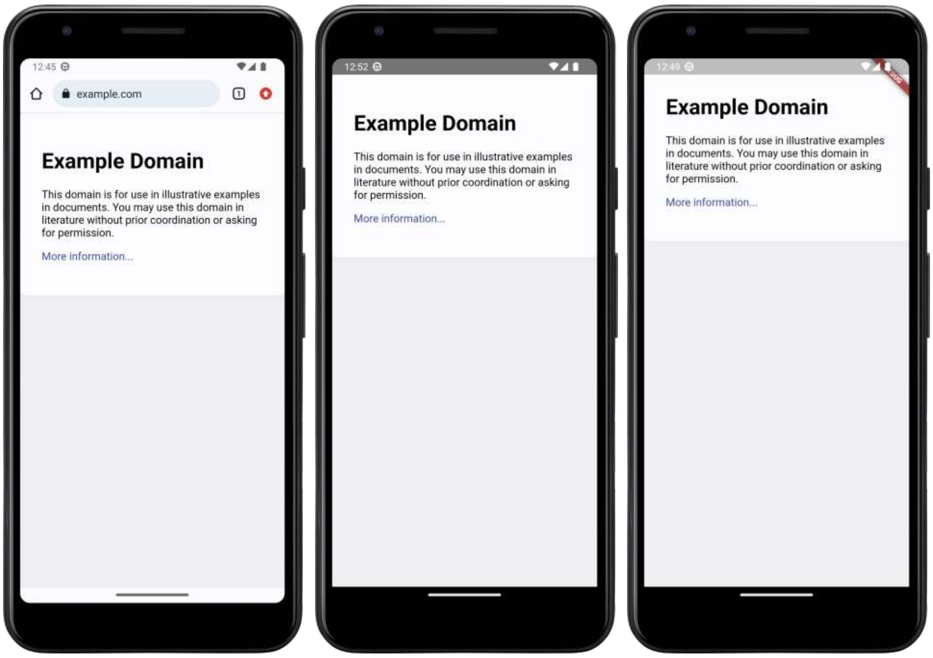
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>mallory</title>
</head>
<body>
  <div name="origin"/>
</body>
<script>
  // untrusted
  origin.textContent = location.origin
  const source = "http://evil.com/"
  ReactNativeWebView.postMessage(source)
</script>
</html>
```

(b) Example of an untrusted iframe *mallory*

```
return (
  <WebView
    source={source}
    onMessage={(event) => {
      // trusted
      const source = event.nativeEvent.data;
      const sink = source;
      setSource({
        uri: sink
      })
    }}
  />
);
```

(c) Example of an `onMessage` handler in React Native

Listing 4.3: PoC of the Origin Validation vulnerability



(a) Default WebView (b) React Native (c) Flutter

Figure 4.4: Comparison of the effects of navigation in a WebView in Android, in React Native, and in Flutter

Figure 4.4 shows what happens once the user clicks a (malicious) link in default WebView, in React Native, and Flutter: In the default WebView, the link is opened with the default browser, and it is apparent to the user that they left the current page (and are no longer in the app). In React Native and Flutter, the default behavior is navigating from the current to the new page. Since WebViews, unlike a browser, have no address bar, the user cannot recognize whether what he or she sees is still legitimate or a phishing attack.

4.4.1 Threat Model

We assume a benign app implemented using XMFs, with no predefined navigation boundaries, such as `originWhitelist` in React Native or `NavigationDelegate` in Flutter. We also assume that the attacker can trick the user into clicking a link within a WebView, possibly via a web-integrated chat feature or other similar functionality in the app. This threat model is significantly less stringent than the one proposed in Section 4.3.

4.4.2 Proof of Concept

We give a PoC of this vulnerability in Flutter: In Listing 4.4a, a web page with a malicious link is provided. In Listing 4.4b, the WebView navigates to this web page but does not have `NavigationDelegate` configured (i.e., navigation for the WebView is not restricted).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>mallory</title>
</head>
<body>
  <a href="https://www.evil.com">evil</a>
</body>
</html>
```

(a) Example of a web page containing malicious links

```dart
final WebViewController controller = WebViewController()
..loadRequest(Uri.parse("http://10.0.2.2:8000"));
```

(b) Example of a WebView not declaring any navigational boundaries

Listing 4.4: PoC of the Navigational Behavior vulnerability

We note that this is the default behavior, and developers must otherwise explicitly specify restrictions. Additionally, only the React Native API reference explicitly mentions that the default behavior is to navigate to all origins starting with `http[s]://` [Rea23c]; for Flutter, we could find no explicit warnings [Goo23d]. Once the user has been tricked into clicking on the link, the malicious web page can be used for either phishing purposes or information leakage if a JavaScript interface is additionally defined.

4.5 HTTP Authentication

React Native allows the WebView component to handle HTTP authentication according to RFC 7617 [Res15]. Listing 4.5 shows how this can be done in React Native. By default, React Native binds the credentials directly to the `onReceivedHttpAuthRequest` callback (see Section 4.2), which means the credentials are sent transparently to any server that asks for them. This behavior leads to a design flaw because the credentials are not bound to a specific origin, as browsers usually do, but to any origin. To make matters worse, in React Native all origins are allowed by default unlike the default behavior on Android (see Section 4.4).

4.5.1 Threat Model

We assume a benign app developed using React Native that uses HTTP authentication. The presumed attacker model consists of two potential attack vectors:

(a) The attacker lures the victim into sending a request to an attacker-controlled resource.

```
<WebView
  basicAuthCredential={{
    username: "username",
    password: "password",
  }}
/>;
```

Listing 4.5: Example usage of HTTP Authentication in React Native

(b) The attacker hijacks insecure connections, such as communication over HTTP.

However, we note that merely a request for so-called *passive content* suffices for the attack. Passive content refers to elements like `<img>` (`src` attribute), `<audio>` (`src` attribute), `<video>` (`src` attribute), `<object>` subresources, which are generally less protected compared to *active content* like `<script>`, `<link>`, `<iframe>` [MDN23a].

4.5.2 Proof of Concept

We developed a PoC to validate the described loophole. For this purpose, we extended the simple HTTP server shipped with Python to implement the HTTP authentication protocol. Listing 4.6 shows the implementation of the server. When a GET request is made, the server first searches for the `Authorization` header and if this is not present, it responds with an error code 401 and appends the `WWW-Authenticate` header to the response. React Native's WebView implementation then transparently sends the credentials to our server, which we then output on the standard output. On the vulnerable website, it is sufficient if any element (even inside iframes) has the server as `src`, e.g., `<img src="http://10.0.2.2:8000/">`. As this is not a navigational request, the attack works independently of any navigation boundaries such as the `originWhitelist`.

4.6 Custom Navigation Boundary

While studying the integration of WebViews in Section 4.2, we found another feature of Flutter that can be considered a vulnerability, which has its roots in the way the `NavigationDelegate` is implemented. The `NavigationDelegate` is responsible for limiting the web pages a WebView can navigate to in Flutter. Unlike React Native, Flutter implements this feature using the `onPageStarted` callback of `WebViewClient` instead of `shouldOverrideUrlLoading`. According to the Android API [And23e], the `onPageStarted` callback is only called for *top-level navigation*, i.e., navigating iframes or clicking on links whose target is an iframe will not trigger this callback. In other words, app developers may try to enforce navigation restrictions, but such restrictions can be easily circumvented by using iframes. Since merely the incorrect callback is used to implement the NavigationDelegate we classify the problem as an implementation error.

```python
from base64 import b64decode
from http.server import HTTPServer, BaseHTTPRequestHandler


class AuthHTTPRequestHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    if auth := self.headers.get("Authorization"):
      self.send_response(200)
      usr, pwd = b64decode(auth[6:]).split(b":")
      print(usr, pwd)
    else:
      self.send_response(401)
      self.send_header("WWW-Authenticate", 'Basic realm="WallyWorld"')
    self.end_headers()


with HTTPServer(("localhost", 8000), AuthHTTPRequestHandler) as httpd:
  httpd.serve_forever()
```

Listing 4.6: Simple RFC 7617-compliant HTTP server in Python

4.6.1 Threat Model

We assume that a benign app, developed using Flutter, loads a web page that uses iframes. The presumed attacker model consists of two potential attack vectors:

(a) The attacker manipulates the victim into clicking a malicious URL inside an iframe.

(b) The attacker hijacks insecure connections, such as communication over HTTP.

4.6.2 Proof of Concept

We developed a PoC to confirm the vulnerability mentioned above, as detailed in Listing 4.7. First, we created a WebView with a specified `NavigationDelegate`, which is designed to prevent any navigation by consistently returning `NavigationDecision.prevent`. The loaded web page contains only a single iframe that fills the entire screen. As all navigation occurs within this iframe, the `NavigationDelegate` is effectively bypassed, demonstrating the vulnerability.

```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>alice</title>
</head>

<body>
  <iframe src="http://www.evil.com" style="width: 100%; height: 100%; border: none;"></iframe>
</body>
</html>
```

(a) Example of a web page containing malicious links

```dart
final WebViewController controller = WebViewController()
..loadRequest(Uri.parse("http://10.0.2.2:8000"))
..setNavigationDelegate(
    NavigationDelegate(
      onNavigationRequest: (NavigationRequest request) {
        return NavigationDecision.prevent;
  },
));
```

(b) Example of a WebView not declaring any navigational boundaries

Listing 4.7: PoC of the Custom Navigation Boundary vulnerability

Framework	Setting	Consequences	Reference
React Native	`JavaScript␣Enabled`	XSS vulnerabilities	Luo et al. [Luo+11]
React Native	`onPermission␣Request`	Permission Delegation	Beer et al. [Bee+22]
Flutter	`JavaScript␣CanOpenWindows␣Automatically`	WebView UI Redressing attacks	Yang, Huang, and Gu [YHG19]

Table 4.6: Overview of the insecure default settings of React Native and Flutter

4.7 Insecure Defaults

As our investigation in Section 4.2 shows, React Native and Flutter use different default settings. From a security perspective, it is fundamentally important, especially for frameworks, that the default settings are configured as securely (and consistently) as possible. Insecure default settings lead to an increased attack surface and intentionally force developers to inform themselves extensively about the security settings and their effects. In addition, app developers may assume certain (secure) settings. However, suppose they differ from framework to framework. In that case, it stands to reason that these settings are not verified separately but assumed to be correctly set, leaving the app more vulnerable than needed. In this section, we summarize the security-relevant differences and discuss their implications.

Table 4.6 gives an overview of the insecure defaults of the WebView integration of React Native and Flutter. We discuss each of these settings below:

(a) React Native has `JavaScriptEnabled` option enabled by default. While we acknowledge that this setting likely has its roots in ensuring compatibility, this feature introduces a potential avenue for XSS vulnerabilities. In light of this, we align ourselves with the recommendations proposed by security experts [And12]. Specifically, this setting should be changed thoughtfully and intentionally rather than being the automatic default.

(b) React Native handles permission handles transparently, i.e., if permissions have already been granted for the app, they are also granted to *all* websites inside the WebView. This was found to be a problem by Beer et al. [Bee+22], who identified that behavior as risky. We agree because developers usually assume that React's WebView works like the default WebView and asks for extra permissions. Adding to the problem, React Native apps, like a browser, allow navigation by default, making this behavior even more of an issue.

(c) By default, Flutter has the `JavaScriptCanOpenWindowsAutomatically` option enabled, which can make it vulnerable to so-called *WebView UI Redressing attacks* as described by Yang, Huang, and Gu [YHG19].

Vulnerability	**Section**	**Framework**	
		React Native	Flutter
Origin Validation	4.3	●	●
Navigational Behavior	4.4	●	●
HTTP Authentication	4.5	●	-
Custom Navigation Boundary	4.6	-	●
Insecure Defaults	4.7	●	●

Table 4.7: Overview of the vulnerabilities found in React Native and Flutter

4.8 Responsible Disclosure

We reported all five of the vulnerabilities we found to the React Native and Flutter developers, following the responsible disclosure process [Goo23c]. We provided them with a comprehensive report detailing the issues found. However, at the time of writing, we have not received a response from the developers.

4.9 Summary

It is difficult to find vulnerabilities solely for XMFs since *modern* XMFs are simply ordinary native apps these days. As for vulnerabilities related to WebView, newer frameworks such as React Native and Flutter have drastically reduced the attack surface compared to Apache Cordova. While inherent vulnerabilities of WebView also apply to the WebViews in XMFs, our investigation is limited to problems caused by the wrapping introduced by XMFs.

Nevertheless, we found some weaknesses in both frameworks, mainly by comparing Android's default WebView with React Native's and Flutter's. In addition, we distinguish between vulnerabilities that are more likely to be due to implementation flaws and those that are more likely to be due to design flaws:

(a) **Origin Validation.** Strictly speaking, Origin Validation can be considered an implementation flaw, resulting from developers making incorrect assumptions about message origins. However, it can also be argued that it is a design flaw, especially when comparing how browsers implement cross-window communication APIs.

(b) **Navigational Behavior.** In modern XMFs, WebViews function similarly to browsers, but lack a proper UI that displays critical information such as the URL of the currently loaded page. Combined with the fact that navigation is allowed by default, it is fairly easy to launch hard-to-detect phishing attacks. We consider the unfavorable combination of allowed navigation and missing feedback to the user about the currently displayed page as a design flaw. In addition, we notice that a default WebView in Android does not allow navigation, i.e., this is a problem that occurs especially with WebViews in modern XMFs.

(c) **HTTP Authentication.** Unlike browsers, React Native's WebView does not tie credentials to a specific origin, i.e., the authentication mechanism is flawed by design.

(d) **Custom Navigation Boundary.** As simply the wrong callback is used to restrict navigation within the WebView, we classify this as an implementation error.

(e) **Insecure Defaults.** Since the default settings are design decisions, we also classify them as design flaws.

Table 4.7 overviews the vulnerabilities found and shows which of the two frameworks is affected.

CHAPTER 5

Security Impact Assessment

In this chapter, we explore the impact of the vulnerabilities described in Chapter 4 by quantifying the apps using related features. Our first step towards this goal is to assemble a representative dataset, which we describe in detail in Section 5.1. We then provide a deeper insight into the methodology of our large-scale analysis in Section 5.2. In Section 5.3 we present a discussion of our results. Next, in Section 5.4, we attempt to find case studies that highlight the practical implications of the proposed attacks. Finally, in Section 5.5, we highlight the limitations of our analysis approach and identify potential areas for future refinement.

5.1 Dataset

To the best of our knowledge, there is no comprehensive catalog of apps built with XMFs. Therefore, our first step was to create a dataset based on an existing dataset provided by our research group. The original dataset was compiled using a list of apps from AndroZoo [All+16], with data current as of December 6, 2022. Additionally, this dataset was filtered to include only those apps that have one million or more installs as indicated by their metadata, and whose package names are listed on the Google Play Store. We refined the dataset by filtering out all apps that were not built using the XMFs that were

Filtering Method	**# of apps**	
AndroZoo [All+16]	2,489,612	(100 %)
Filter 1: $\geq$ 1,000,000 downloads	54,988	(2.22 %)
Filter 2: Available on Google Play Store	50,831	(92.44 %)
Filter 3: Built using XMFs	3,441	(6.77 %)

Table 5.1: Number of apps remaining in the dataset after applying each filter

Framework	Recognition Method
Apache Cordova [The23a]	Path `res/xml/config.xml` exists [Mal+15]
React Native [Met23c]	File `libreactnativejni.so` exists
Flutter [Goo23a]	File `libflutter.so` exists

Table 5.2: Artifacts in Android apps that can be used to detect the framework

the focus of this thesis, namely Cordova, React Native, and Flutter. This was done using the recognition methods outlined in Table 5.2 and resulted in a final dataset of 3,441 candidate apps.

Table 5.1 shows the number of apps remaining after applying each of the *filters* described above.

To find out what XMF apps are using, we analyzed some of the apps from the raw dataset with JADX [Sky23] and found a proper way to identify the framework used.

Table 5.2 shows the result of this analysis, i.e., it shows the files that must be included in the APK files if the app was written using one of the XMFs in question. We note that Malavolta et al. [Mal+15] have already investigated this for older frameworks. However, to our knowledge, *modern* frameworks such as React Native and Flutter have not yet been studied.

5.2 Evaluation Methodology

In terms of analysis methodology, we chose static program analysis to analyze the 3,441 applications in our large-scale analysis. In doing so, we follow previous research that highlights the need for a static approach because WebViews often require specific, hard-to-reproduce states [YHG19]. A motivating example is Facebook Messenger, where it is necessary to be logged in and receive a message containing URL to open a WebView.

We based our analysis on the bulk analysis feature of AndroGuard, known as *AndroAuto* [DGB23]. Below, we describe the steps our analysis pipeline goes through and the key customizations we made to AndroGuard:

(a) AndroGuard uses multithreading for parallelization, but Python's approach is not efficient for heavy computational tasks due to the global interpreter lock (GIL) [Pyt23]. To counter this, we adapted the AndroGuard implementation to use multi-processing instead, which allowed us to fully utilize the 25 cores on our evaluation VM. As part of the analysis pipeline, we divided the dataset into $\left\lceil \frac{n_{APK}}{n_{\mathrm{CPU}}} \right\rceil$ batches, each handled by a dedicated worker process.

(b) We then checked if an app is a split APK and merged it into a normal APK using APKEditor [REA23], as split APKs are not supported by AndroGuard.

(c) We then used AndroGuard's `APK.get_files` method to identify the XMF used (see Table 5.2), and the `Analysis.find_classes` method to determine if the app uses WebViews, indicated by classes starting with `"Landroid/webkit/WebView"`. If neither condition was met, we stopped further analysis.

(d) For React Native apps, we first determined if the react-native-webview plugin was used by searching for classes starting with `"Lcom/reactnativecommunity/webview"`. Additionally, we retrieved the *bundle*, which is typically located at `assets/index.android.bundle`. This bundle contains the business logic of the app, essentially the JavaScript code written by the app developers. Depending on the version of React Native being used, this bundle may contain minified JavaScript or Hermes bytecode. Given the heterogeneity of the bundle and the lack of practical analysis tools for React Native, we decided to rely on regular expressions to search for matches of `/ReactNativeWebView.postMessage\(.*?\)/`. Based on the calls to `postMessage`, we then inferred the functionality of each handler. For example, from `ReactNativeWebView.postMessage(JSON.stringify({message: 'openUrl', url}))`, we inferred that the handler could be used to open URLs. We also looked for matches of `/originWhitelist:\[.*?\]/`, and `/basicAuthCredential:\{.*?\}/`. However, these could only be found in bundles that are not based on Hermes bytecode.

(e) For apps developed with Flutter, we verified that the webview_flutter plugin is used by detecting classes starting with `"Lio/flutter/plugins/webviewflutter"`. The business logic for Flutter apps is contained in a shared object, usually named `libapp.so` [Ost23]. However, the regular expression approach, while effective for React Native, failed to produce results for Flutter-based apps.

(f) Finally, we generated a report that was exported as a JSON file and used for further analysis.

5.3 Experimental Results

Taking the individual reports in JSON format, we imported all the data into a Pandas `DataFrame` [Pan23]. We then used the query capabilities of Pandas to make a quantitative assessment of the prevalence of potentially vulnerable features, which allows us to assess the potential impact of the vulnerabilities identified in Chapter 4.

5.3.1 Usage of XMFs

Before quantifying the usage of features with potential vulnerabilities, we determined the number of apps within the dataset that use each XMFs.

Figure 5.1 visualizes how many of the 3,441 candidate apps in our dataset are implemented using one of the three XMFs. In total, we detected 1,578 (45.86 %) apps based on React Native, 1,147 (33.33 %) apps based on Flutter, and 716 (20.81 %) based on Cordova. The

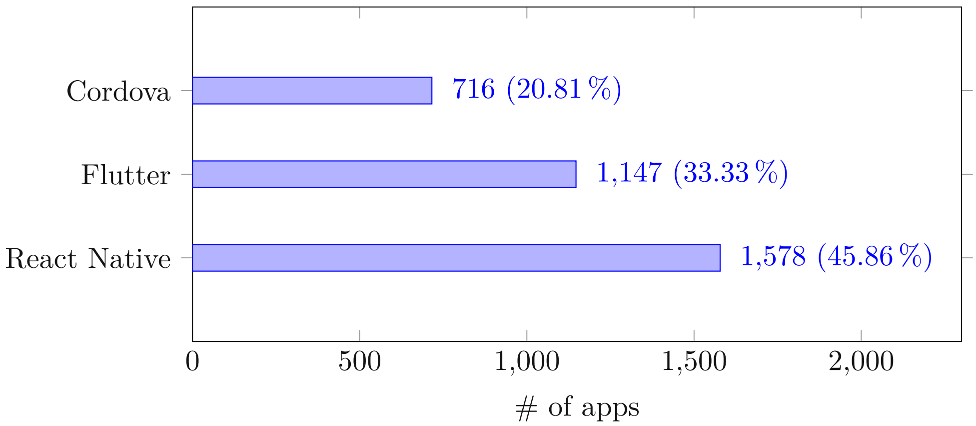


Figure 5.1: Number of apps built using XMFs within 3,441 candidate apps

breakdown is consistent with the survey of Lionel Sujay Vailshery [Lio22] in that React Native and Flutter are undisputedly the most important XMFs. React Native's lead can be explained by the fact that React Native has been more popular for a longer period and is, therefore, more represented in our dataset of the top one million apps.

To further emphasize the widespread use of XMFs, we provide a comprehensive breakdown of XMF usage in the ten most common categories, detailed in Figure 5.2. In addition, we provide a list of the top five most downloaded apps for each XMF in Table 5.3

5.3.2 Usage of WebView Plugins

In this thesis, we aimed to find bugs in the implementation of WebViews in XMFs, and we decided to focus on the WebView plugins of React Native and Flutter. In principle, however, both XMFs can be used completely without WebViews. Therefore, the next step was to investigate how many of the apps in our dataset use these WebView plugins at all, so that we can narrow down our further analysis.

Figure 5.3 shows that 579 (36.69 %) out of 1,578 React Native-based apps use the react-native-webview plugin, and 1,004 (87.53 %) out of 1,147 Flutter-based apps use the webview_flutter plugin.

Our results show that WebView plugins are not a minor feature in either React Native or Flutter, but rather are widely used. In particular, we find that React Native-based apps make significantly less use of WebViews, compared to Flutter-based apps. We suspect that this discrepancy is because it may be easier to migrate web code into app code in React Native, as both are written in JavaScript.

5.3.3 Usage of the JavaScript Interface / Origin Validation

Referring to the vulnerability presented in Section 4.3, we investigate how many apps make use of the JavaScript interface. We only consider React Native here, as there is no

App Name	**App ID**	**# Installs**
Facemoji	`com.simejikeyboard`	500M
LINE	`jp.naver.line.android`	500M
YONO SBI	`com.sbi.lotusintouch`	500M
Slotomania^{TM}	`air.com.playtika.slotomania`	50M
Toon Cup	`com.turner.tooncup`	50M

(a) Apache Cordova

App Name	**App ID**	**# Installs**
B612 AI Photo & Video Editor	`com.linecorp.b612.android`	500M
Google Pay	`com.google.android.apps.nbu.paisa.user`	500M
PUBG MOBILE	`com.tencent.ig`	500M
StarMaker	`com.starmakerinteractive.starmaker`	100M
YouCam Makeup	`com.cyberlink.youcammakeup`	100M

(b) Flutter

App Name	**App ID**	**# Installs**
Microsoft Excel	`com.microsoft.office.excel`	1B
Microsoft PowerPoint	`com.microsoft.office.powerpoint`	1B
Microsoft Word	`com.microsoft.office.word`	1B
Skype	`com.skype.raider`	1B
Viber	`com.viber.voip`	1B

(c) React Native

Table 5.3: Top five most downloaded apps for each XMFs

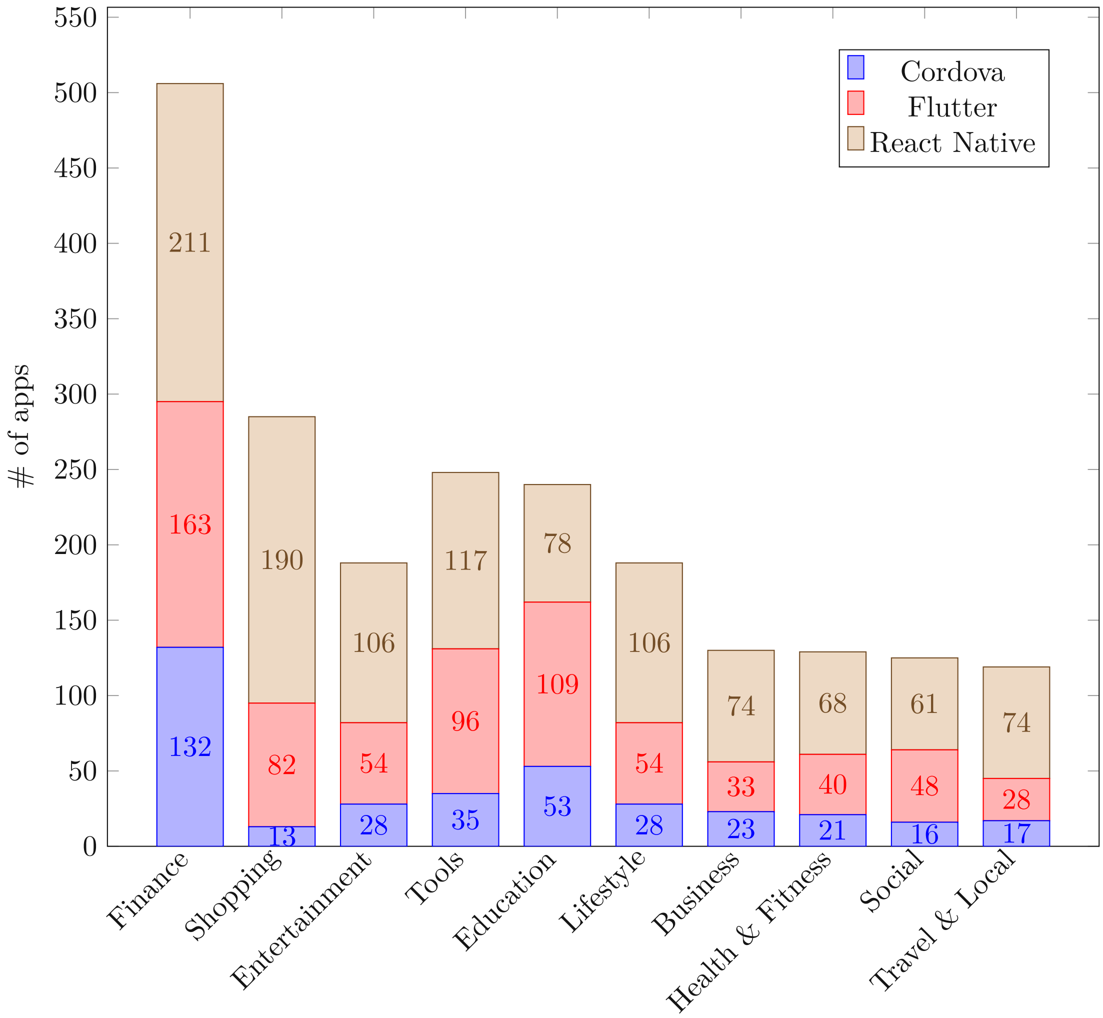


Figure 5.2: Breakdown of XMF-usage among the ten most common categories in the Google Play Store

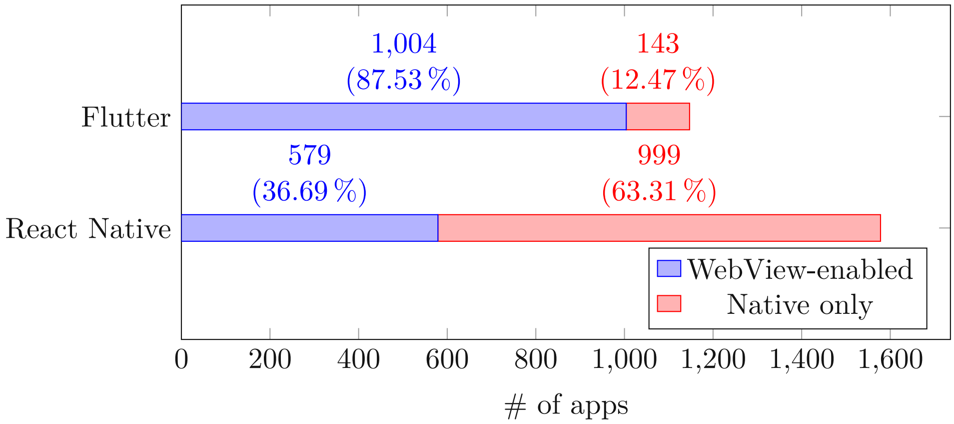


Figure 5.3: Number of XMF-based apps using WebView plugins

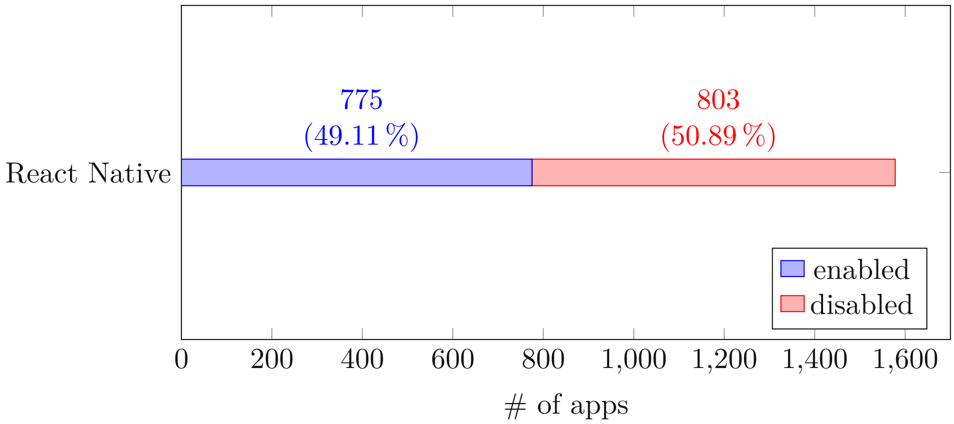


Figure 5.4: Number of React Native-based apps using the JavaScript interface

analysis methodology for Flutter. Overall, we detected that 775 (49.11 %) of the 1,578 React Native apps use the JavaScript interface (see Figure 5.4). For the remaining 803 (50.89 %) apps we were unable to detect the use of the JavaScript interface.

To get a comprehensive view of the usage of the JavaScript interface, we performed a detailed manual analysis of the calls to `ReactNativeWebView.postMessage`. To do this, we used certain *keywords* to categorize the apps. If any of these keywords were found within a call to `ReactNativeWebView.postMessage`, we added the the appropriate category. We note that an app could potentially fall into multiple categories based on multiple keywords occurring in the calls.

Table 5.5a shows the keywords we used to broadly categorize the calls, along with a typical message for the category. For the sake of brevity, we have omitted `ReactNativeWebView.postMessage` from the table, and only list the parameters.

Figure 5.5b shows the results of our categorization approach, quantifying the different purposes for which developers use the JavaScript interface. An important finding is that 70 (9.03 %) apps likely implement navigation, while likely 63 (8.13 %) apps execute native methods through the JavaScript interface. These apps are particularly noteworthy because of their relevance to the Origin Validation vulnerability (see Section 4.3) we identified. In addition, the most common use case seems to be event handling, which includes actions such as screen resizing and error handling, observed in 294 (37.93 %) apps. Despite our efforts, the purpose of communication could not be determined for 213 (27.48 %) apps based solely on calls to `ReactNativeWebView.postMessage`. We also identified 122 (15.74 %) apps that use a plugin called react-native-youtube-iframe [Kan23], which inherently uses the JavaScript interface. We also found that 44 (5.68 %) apps utilize the JavaScript interface for authentication purposes.

5.3.4 Navigational Behavior

Referring to the vulnerability presented in Section 4.4, we explore the extent to which navigation is allowed in apps using the react-native-webview plugin. Again, we are only considering React Native here, as there is no analysis methodology for Flutter. Overall, we were able to detect the use of the origin allowlist in 315 out of 1578 apps (19.96 %); we suspect that the remaining apps fall back to the default `['http://*', 'https://*']`.

Figure 5.6 illustrates the URI schemes which are allowlisted in React Native-based apps. We note that due to apps containing multiple whilelisted schemas, or even multiple WebViews, an app may fall into multiple buckets. It turns out that most React Native developers do not restrict navigation to specific origins at all. Even worse, developers rather relax the default restrictions on HTTP(S) and allow, e.g., `"*"` or file URLs. If unrestricted (i.e., `"*"`) navigation is considered vulnerable, 282 apps (89.52 %) fall into this category. However, this vulnerability is basically at the beginning of the exploit chain. Starting from here, the WebView can be navigated to a malicious web page, which then allows further steps such as displaying a phishing page or exfiltrating information via the JavaScript interface.

5.3.5 HTTP Authentication

Regarding the vulnerability presented in Section 4.5, we investigated how many and which applications use the HTTP authentication feature of the react-native-web-view plugin. However, our analysis shows that no application in our dataset uses HTTP authentication.

5.3.6 Custom Navigational Boundary

Regarding the vulnerability presented in Section 4.6, we are unable to provide specific numbers due to the lack of a analysis methodology for Flutter. In principle, however, all apps that use iframes can bypass the navigation restrictions enforced by the app developers within the iframes.

Category	**Keywords**	**Example Message**

Authentication	auth, token, authentication	`{`
		`action: true,`
		`event: "APPLE_AUTHENTICATION",`
		`params: {},`
		`}`

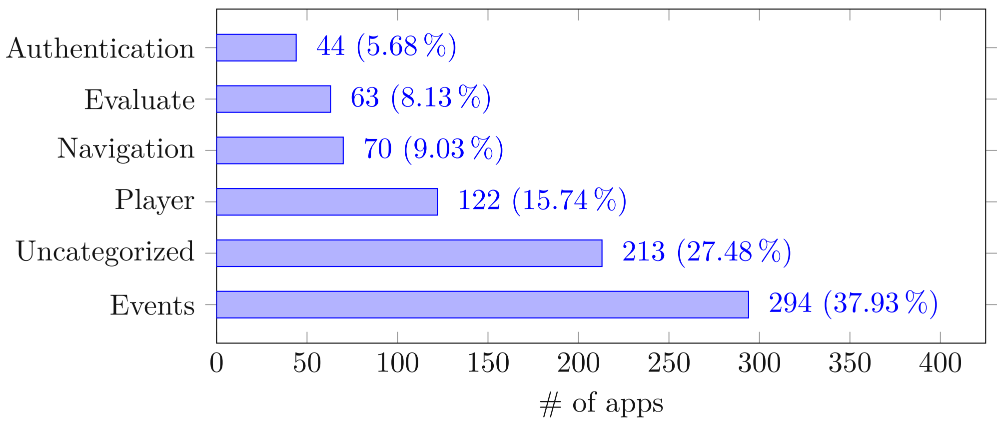
Evaluate	action, method, eval, set	`{`
		`"methodName": "closeMiniApp"`
		`}`

Events	event, zoom, error, ready, failed, complete	`{`
		`action: "injectReady"`
		`}`

Navigation	navigation, href, link	`{`
		`href: link.getAttribute('href')`
		`}`

Player	player.	`{`
		`eventType: "getVideoUrl",`
		`data: player.getVideoUrl()`
		`}`

(a) Keywords used to classify the purpose of using the JavaScript interface



(b) Intended use of the JavaScript interface in React Native-based apps

Figure 5.5: Analysis of the JavaScript interface utilization in React Native-based apps

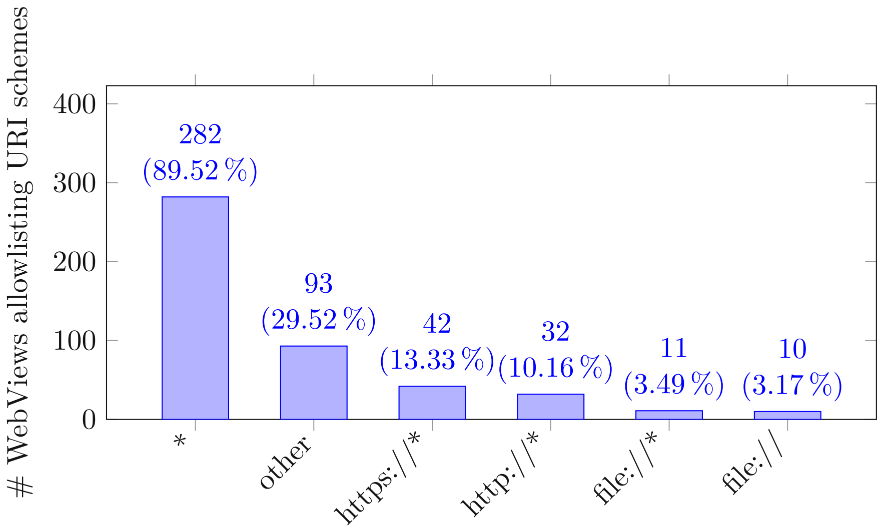


Figure 5.6: Breakdown of URI schemes allowlisted in React Native-based apps

5.4 Case Studies

To illustrate the practical implications of the proposed attacks, we back them up with relevant case studies. However, as manual analysis proved to be too time-consuming, we were not able to find a case study for every attack.

5.4.1 Origin Validation

We have tried to find an exploitable app for the vulnerability presented in Section 4.3 but without success.

Table 5.4 lists the apps that we have manually analyzed in detail. All apps use the JavaScript interface, as can be seen from the given example message. We would like to point out that the manual analysis is comparatively time-consuming, as getting through the registration wall and finding the WebViews within the apps already takes quite some time. After identifying the WebViews, we first tried to simulate a network attacker via HTTP Toolkit [Tim23], but were already unsuccessful, i.e., we did not make any further attempts as a web attacker. Therefore, we leave the search for exploitable apps to future work.

5.4.2 Navigational Behavior

The Dictionary.com (`com.dictionary`) app, a widely used English dictionary and thesaurus tool built with React Native, serves as a strong example of the *Navigational Behavior* vulnerability described in Section 4.4. Many of the app's features are provided within a WebView.

App Name	App ID	Example Message
Best Buy	com.bestbuy.android	e.data

Hornbach	de.hornbach	{
		  type:'updateToken'
		}

MetaMask	io.metamask	{
		  type: 'NAV_CHANGE',
		  payload: {
		    url: location.href,
		    title: title
		  }
		}

Microsoft Outlook	com.microsoft.office.outlook	newTabLinks
Ryanair	com.ryanair.cheapflights	e
Sportschau	de.apptelligence.sportschau.android	e.target.href

Steuerbot	com.steuerbot	{
		  type: 'externalLink',
		  url: current.href
		}

UEFA Gaming	com.uefa.eurofantasy	e.target.href
Wizz Air	com.wizzair.WizzAirApp	e

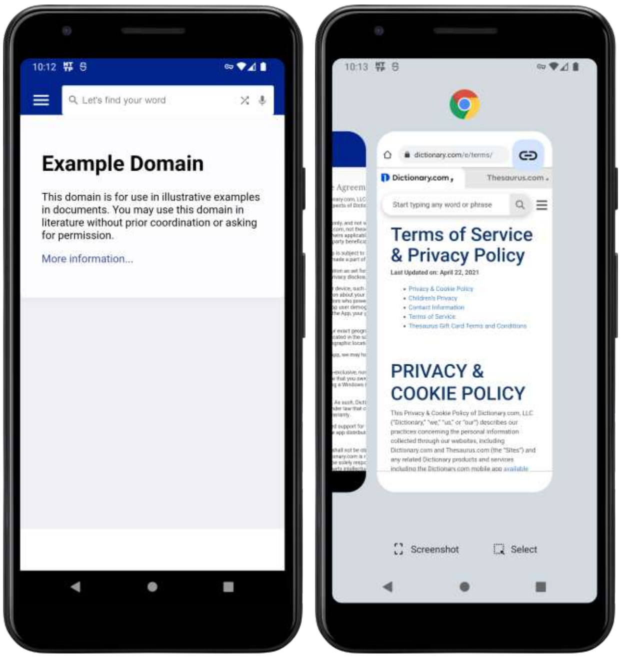
Zip	com.quadpay.quadpay	{
		  url: window.location.href
		}

Table 5.4: Apps manually analyzed in an attempt to find an exploitable app concerning Origin Validation

Our analysis shows that all web pages are allowlisted, meaning that the main WebView allows navigation to any website, including malicious ones. This is particularly problematic because users cannot tell whether they are on a legitimate or malicious site as there is no address bar. We were able to navigate to the page we wanted simply by using the social media links already on the website. These can be found at the bottom of the page, and as users can link to external content from there, they can be used to navigate the WebView to arbitrary web pages.

The presence of a second WebView that behaves correctly is strong evidence of an unintentional bug in the main WebView. The second WebView is located in Settings, *About Dictionary.com*, *End-User License Agreement*. Clicking a link there launches it in the default browser, as expected.

Figure 5.7 demonstrates the contrast between the navigation behaviors of the two WebViews. In Figure 5.7a, the WebView navigates unhindered and could display a phishing page. This scenario poses a risk because users do not have the tools to determine the authenticity of the displayed page (e.g., an address bar). Conversely, Figure 5.7b shows a noticeable change in the view, allowing users to identify the new page.



(a) Vulnerable WebView (b) Expected Behavior

Figure 5.7: Comparison of the effects of navigation in the two WebViews of the Dictionary.com app

5.5 Limitations

Considering that we are the first to systematically examine XMFs and React Native and Flutter apps in particular for vulnerabilities, our assessment approach has some limitations.

5.5.1 False Negatives

- As we have chosen a static approach, we inherit the characteristic weaknesses of these approaches, namely: (a) their susceptibility to obfuscation, and (b) the inability to examine dynamically loaded content. These inherent properties of static analysis can introduce false negatives (FNs). For example, should a React Native-based app load a remote URL (or a local file) instead of a local static web page, we would not find any use of the string `"ReactNativeWebView.postMessage"` even though the app might use the JavaScript interface extensively.
- As our approach for finding further details of XMF-based apps is essentially based on a simple string-matching inside the bundle, this approach only worked for React Native-based apps. We were not able to get details from Flutter-based apps, which again leads to FNs.

- As we only searched for the bundle for React Native-based apps under `assets/android.index.bundle` (the default directory), 161 apps escaped further analysis, leaving them all as potential FNs.
- As we looked for classes that start with `"Lcom/reactnativecommunity/webview"` to find out if a React Native-based app uses WebViews, our approach only works for apps based on React Native version 0.60 (July 3, 2019) or later. Prior to version 0.60, the react-native-webview plugin was part of React Native itself, meaning that any app would technically be marked as not using the WebView plugin, leaving them all as potential FNs.

5.5.2 False Positives

- While the primary concern is often FNs, a static approach can also introduce false positives (FPs). For example, any app that contains the string `"ReactNativeWebView.postMessage"` will be categorised as using the JavaScript interface, even if that string simply exists in the codebase. This condition can potentially lead to the introduction of FPs if this specific code path is never executed.

CHAPTER 6

Conclusion and Future Work

In this chapter, we first conclude our work in Section 6.1 and further outline possible future work in Section 6.2.

6.1 Summary

We summarize the main findings of this thesis through a review of the research questions in Section 1.1.

> **[RQ1]** How are WebViews integrated into XMFs? What bridging mechanisms are used by the most popular frameworks?

The XMFs studied in this thesis, Cordova, React Native, and Flutter, integrate WebViews in different ways: Cordova uses WebViews as the main app component. A Cordova app is just a full-screen WebView running a web app. Contrarily, *modern* XMFs such as React Native and Flutter do not necessitate WebViews. However, a WebView component can be integrated via plugins to display web content within an app. This approach is similar to ordinary native apps, where developers can decide whether or not to include WebViews by deriving the WebView class.

The bridging mechanisms used also differ: In Cordova, the use of *plugins* (not to be confused with WebView plugins) allows the unification of the web and native realms. These plugins typically consist of a native code segment that abstracts the underlying platform and a web code segment that allows developers to access native functionality. Communication between these components is facilitated by four different bridges (see Section 3.2), which can optionally use WebView's APIs such as `addJavascriptInterface`, `loadUrl` or `evaluateJavascript`. In contrast, *modern* XMFs use interfaces reminiscent of the postMessage API typically found on the web. As such, they rely on the `addJavascriptInterface` underneath the surface. However, there is an important distinction to be made here: while Cordova uses its bridging mechanisms to add native functionality to the web, modern XMFs provide a communication channel. This channel allows web apps to interact with the native environment for various purposes rather than explicitly adding native functionality to the web domain.

> **[RQ2]** What vulnerabilities arise from the integration of WebViews in XMFs? Do these bridging mechanisms allow attackers to escalate privileges beyond the web context?

Our focus was primarily on React Native and Flutter, given that Cordova's vulnerabilities have been thoroughly explored in previous work. In the course of our investigation, we were able to identify and detail five unique vulnerabilities associated with the WebView integration in React Native and Flutter. These vulnerabilities include surreptitious access to methods injected via the JavaScript interface (see Section 4.3), lack of effective restrictions in terms of navigation (see Section 4.4), an HTTP authentication which sends the credentials mindlessly (see Section 4.5), a navigational boundary that overlooks navigation within iframes (see Section 4.6), and insecure defaults (see Section 4.7).

Despite these vulnerabilities, we note that the WebViews in modern XMFs are largely analogous to those in native apps. This similarity suggests that escaping the web context would pose a significant challenge. Furthermore, if such an escape were to occur, it would likely not be exclusive to XMFs, but possible in all apps.

> **[RQ3]** How many apps are affected and what are the consequences of the vulnerabilities?

As the first to systematically analyze modern XMFs, namely React Native and Flutter, we have encountered the inherent complexity these new platforms introduce to security researchers. This complexity is primarily due to proprietary runtimes lacking sufficient tool support, making comprehensive security assessment difficult. For this reason, we could not determine the exact number of vulnerable apps but can only narrow it down to apps that use functions related to the vulnerabilities we found.

First, out of 50,831 apps in our dataset, we found that 3,441 were built using XMFs. In terms of platform distribution, 20.81 % use Cordova, 33.33 % use Flutter, and 45.86 % are built with React Native. In addition, our analysis confirmed that WebViews are a key feature of XMFs, highlighting the importance of investigating their security. Specifically, a significant 87.53 % of Flutter apps and 36.69 % of React Native apps include WebView plugins.

Regarding the vulnerabilities we identified, we found that some apps are likely to be affected. Specifically, the Origin Validation (see Section 4.3) vulnerability could potentially impact 133, while the Navigational Behavior (see Section 4.4) vulnerability could potentially impact a further 282 apps. Despite our extensive research, we have not identified any apps that could potentially be affected by the HTTP Authentication (see Section 4.5) and Custom Navigational Boundary (see Section 4.6) vulnerabilities.

The consequences range from information leakage (Origin Validation, HTTP Authentication) to phishing (Navigational Behavior, Custom Navigation Boundary).

It is important to note that our ability to assess the impact entirely is hampered by the lack of advanced tools to comprehensively analyze modern XMFs. We, therefore, hypothesize that a larger pool of apps could potentially be affected than our current assessment suggests.

6.2 Future Work

The following briefly lists some ideas to extend our work further.

- The security of other components within XMFs deserves more scrutiny. Our research focused on WebViews, but further investigation into other aspects of these frameworks, including data storage, network communication, and permissions management, may reveal additional vulnerabilities.
- The purpose of XMFs is to build cross-platform apps from a single codebase. Our research focused on the Android implementation of XMFs, but further investigation on other platforms such as iOS (and desktop) may determine the applicability of the vulnerabilities we found, and even identify additional ones.
- React Native apps are developed in JavaScript, and Flutter apps in Dart. Subsequently, this code is compiled into a custom, platform-independent bytecode and executed by either the Hermes or Dart VM. This process differs significantly from traditional Android app development and therefore limits the effectiveness of conventional tools such as AndroGuard [Ant23], Apktool [Con23], or JADX [Sky23]. Accordingly, we consider the development of practical analysis tools for XMFs as a task for future work.
- React Native developers write web code and naturally assume that web code is *sandboxed*. However, in React Native, this is not necessarily the case, and although Android has the App Sandbox, errors in the app code have more severe consequences than in ordinary web apps. For example, an error in the JavaScript code in a web app will have consequences like XSS in React Native but may lead directly to hardware access. Accordingly, we recognize finding vulnerabilities in the app code of React Native apps as a task for future work.

List of Figures

List of Tables

Acronyms

ACE arbitrary code execution. 10, 22

API application programming interface. 1, 3, 8–11, 15, 18, 20, 24, 30–32, 35, 36, 40, 57, 61, 63

APK Android Package. 7, 44

CSS Cascading Style Sheets. 13, 14, 20

CVE Common Vulnerabilities and Exposures. 10

FN false negative. 54, 55

FP false positive. 55

GIL global interpreter lock. 44

HTML Hypertext Markup Language. 13, 14, 20

HTTP Hypertext Transfer Protocol. ix, xi, 3, 9, 27, 35–37, 40, 41, 50, 52, 58, 59

I/O input/output. 18

ID identifier. 47

IDE integrated development environment. 7

IoT Internet of Things. 11

JNI Java Native Interface. 24

JSON JavaScript Object Notation. 45

OS operating system. 3, 5, 8, 27

PITM person-in-the-middle. 8, 30

PoC proof of concept. 3, 7, 32–38

SDK software development kit. 11, 24, 63

SOP same-origin policy. 11

UI user interface. 5, 8, 9, 20, 32, 39, 40

URI Uniform Resource Identifier. 50, 52, 61

URL Uniform Resource Locator. 5, 9, 15, 24, 27–29, 32, 37, 40, 44, 45, 50, 54

UXSS universal cross-site scripting. 32

VM virtual machine. 16, 22, 44, 59

XMF cross-platform mobile framework. ix, xi, 1–5, 8–11, 13, 14, 20–22, 24, 30, 32, 34, 40, 41, 43–49, 54, 57–59, 61, 63

XSS cross-site scripting. 32, 39, 59

Bibliography

[All+16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. „AndroZoo: Collecting Millions of Android Apps for the Research Community". In: *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2016, pp. 468–471. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903508.

[And12] Android Developers. *Using setJavaScriptEnabled*. 2012. URL: https://googlesamples.github.io/android-custom-lint-rules/checks/SetJavaScriptEnabled.md.html (visited on 07/30/2023).

[And23a] Android Developers. *Meet Android Studio*. 2023. URL: https://developer.android.com/studio/intro (visited on 09/10/2023).

[And23b] Android Developers. *WebChromeClient*. 2023. URL: https://developer.android.com/reference/android/webkit/WebChromeClient (visited on 06/14/2023).

[And23c] Android Developers. *WebSettings*. 2023. URL: https://developer.android.com/reference/android/webkit/WebSettings (visited on 06/12/2023).

[And23d] Android Developers. *WebView*. 2023. URL: https://developer.android.com/reference/android/webkit/WebView (visited on 05/06/2023).

[And23e] Android Developers. *WebViewClient*. 2023. URL: https://developer.android.com/reference/android/webkit/WebViewClient (visited on 06/13/2023).

[Ant23] Anthony Desnos. *androguard/androguard: Reverse engineering and pentesting for Android applications*. 2023. URL: https://github.com/androguard/androguard/tree/bd85695 (visited on 07/24/2023).

[AS17] Abeer AlJarrah and Mohamed Shehab. „The Demon is in the Configuration: Revisiting Hybrid Mobile Apps Configuration Model". In: *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*. Reggio Calabria Italy: ACM, Aug. 29, 2017, pp. 1–10. ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3105825. (Visited on 03/27/2023).

[Bee+22] Philipp Beer, Lorenzo Veronese, Marco Squarcina, and Martina Lindorfer. „The Bridge between Web Applications and Mobile Platforms is Still Broken". In: *2022 IEEE Security and Privacy Workshops (SPW)*. San Francisco, CA, USA: IEEE, May 2022. ISBN: 978-1-66549-643-8. URL: https://secweb.work/papers/2022/beer2022brokenbridge.pdf (visited on 05/29/2023).

[BJM09] Adam Barth, Collin Jackson, and John C. Mitchell. „Securing Frame Communication in Browsers". In: *Communications of The Acm* 52.6 (June 2009), pp. 83–91. ISSN: 0001-0782. DOI: 10.1145/1516046.1516066.

[Con23] Connor Tumbleson. *Apktool*. 2023. URL: https://apktool.org/ (visited on 07/24/2023).

[CW14] Erika Chin and David Wagner. „Bifocals: Analyzing WebView Vulnerabilities in Android Applications". In: *Information Security Applications*. Ed. by Yongdae Kim, Heejo Lee, and Adrian Perrig. Vol. 8267. Cham: Springer International Publishing, 2014, pp. 138–159. ISBN: 978-3-319-05148-2 978-3-319-05149-9. DOI: 10.1007/978-3-319-05149-9_9.

[DGB23] Anthony Desnos, Geoffroy Gueguen, and Sebastian Bachmann. *Bulk Analysis / Androguard 3.4.0 documentation*. 2023. URL: https://androguard.readthedocs.io/en/latest/intro/bulk.html (visited on 06/29/2023).

[GJS14] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. „Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks". In: *Proceedings 2014 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA: Internet Society, 2014. ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23323.

[Goo23a] Google LLC. *Flutter*. Build apps for any screen. 2023. URL: https://flutter.dev/ (visited on 05/02/2023).

[Goo23b] Google LLC. *Flutter architectural overview | Flutter*. 2023. URL: https://docs.flutter.dev/resources/architectural-overview (visited on 06/16/2023).

[Goo23c] Google LLC. *Security | Flutter*. 2023. URL: https://docs.flutter.dev/security (visited on 09/10/2023).

[Goo23d] Google LLC. *webview_flutter*. 2023. URL: https://pub.dev/packages/webview_flutter (visited on 05/06/2023).

[HMM17] Vincent Haupert, Dominik Maier, and Tilo Müller. „Paying the Price for Disruption: How a FinTech Allowed Account Takeover". In: *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium (ROOTS)*. Vienna Austria: ACM, Nov. 16, 2017, pp. 1–10. ISBN: 978-1-4503-5321-2. DOI: 10.1145/3150376.3150383. (Visited on 05/31/2023).

[Hu+18] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. „A tale of two cities: how WebView induces bugs to Android applications". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. Montpellier France: ACM, Sept. 3, 2018, pp. 702–713. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238180.

[Imm23] Immunix. *AppArmor*. 2023. URL: https://apparmor.net/ (visited on 09/20/2023).

[Jup23] Jupyter Team. *The Jupyter Notebook*. 2023. URL: https://jupyter-notebook.readthedocs.io/en/latest/notebook.html (visited on 09/10/2023).

[Kan23] Ananthu P Kanive. *LonelyCpp/react-native-youtube-iframe: A wrapper of the Youtube-iframe API built for react native*. 2023. URL: https://github.com/LonelyCpp/react-native-youtube-iframe/tree/d7b0c5a (visited on 07/23/2023).

[KLK22] Anssi Kostiainen, Mounir Lamouri, and Raphael Kubo da Costa. *Battery Status API*. W3C, 2022. URL: https://www.w3.org/TR/2022/WD-battery-status-20220203/ (visited on 05/03/2023).

[Kra22] Felix Krause. *iOS Privacy: Announcing InAppBrowser.com - see what JavaScript commands get injected through an in-app browser*. Aug. 18, 2022. URL: https://krausefx.com/blog/announcing-inappbrowsercom-see-what-javascript-commands-get-executed-in-an-in-app-browser (visited on 03/22/2023).

[Lio22] Lionel Sujay Vailshery. *Cross-platform mobile frameworks used by global developers 2021*. Feb. 21, 2022. URL: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ (visited on 03/14/2023).

[Lor23] Lorenzo Pichilli. *flutter_inappwebview*. 2023. URL: https://pub.dev/packages/flutter_inappwebview (visited on 05/06/2023).

[Luo+11] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. „Attacks on WebView in the Android System". In: *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 343–352. ISBN: 978-1-4503-0672-0. DOI: 10.1145/2076732.2076781.

[Mal+15] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. „Hybrid Mobile Apps in the Google Play Store: An Exploratory Investigation". In: *2015 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Florence, Italy: IEEE, May 2015, pp. 56–59. ISBN: 978-0-7695-5566-9. DOI: 10.1109/MobileSoft.2015.15.

[MDN23a] MDN contributors. *Mixed content - Security on the web*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content (visited on 07/21/2023).

[MDN23b] MDN contributors. *Window: postMessage() method - Web APIs*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage (visited on 06/09/2023).

[Met23a] Meta Platforms, Inc. *Core Components and Native Components | React Native*. 2023. URL: https://reactnative.dev/docs/intro-react-native-components (visited on 05/18/2023).

[Met23b] Meta Platforms, Inc. *Cross Platform Implementation | React Native*. 2023. URL: https://reactnative.dev/architecture/xplat-implementation (visited on 05/05/2023).

[Met23c] Meta Platforms, Inc. *React Native*. Learn once, write everywhere. 2023. URL: https://reactnative.dev/ (visited on 05/02/2023).

[Met23d] Meta Platforms, Inc. *react-native-website/website/versioned_docs/version-0.63/webview.md at cb1de5d*. 2023. URL: https://github.com/facebook/react-native-website/blob/cb1de5d/website/versioned_docs/version-0.63/webview.md?plain=1#L50 (visited on 05/05/2023).

[MS21] Abhinav Mohanty and Meera Sridhar. „HybriDiagnostics: Evaluating Security Issues in Hybrid SmartHome Companion Apps". In: *2021 IEEE Security and Privacy Workshops (SPW)*. San Francisco, CA, USA: IEEE, May 2021, pp. 228–234. ISBN: 978-1-66543-732-5. DOI: 10.1109/SPW53761.2021.00037.

[Mut+15] Patrick Mutchler, Adam Doupe, John Mitchell, Chris Kruegel, and Giovanni Vigna. „A Large-Scale Study of Mobile Web App Security". In: *2015 IEEE Security and Privacy Workshops (SPW)*. May 21, 2015. URL: https://www.ieee-security.org/TC/SPW2015/MoST/papers/s2p3.pdf (visited on 05/29/2023).

[NLP13] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. „A View to a Kill: WebView Exploitation". In: *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 13)*. Washington, D.C.: USENIX Association, Aug. 2013. URL: https://www.usenix.org/conference/leet13/workshop-program/presentation/neugschwandtner.

[Ost23] Ostorlab Team. *Flutter Reverse Engineering and Security Analysis | Ostorlab: Mobile App Security Testing for Android and iOS*. 2023. URL: https://blog.ostorlab.co/flutter-reverse-engineering-security-analysis.html (visited on 07/04/2023).

[Pan23] Pandas. *Pandas documentation*. 2023. URL: https://pandas.pydata.org/docs/ (visited on 07/18/2023).

[Pyt23] Python Software Foundation. *threading — Thread-based parallelism — Python 3.11.4 documentation*. 2023. URL: https://docs.python.org/3/library/threading.html (visited on 06/29/2023).

[Ras+16] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. „Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces“. In: *Proceedings 2016 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA: Internet Society, 2016. ISBN: 978-1-891562-41-9. DOI: 10.14722/ndss.2016.23234.

[REA23] REAndroid. *REAndroid/APKEditor: Powerful android apk editor - aapt/aapt2 independent*. 2023. URL: https://github.com/REAndroid/APKEditor/tree/e20b6b9 (visited on 06/29/2023).

[Rea23a] React Native Community. *react-native-webview/Guide.md at a6c8d28*. 2023. URL: https://github.com/react-native-webview/react-native-webview/blob/a6c8d28/docs/Guide.md (visited on 05/05/2023).

[Rea23b] React Native Community. *react-native-webview/react-native-webview: React Native Cross-Platform WebView*. 2023. URL: https://github.com/react-native-webview/react-native-webview/tree/681aac4 (visited on 05/07/2023).

[Rea23c] React Native Community. *react-native-webview/Reference.md at a6c8d28*. 2023. URL: https://github.com/react-native-webview/react-native-webview/blob/a6c8d28/docs/Reference.md (visited on 05/05/2023).

[Res15] Julian Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. RFC Editor, Sept. 2015. DOI: 10.17487/RFC7617.

[Sky23] Skylot. *skylot/jadx: Dex to Java decompiler*. 2023. URL: https://github.com/skylot/jadx/tree/5aaceee (visited on 05/06/2023).

[Sta23] StatCounter. *Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 4th quarter 2022*. 2023. URL: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/ (visited on 08/26/2023).

[The23a] The Apache Software Foundation. *Apache Cordova*. 2023. URL: https://cordova.apache.org/ (visited on 05/02/2023).

[The23b] The Apache Software Foundation. *Architectural overview of Cordova platform | Apache Cordova*. 2023. URL: https://cordova.apache.org/docs/en/11.x/guide/overview/index.html (visited on 05/03/2023).

[The23c] The Apache Software Foundation. *Battery Status | Apache Cordova*. 2023. URL: https://cordova.apache.org/docs/en/11.x/reference/cordova-plugin-battery-status/index.html (visited on 05/03/2023).

[Tim23] Tim Perry. *HTTP Toolkit*. 2023. URL: https://httptoolkit.com/ (visited on 07/18/2023).

[Wan+13] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. „Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 635–646. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516727.

[Way18] Wayback Machine. *WebView*. Dec. 3, 2018. URL: https://web.archive.org/web/20181203075801/https://developer.android.com/reference/android/webkit/WebView (visited on 07/21/2023).

[Way23] Wayback Machine. *WebView*. June 1, 2023. URL: https://web.archive.org/web/20230601161848/https://developer.android.com/reference/android/webkit/WebView (visited on 07/21/2023).

[Wit13] WithSecure™ Labs. *WebView addJavascriptInterface Remote Code Execution*. Sept. 23, 2013. URL: https://labs.withsecure.com/publications/webview-addjavascriptinterface-remote-code-execution (visited on 06/20/2023).

[Xia+22] Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. „Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Los Angeles CA USA: ACM, Nov. 7, 2022, pp. 2975–2988. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3559340.

[Yan+17] Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu. „Precisely and Scalably Vetting JavaScript Bridge in Android Hybrid Apps". In: *Research in Attacks, Intrusions, and Defenses (RAID)*. Ed. by Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis. Cham: Springer International Publishing, 2017, pp. 143–166. ISBN: 978-3-319-66332-6. DOI: 10.1007/978-3-319-66332-6_7.

[Yan+18] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. „Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications". In: *2018 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA: IEEE, May 2018, pp. 742–755. ISBN: 978-1-5386-4353-2. DOI: 10.1109/SP.2018.00043.

[YHG18] Guangliang Yang, Jeff Huang, and Guofei Gu. „Automated Generation of Event-Oriented Exploits in Android Hybrid Apps". In: *Proceedings 2018 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA: Internet Society, 2018. ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23236.

[YHG19] Guangliang Yang, Jeff Huang, and Guofei Gu. „Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 977–994. ISBN:

978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/yang-guangliang (visited on 03/29/2023).

[Zha+21] Zicheng Zhang, Daoyuan Wu, Li Lixiang, and Debin Gao. „On the Usability (In)Security of In-App Browsing Interfaces in Mobile Apps“. In: *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. San Sebastian Spain: ACM, Oct. 6, 2021, pp. 386–398. ISBN: 978-1-4503-9058-3. DOI: 10.1145/3471621.3471625.

[Zha+22] Lei Zhang, Zhibo Zhang, Ancong Liu, Yinzhi Cao, Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang. „Identity Confusion in WebView-based Mobile App-in-app Ecosystems“. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1597–1613. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-lei (visited on 03/29/2023).