Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications

Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza
Texas A&M University
{ygl, jeffhuang, guofei, abmendoza}@tamu.edu

***Abstract*—postMessage is popular in HTML5 based web apps to allow the communication between different origins. With the increasing popularity of the embedded browser (i.e., WebView) in mobile apps (i.e., hybrid apps), postMessage has found utility in these apps. However, different from web apps, hybrid apps have a unique requirement that their native code (e.g., Java for Android) also needs to exchange messages with web code loaded in WebView. To bridge the gap, developers typically extend postMessage by treating the native context as a new frame, and allowing the communication between the new frame and the web frames. We term such extended postMessage *"hybrid postMessage"* in this paper. We find that hybrid postMessage introduces new critical security flaws: all origin information of a message is not respected or even lost during the message delivery in hybrid postMessage. If adversaries inject malicious code into WebView, the malicious code may leverage the flaws to passively monitor messages that may contain sensitive information, or actively send messages to arbitrary message receivers and access their internal functionalities and data. We term the novel security issue caused by hybrid postMessage *"Origin Stripping Vulnerability"* (OSV).**

In this paper, our contributions are fourfold. First, we conduct the first systematic study on OSV. Second, we propose a lightweight detection tool against OSV, called *OSV-Hunter*. Third, we evaluate OSV-Hunter using a set of popular apps. We found that 74 apps implemented hybrid postMessage, and all these apps suffered from OSV, which might be exploited by adversaries to perform remote real-time microphone monitoring, data race, internal data manipulation, denial of service (DoS) attacks and so on. Several popular development frameworks, libraries (such as the Facebook React Native framework, and the Google cloud print library) and apps (such as Adobe Reader and WPS office) are impacted. Lastly, to mitigate OSV from the root, we design and implement three new postMessage APIs, called *OSV-Free*. Our evaluation shows that OSV-Free is secure and fast, and it is generic and resilient to the notorious Android fragmentation problem. We also demonstrate that OSV-Free is easy to use, by applying OSV-Free to harden the complex "Facebook React Native" framework. OSV-Free is open source, and its source code and more implementation and evaluation details are available online.

I. Introduction

Cross-origin communication using the HTML5 postMessage facility [1] has been a popular and often necessary technique on the web platform. It relaxes the restrictions enforced by the well-known same origin policy (SOP) security model [2] by allowing bidirectional messaging between mutually distrusting web frames or windows. With the increasing amalgamation of the web and mobile platforms, postMessage has also found utility on the mobile platform, as exhibited by the popularity of the embedded browser (i.e., *WebView*) in mobile apps (i.e., hybrid apps) [3].

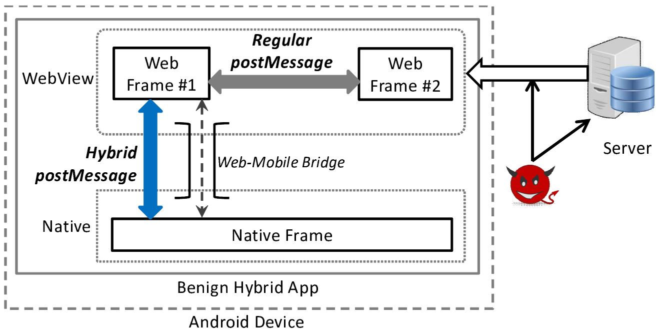

Figure 1: Overview of regular and hybrid postMessage

In addition to cross-origin communication, the hybrid mobile app model introduces the necessity for *cross-platform* communication between the web platform and the mobile platform. Not only do hybrid apps need to communicate between different origins loaded in a WebView, they must also facilitate communication between those origins and the native layer (e.g., the Android Java code). While hybrid apps can already utilize web-mobile bridges (such as the JavaScript Bridge) [4] for cross-platform execution, cross-platform messaging in the form of HTML5 postMessage is not available.

Android 6.0 partially addresses this shortcoming by providing a new cross-platform API called postWebMessage(). However, this API is plagued by the notorious Android fragmentation problem [5] and does not scale well. Moreover, it is limited to unidirectional communication from native to web but does not support communication from web to native. In our empirical study on a set of popular hybrid apps, we found postWebMessage() was rarely used in practice.

As a result, developers have resorted to customizing postMessage in hybrid apps using ad-hoc methods such as web-mobile bridges (see Figure 1). In general, this customization treats the native context as a new different-origin frame. This results in "*hybrid postMessage*", which provides both native-to-web ($N{\rightarrow}W$) and web-to-native ($W{\rightarrow}N$) messaging.

Security Issue. Unfortunately, while hybrid postMessage provides easy and convenient cross-platform communication, it

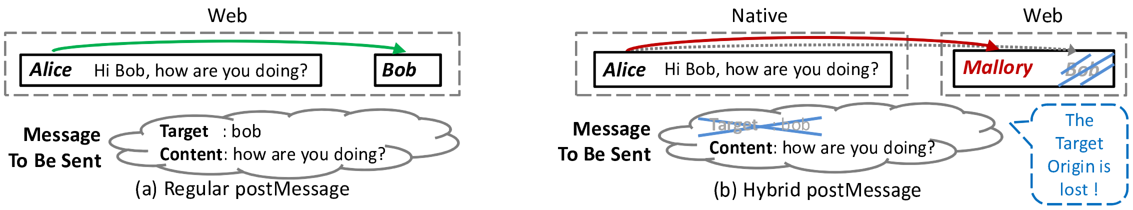
Web
Alice Hi Bob, how are you doing?
Bob
Message To Be Sent
Target : bob
Content: how are you doing?
(a) Regular postMessage
Native
Web
Alice Hi Bob, how are you doing?
Mallory
~~Bob~~
~~Target : bob~~
Content: how are you doing?
The Target Origin is lost !
(b) Hybrid postMessage

Figure 2: Sending Messages Through Regular And Hybrid postMessage

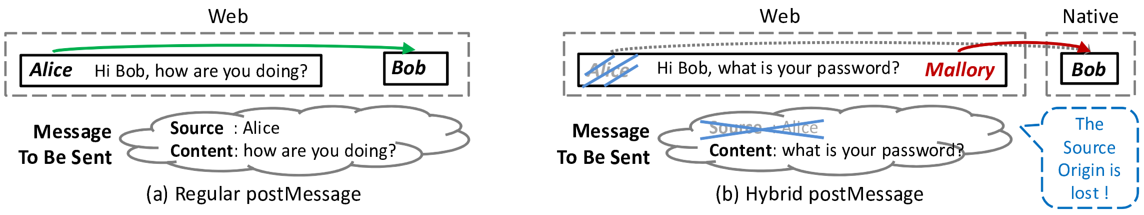
Web
Alice Hi Bob, how are you doing?
Bob
Message To Be Sent
Source : Alice
Content: how are you doing?
(a) Regular postMessage
Web
Native
~~Alice~~
Hi Bob, what is your password?
Mallory
Bob
~~Source : Alice~~
Content: what is your password?
The Source Origin is lost !
(b) Hybrid postMessage

Figure 3: Receiving Messages Through Regular And Hybrid postMessage

also opens a door for adversaries through code injection attacks (such as web or network attacks shown in Figure 1) to launch denial-of-service (DoS) attacks, steal sensitive information, silently access local hardware (such as the microphone), and perform other nefarious actions. The security problem is rooted in the loss of the origin information when messages move across the web and native layers. More specifically, the origin information of the message sender (*source*) and message receiver (*target*) is either not respected or totally lost. There are two main reasons: 1) Hybrid postMessage may not provide any interface to allow the message sender to specify the ***target origin***, which is critical in the regular HTML5 postMessage to control the message receiver; 2) Hybrid postMessage may not provide the ***source origin*** of a received message, which means it is impossible for the message receiver to validate the message. This adds a new layer to the known security problem of client-side validation (CSV) in the web platform [6] [7] [8]. For convenience, we term the novel security issue caused by hybrid postMessage *"Origin Stripping Vulnerability"* (*OSV*).

Figures 2-3 illustrate that OSV may compromise the confidentiality and integrity of cross-platform communication. Consider that adversaries inject malicious code into WebView through web or network attacks. The malicious code may leverage hybrid postMessage to passively receive and monitor messages that contain sensitive information, or actively send messages to arbitrary message receivers to access their internal functionalities or data.

In Figure 2-a, Alice sends a message to Bob through the regular postMessage. The message contains the message content ("How are you doing?"), and the target origin (Bob), which determines that only Bob can receive the message. However, hybrid postMessage breaks this convention by stripping the target origin (Figure 2-b). As a result, Mallory, an adversary who runs malicious code in another web frame can receive and read the message. If the message carries sensitive information, Mallory can easily violate the confidentiality of Alice and Bob's communication. In Figure 3-a, Bob is receiving a message from Alice. When the message arrives, Bob can validate that the source origin of the message is Alice. However, hybrid postMessage loses the source origin information (Figure 3-b), which means that it is impossible for Bob to conduct validation. Therefore, Mallory may send a message ("What's your password?") to Bob and access its confidential data.

The Root Cause of OSV. Although the detailed implementation guideline and security model for postMessage are established in HTML5 [1], it is challenging for developers to implement hybrid postMessage conforming to it. The main obstacle is the gap between the web and native platforms. Web-mobile bridges may be applied to fill the gap. However, as shown in prior work [4] [9] [10], these bridges are often the cause of security vulnerabilities, because *any* code loaded in WebView may freely access them.

For example, we found hybrid postMessage was implemented in the popular "Facebook React Native" framework using the JavaScript Bridge. As shown in Listing 1, the crucial JavaScript method *window.postMessage()* is rewritten to allow all messages to be sent to the native frame. However, due to the intrinsic weakness of the JavaScript Bridge, the native frame cannot distinguish the identity of the message senders, or even safely obtain the source origin.

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

Listing 1: Implementing $W \to N$ In Facebook React Native

State-Of-The-Art WebView Defense Solutions. Existing defense solutions, such as NoFrak [4], Draco [9], MobileIFC [11], WIREframe [12], and HybridGuard [13], were designed to provide protection for WebView and web-mobile bridges by either extending SOP to the native layer, or enforcing security policies to offer access control. However, they are circumscribed to prevent OSV for several reasons. First, most

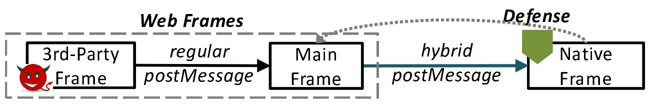

Figure 4: Communication Among Three Frames

existing defense solutions can only protect $W \rightarrow N$, but not $N \rightarrow W$. Only WIREframe can offer protection in two directions. However, unfortunately, its security policies enforced in $N \rightarrow W$ may be under the control of adversaries. Second, existing defense solutions are coarse-grained, and may have high false negatives. Their provided protection is usually performed based on the origins of web frames, and thus it is difficult for them to limit the behaviors of the embedded JavaScript code.

Moreover, existing defense solutions may be hindered by the blend of OSV and CSV vulnerabilities. Consider a scenario in Figure 4 which we found in a real-world advertisement library. In the web platform, a nested third-party iframe can send messages to the main frame, where a message handler receives the messages but does not validate their source origins (i.e., CSV vulnerability). It then forwards the received messages to the native frame through hybrid postMessage. After that, the defense solutions are enforced to protect $W \rightarrow N$. They attempt to obtain the message sender's origin to apply their policies. However, they can only obtain is the main frame's origin, rather than the real message sender's origin (i.e., the third-party frame's).

CSV detection and defense solutions [6] [7] [8] may be applied to mitigate the above threat. However, their performance may also be limited. They rely on the analysis or detection of source origins of received messages. The messages received by the message handler of the main frame include not only messages ("M_1") from the third-party frame, but also messages ("M_2") from the native frame. They may protect M_1, but not M_2, because the source origin of M_2 may not be provided in hybrid postMessage.

Contributions. In this paper, our contributions are four-fold. First, we conduct the first systematic study on hybrid postMessage and identify the novel security issue "OSV". Second, to evaluate the prevalence and presence of hybrid postMessage and OSV in Android hybrid apps, we design a lightweight detection tool, called *OSV-Hunter*, that can help developers and analysts identify hybrid postMessage and discover potential OSVs. Different from existing detection tools [10], [14], which fall short of filling the web-mobile gap and tracking origins, OSV-Hunter automatically discovers message senders and receivers, and analyzes the semantics of the link between them.

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

We have reported all our findings to the Android security team, and the relevant framework, library, or app developers. We are actively helping them fix the discovered OSV problem. The Facebook security team has confirmed our findings in the React Native development framework, and they also admitted that it was difficult to eliminate the security problem caused by OSV in their current implementation. Instead, they explicitly added a security warning in their development documentation [15].

Lastly, motivated by the above difficulty faced by developers to eliminate OSV, we design and implement a set of new hybrid postMessage APIs in the newest WebView, called *OSV-Free*. Our evaluation shows that OSV-Free is secure and fast, and it is generic and resilient to the notorious Android fragmentation problem. We also demonstrate that OSV-Free is easy to use, by applying OSV-Free to harden the complex "Facebook React Native" framework. OSV-Free is open source, and its source code and more implementation and evaluation details are available online: http://success.cse.tamu.edu/lab/osv-free.php.

Paper Organization. The rest of the paper is organized as follows. We first introduce the necessary background and the threat model and define the OSV problem (Section II). Next, we present the design and implementation details of our detection tool OSV-Hunter (Section III). Then, we show our study results about hybrid postMessage and OSV (Section IV). After that, we present the design and evaluation of our mitigation solution OSV-Free (Section V). Last, we present related work (Section VI) and discussion (Section VII), and conclude in Section VIII.

II. Background and Problem Statement

A. Background: postMessage and WebView

```
1 // Send a message
2 window.postMessage(m, t)
3
4 // Enable the first message handler
5 function message_handler(e) { ... }
6 window.addEventListener("message", message_handler, false
      )
7
8 // Enable the second message handler
9 onmessage = function (e) { ... }
```

Listing 2: Usage of postMessage

postMessage. postMessage is frequently used to exchange data between different origins in HTML5-enabled web applications. Listing 2 presents the basic usage of postMessage. In Line 2, *window.postMessage()* is called to send the message content *m* to the target origin *t*. From Line 4 to Line 9, two message handlers are enabled in two different manners : 1) calling the method *addEventListener()* to register the message handler '*message_handler()*' (Line 6); 2) or rewriting the global object *onmessage* to enable an anonymous message handler (Line 9). Please note that when a message arrives, both these two message handlers will be called to handle it.

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

The message handler (receiver) is responsible for validating the source origin to ensure the message is from a trusted origin. This requirement is deferred to the message handler implementation and not enforced by the OS or framework. The absence of such validation will cause the client-side validation vulnerability (i.e., CSV), which is well studied by existing work [6]–[8].

WebView. WebView is an embedded UI component used to render web pages and run JavaScript code within mobile apps. For this purpose, WebView provides APIs to directly load web content or run JavaScript in WebView, such as *loadUrl()*. Please note that if the API parameter is JavaScript code, the code will be executed in the *main* web frame.

WebView is powerful and customizable. WebView can specify event handlers to handle web events that occur in WebView. For example, *shouldInterceptRequest()* can handle the content loading event.

The Official Hybrid postMessage APIs in WebView. In Android 6.0, cross-document APIs (such as *"WebView.postWebMessage()"*) and channel messaging APIs (such as *"WebView.createWebMessageChannel()"*) [16] are added. However, both suffer from the Android fragmentation problem [5]. Based on the new Android version distribution data [17] (Nov. 2017), almost 42% of Android devices do not support these official APIs. Furthermore, compared with *postWebMessage()*, *createWebMessageChannel()* can allow bidirectional communication. However, in our empirical study, we found channel messaging was heavy, and rarely implemented and used in hybrid postMessage.

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

However, WebView does not provide any access control on *JavaScript Bridge*. Any JavaScript code loaded in WebView can easily access it without any limitations. This has been well studied by existing work [4] [9] [10].

Several defense solutions [4] [9] have been proposed to protect JavaScript Bridge, and cure its intrinsic weakness. However, as discussed in Section I, if JavaScript Bridge is applied in the hybrid postMessage implementation, existing defense solutions cannot defend against attacks.

B. Threat Model

In this paper, we focus on hybrid-postMessage enabled Android hybrid apps. We assume the native code is benign, and the content loaded in WebView may be untrusted. We consider the following two scenarios.

- *Web Attacks*: Adversaries control several domains and web servers. When these servers are accessed, adversaries can inject malicious code. However, adversaries do not have capabilities to monitor the communication between apps and other domains or servers that do not belong to adversaries. Generally, we assume the content from the first-party server is trusted, while content from third-party servers may be malicious or harmful.
- *Network Attacks*: Adversaries can hijack unsafe connections (such as communication over HTTP) through man-in-the-middle attacks (MITM). These are common in some practical scenarios such as public WiFi access.

C. The OSV Problem Definition

We define OSV based on the possible violation on postMessage's security model (or design guideline) [1], which is defined as follows. We assume *SF* and *RF* are the frames which a message sender and its corresponding message receiver belong to respectively. The security model can be defined using the following two rules.

- *Rule I*: When a message is being sent, its target origin T_{origin} should satisfy that 1) T_{origin} is specified or implied; 2) $T_{origin} = RF_{origin}$ or $T_{origin} =$ "*".
- *Rule II*: When a message is being received, its source origin S_{origin} should meet that 1) S_{origin} is defined; 2) $S_{origin} = SF_{origin}$; 3) S_{origin} is unique for *SF*.

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

Direction	Native $\rightarrow$ Web		Web $\rightarrow$ Native	
Violated Rule	Rule I	Rule II	Rule I	Rule II
Sub-Vulnerability Type	V_1	V_2	V_3	V_4

Table I: Definitions of Four Sub-Types of OSV

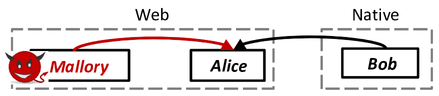

Figure 5: Attacks On V_2

The four OSV sub-vulnerabilities disclose more attack patterns than those discussed in Section I. For example, consider a scenario in Figure 5. Alice and Mallory are web frames, while Bob is a native frame. Bob sends messages to Alice through hybrid postMessage. Due to V_2, the source origin of the native frame may not be provided or not unique. Mallory may be able to forge a message with the same source origin, by creating a nested controllable iframe that has the same origin, and then sending a crafted message from the new iframe to Alice using the typical web postMessage. When Alice receives the message, Alice notices that the source origin is the same as the native frame's. As a result, Alice treats Mallory as Bob and allows Mallory to access the internal functionalities. If Alice carries critical functionalities or data, serious consequences may be caused.

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

III. OSV-Hunter Design and Implementation

A. Design observations

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

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

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

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

  To implement it, a special and unique string *ID*, such as *"PM_Case1_<Random Number>"* for the first case and *"PM_Case2_<Random Number>"* for the second case, is injected into c and tracked. More specifically, in the native frame, all native function invocations should be checked to verify if their parameters contain *ID*. If *ID* is found, there should be a link between *window.postMessage()* and the firstly found native function. For the second case, all message handlers of web frames should be monitored. Once *ID* appears in the message handlers of a web frame, there should also be a link from the native function that executes *window.postMessage()* through *WebView.loadUrl()* to the message handlers of the web frame.
- *A message handler of a web frame may be a message proxy, or receiver*: It is possible for a message handler to 1) receive messages from the native frame (i.e., $N \rightarrow W$), or 2) forward messages received from other web frames to the native frame (i.e., $W \rightarrow N$). The above possibilities can be verified respectively. For the first possibility, the value of the parameter of the message handler should be monitored to check if ID exists. For the second possibility, similar with how *window.postMessage()* is handled, the received message content of the message handler should be tracked. For this purpose, if no ID exists in the received message content, a new *ID*, such as *"MH_ForwadingMessage_<Random Number>"*, should be injected into the received message content. When the message content is forwarded, if the *ID* appears in a native function in the native frame, the native function is likely a message receiver. Hence, there may be a link between the message handler of the web frame and the native function of the native frame.
- *The APIs (such as web-mobile bridges) that provide cross-platform functionalities are likely utilized to implement hybrid postMessage*: For example, apps may execute JavaScript code to trigger a message event using the JavaScript execution APIs (like *WebView.loadUrl()*). Hence, the parameters of these APIs should be carefully handled. Additionally, *WebView.postWebMessage()* should also be monitored, since it can be used for $N \rightarrow W$ messaging.

B. Design Details

Guided by these observations, we designed two main phases in OSV-Hunter containing a number of sub-modules, as shown in Figure 6. In Phase#1, *"hybrid postMessage Identification"* fills the semantic gap between the native and web frames, and identifies the implementation of hybrid postMessage. In Phase#2, *"Message Origin Analysis"* collects all delivered messages between message senders and receivers, and performs origin analysis to determine the existence of OSV.

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

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

1) Hybrid postMessage Identification:

a) Tester: To trigger WebView and run native code (for triggering message senders in the native frame), we use a random UI explorer "Monkey" to simulate users' behaviors [18]. Once WebView is started, network activities may occur. Then, the pre-defined JavaScript fuzzing code is injected into network traffic based on our threat model (Section II-B), which is done using the popular proxy tool "mitmproxy" [19]. Please note that in order to perform network attacks, network links are crawled to check if a HTTP link can be navigated. For convenience, we limit the crawl depth as three.

The above injected JavaScript fuzzing code is designed to drive the test on $W \rightarrow N$. Usually, the JavaScript methods that send messages (e.g., *window.postMessage()*) are called in all

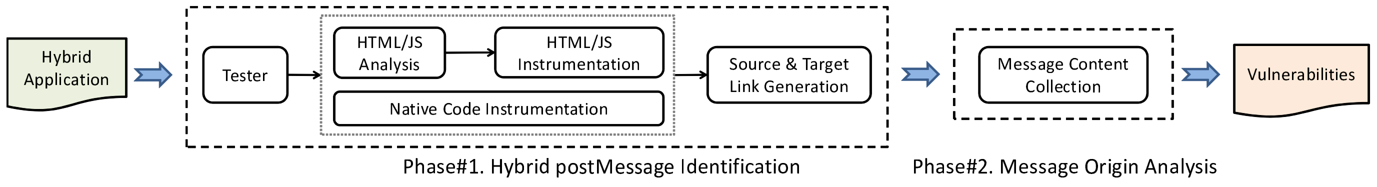
Hybrid Application
Tester
HTML/JS Analysis
HTML/JS Instrumentation
Native Code Instrumentation
Source & Target Link Generation
Message Content Collection
Vulnerabilities
Phase#1. Hybrid postMessage Identification
Phase#2. Message Origin Analysis

Figure 6: OSV-Hunter's Workflow

kinds of environments. It is implemented mainly based on existing work, such as the work of Schwenk et al. [20].

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

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

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

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

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

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

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

2) Message Origin Analysis:

a) Message Content Collection: To determine the existence of OSV, the content of all delivered messages are fully dumped and collected. In the native frame, the content of all related low level objects (e.g., StringObject) are dumped. In the web frames, the content of all JavaScript variables is printed. If a variable is an object, all its fields (including inherited fields), and the corresponding values are logged.

All other critical logs are also gathered, such as the ones containing origin information of message senders and receivers.

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

C. Implementation

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

IV. Study of Hybrid postMessage and OSV

A. Data Set

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

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

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

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

B. Results

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

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

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

Although we did not find a good counter-example to prove the origin "undefined" is wrong for the native frame (such as "undefined" may be not unique), "undefined" is meaningless and hard to manage. As discussed in Section II-C, such meaningless origins may cause more security issues, such as CSV. A similar problem is also found in *WebView.postWebMessage()*, which provides a meaningless origin (empty string) as the source origin. It is because in the native layer, the internal implementation of *postWebMessage()* does not explicitly define the origin of the native frame, and NULL is used at default. Correspondingly, in the web space, an empty string is treated as the source origin.

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

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

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

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

C. Findings

From our study results, we have the following findings.

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

Vulnerability Name (App or Framework)	*Impacted Apps / Total Apps*	*Example App*	Vulnerability Type				*Consequences*
			Native → Web		Web → Native		
			V_1	V_2	V_3	V_4	
Facebook React Native	43/43	com.altvr.xxx com.giantfood.xxx ...	✔	?	✗	✔	Monitoring Audio, Data Race, Internal Critical Data Manipulation, ...
Google Print	30/30	com.adobe.xxx cn.wps.xxx ...	▨	▨	✗	✔	Denial of Service
Eclipse Source	1/1	com.eclipsesource.xxx	✔	✔	✗	✔	Sending a message with a source origin not belonging to itself
WebView's postWebMessage()	0/0		✗	?	▨	▨	
Total Vulnerable Apps / Total Apps	74/74	Please note that ✔ means the sub-vulnerability exists; ✗ means the sub-vulnerability does not exist; **?** indicates there are no strong evidences to verify whether the sub-vulnerability exists or not. The cell marked with the grey color means the communication in that direction is not implemented.					

Table II: The Evaluation Result

loaded in WebView is fully trusted, which may cause serious consequences.

- *The requirement of a feasible hybrid postMessage implementation may be urgent*: Regular postMessage is still very popular in hybrid mobile apps. However, compared with regular postMessage, a feasible hybrid postMessage implementation is more preferred. For instance, in many apps, $W \rightarrow N$ is implemented by rewriting the JavaScript method *window.postMessage()*, which breaks the regular postMessage functionality.
- *In all web frames, only the main web frame usually has the capability to communicate with the native frame, but some main web frames are treated as message proxies during message delivery*: Within our data set, we found 73/74 (98.6%) apps only allow the main web frame to exchange data with the native frame, and 30/74 (40.5%) apps leverage the main web frame as proxies.
- *The blended vulnerabilities of CSV and OSV exist in real world apps*: 30 apps use the main web frame as message proxies, where both CSV and OSV exist. As discussed in Section I, the blended vulnerabilities may result in that existing WebView defense solutions may be fooled.
- *The official hybrid postMessage APIs are rarely used in practice*: Within our whole dataset, no apps use the official WebView APIs. Compared with developers' implementations, the functionality provided by *WebView.postWebMessage()* is too simple.
- *The communication "$W \rightarrow N$" is usually implemented relying on JavaScript Bridge*: JavaScript Bridge opens bridges linking web code with native code. However, as JavaScript Bridge usually does not carry any origin information, OSV is likely caused. Although there are several solutions proposed to protect JavaScript Bridge, all are limited in their ability to prevent OSV (Section I).

D. Case Studies

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

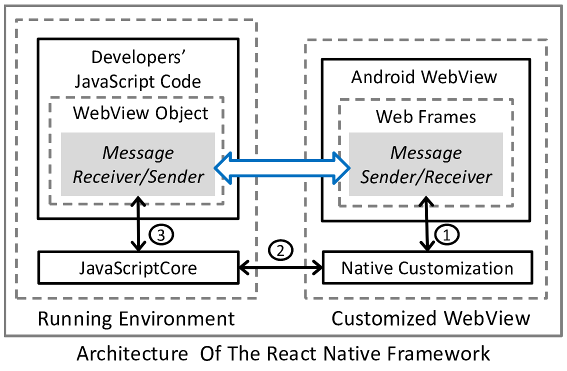

Figure 7: hybrid postMessage in Facebook React Native

The architecture of the React Native framework is shown in Figure 7. In run-time, the *running environment* is first created. Developers' JavaScript code "*DJ*" is parsed and executed by the embedded generic and powerful JavaScript engine "JavaScriptCore". Through JavaScriptCore, *DJ* can interact with Android, such as creating native UI components, and handling UI events.

WebView (i.e., *customized WebView in Figure 7*) is also available in the React Native framework. To enable it, it is required for *DJ* to create a WebView object *O* as the reference. Listing 3 illustrates how to create a WebView object in *DJ* (Line 9), and let WebView to show a remote web page (Line 13).

```
 1 // A message handler
 2 handleMessage(e) {
 3     // The message content is saved in e.nativeEvent.data
 4     // However, the source origin is lost.
 5     this.webview.postMessage("[native] received a message
         : " + e.nativeEvent.data);
 6 }
 7 // Configure UI layout
 8 render() {
 9     return (<WebView // Create a WebView component 'O'
10         // Enable JavaScript
11         javaScriptEnabled={true}
12         // Load a remote web page in WebView
13         source={{uri: "https://developer.com"}}
14         // Register a message handler
15         onMessage={this.handleMessage}
16         .../>
17     );
18 }
```

Listing 3: Example Code of Creating A WebView Object in *DJ*

In the React Native framework, hybrid postMessage is implemented to allow the communication between O and the JavaScript code loaded in the native WebView component (for convenience, we denote the latter JavaScript code as "WJ"). For this purpose, two APIs are added in O : 1) *WebView.postMessage()* (Line 5 of Listing 3), which sends a message from O to the main web frame of WJ; and 2) *WebView.onMessage()* (Line 15 and Lines 2-6 of Listing 3), which receives messages from the main web frame of WJ.

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

Explanation. To support hybrid postMessage, the React Native framework customizes Android WebView, where the origin information is not carefully handled. More specifically, as shown in "Customized WebView" of Figure 7, when a message is sent from WJ, it first enters the native context (i.e., "Native Customization") through a pre-imported JavaScript Bridge, where the origin information is lost. Then, the message is delivered to the embedded JavaScript engine, and further forwarded to O.

The key implementation is shown in Listing 1, and partially discussed in Section I. In Customized WebView, the JavaScript method *window.postMessage()* is rewritten. So that when *window.postMessage()* is called in WJ, the message is redirected to a pre-defined native function in the JavaScript Bridge "__REACT_WEB_VIEW_BRIDGE". However, during the message delivery, the source origin information is lost.

To implement sending a message in the opposite direction, the code shown in Listing 4 is used. The message content to be sent is wrapped in a message event (Lines 3-6), and then is dispatched to message handlers in the main web frame (Line 12). Since the message origin is not defined in the event wrapper, "undefined" appears as the source origin. More importantly, the implementation cannot ensure the code is executed in the correct context (e.g., the target origin may not be right).

```
1  WebView.loadUrl("javascript:(function () {" +
2      "var event;" +
3      // Carrying message content in the customized data
           structure
4      "var data = {'data': " + message_content + "};" +
5      "try {" +
6          // Creating an event
7          "event = new MessageEvent('message', data);" +
8      "} catch (e) { ... }" +
9      // Sending the event to message handlers of the main
           web frame
10     "document.dispatchEvent(event);" +
11 "})();")
```

Listing 4: Sending Messages To The Main Frame Through WebView.loadUrl() In The Native Context

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

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

```
1 window.postMessage(
2     '{' +
3         '"method":"enterSpaceForceVR",' +
4         '"args":{' +
5             '"Url":"<event_url>"' +
6         '}, ...' +
7     '}')
```

Listing 5: Example Attack Code To Let Apps Forcely Join Any Events

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

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

Furthermore, the above attack code may also cause data race. When the app is opened, the app usually takes a long time for initialization, especially when the microphone is enabled. At that period, if the attack code shown in Listing 5 is injected and executed, a data race occurs. In our test, the data race can be stably triggered. When a third-party JavaScript lib is fetched by the app's WebView, adversaries can immediately inject and run attack code. Then, the data race can be triggered. In addition, the influence of the data race is *continuous*, and can only be avoided by totally cleaning user data, or re-installing the app.

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

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

The implementation of the app's message receiver is shown in Listing 6. When a message is received, the message content is retrieved and parsed (Line 5). Then, the message receiver executes an arbitrary method whose name and arguments are determined by the fields "method" and "args" of the received message (Lines 9). Finally, the execution result "r" is returned through *WebView.postMessage()* (Line 13).

```
1 // e is a WebView object in O
2 // Registering a message handler
3 e.onMessage = function(t) {
    // Reading message content to a
    var a = JSON.parse(t.nativeEvent.data);
    ...

    // Executing an arbitrary method in the WebView
        object e
    r = e[a.method](a.args);
    ...

    // Returning the execution result to WJ
    e.refs.wv.postMessage(JSON.stringify({..., value: r
        , ...}));
}),
```

Listing 6: Code Snippet of onMessage()

- *Case#2 com.giantfood.xxx*: It is a food shopping management app. The operation on users' cart (i.e., the shopping list) relies on data exchange over hybrid postMessage. In $WJ \rightarrow O$, the main frame of WJ can send a command to ask for corresponding actions, such as opening and editing cart, and adding and removing items to or from the cart.
  Hence, a feasible attack scenario is that an attacker injects malicious code through an HTTP link, and then, sends messages through $WJ \rightarrow O$ to manipulate the app's internal data.
  The implementation of the message receiver of O is shown in Listing 7. When a message is received, its content is directly parsed and dispatched to the corresponding event handler. Hence, if the content of the transferred message is equal to the values in "SHOPPING_LIST", all internal functionalities can be accessed.

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

Listing 7: Code Snippet of onMessage()

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

```
1  public void onPageFinished(WebView view, String url) {
2      webView.loadUrl("javascript:" +
3          // Registering a message handler as message proxy
4          "window.addEventListener(" +
5              "'message'," +
6              // Forwarding all received message content to
                   the native frame
7              "function(evt) {" + // CSV exists
8                  " window." + JS_INTERFACE + ".
                       onPostMessage(evt.data)" +
9              "}, " +
10             "false" +
11         ")");
12 }
```

Listing 8: The Source Code of Registering A Message Handler In Google Print

Please note that although a JavaScript Bridge is used in the message handler of the main web frame, we still count the JavaScript Bridge as part of the implementation of hybrid postMessage. It is because in this scenario, the native function (*"onPostMessage()"*) of the JavaScript Bridge is the essential message receiver that handles the received message content. It is also reflected in its implementation, which is shown in Listing 9. In the native function, the message content is handled and parsed. If it is equal to a constant value, which is saved in the variable "CLOSE_POST_MESSAGE_NAME", the service will be finished.

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

Listing 9: Source Code of The Message Handler In Google Print

The above implementation of $W{\rightarrow}N$ suffers from V_4, since the source origin is lost. As a result, DoS may be caused, considering the following situations: 1) based on our URL crawler (Section III-B1a), the web page loaded in WebView contains an HTTP link, which may be leveraged to inject malicious code; 2) adversaries can leverage hybrid postMessage to send a special message to the native frame to stop the service. If the content of the sent message is equal to the value of the variable "CLOSE_POST_MESSAGE_NAME", DoS may be caused.

In addition, the message handler of the main frame is also a message proxy. However, CSV exists, which indicates that the scenario about the blended attacks on OSV and CSV is feasible (Figure 4).

V. The Mitigation Solution : OSV-Free APIs

A. Goals

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

- *Meeting the development requirements*: The new APIs should provide both $N{\rightarrow}W$ and $W{\rightarrow}N$ functionalities.
- *Secure*: The APIs should not be affected by OSV.
- *Fast*: The APIs should only introduce low overhead.
- *Easy to use*: The APIs should be easily applied and integrated.
- *Generic*: The APIs should be resilient to the notorious Android fragmentation problem, and support as many devices as possible.

B. Overview

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

API Context	*Role*	*API*	*Description*
Web	Message Sender	void postMessageToNativeFrame(String msg)	Sending *msg* to the native frame
Native	Message Sender	void postMessageToMainFrame(String msg, Uri targetOrigin)	Sending *msg* to the main web frame whose origin is targetOrigin
	Message Receiver	void receiveMessageFromMainFrame(Callback callback)	Registering a callback function to receive messages from the main web frame

Table III: OSV-Free APIs

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

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

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

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

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

Listing 10: The Prototype of onMessage

Listing 10 shows the prototype of the native callback function *"onMessage"*. When a message is received by the callback function, three levels of origin information is provided so that the callback function can perform validation in a fine-grained way, and also obtain hints about the whole picture of the message delivery process. More specifically, the first provided origin *"frameOrigin"* indicates the origin of main web frame; the second origin *"scriptOrigin"* provides the origin of the embedded script, where the JavaScript method that sends the message is located; the third variable flag *"isProxyInvolved"* indicates whether the main web frame is forwarding a message as a proxy. If the flag is true, the scenario similar to what is shown in Figure 4 is faced. Hence, developers should carefully handle this situation.

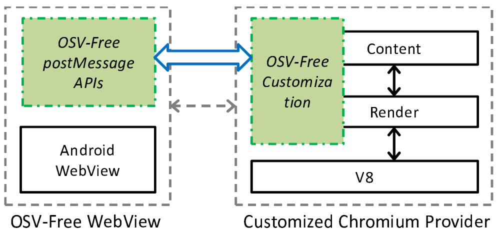

Figure 8: OSV-Free's Design

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

C. Design and Implementation

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

In Android, users can select a browser provider as the library. Currently, Chromium [21] is the default provider. Roughly, Chromium consists of three modules : 1) content, which links Android WebView with the render module together; 2) render, which is responsible to handle rendering tasks and interact with the JavaScript engine V8; 3) V8, which is a open-source JavaScript engine developed by Google.

OSV-Free's design is shown in Figure 8. OSV-Free mainly consists of two parts : *OSV-Free WebView* and *Customized Chromium Provider*. OSV-Free WebView is a WebView wrapper that declares the native APIs *postMessageToMainFrame()* and *receiveMessageFromMainFrame()*, while Customized Chromium Provider provides the essential implementations of the above two native APIs. For the remaining JavaScript method *postMessageToNativeFrame()*, Customized Chromium Provider can automatically enable it in the main web frame, when a callback function is registered through the native API *receiveMessageFromMainFrame()*. Please note that *OSV-Free*

WebView should be integrated into vulnerable apps to replace the original WebView.

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

- *postMessageToMainFrame()*: This API is implemented by reusing existing methods. When the API is called, the customized content module is started, and then an internal API, called *postMessageToFrame()*, is invoked to handle the whole task of the $N \to W$ message.
- *receiveMessageFromMainFrame() And postMessageToNativeFrame()*: *receiveMessageFromMainFrame()* is implemented by instrumenting the content and render modules. When the API is called, the content module is entered, where the API's parameter is cached, parsed, and checked to make sure the format is correct and its internal callback function is not empty. Then, a message is sent to the render module to notify that a callback function is being registered. After that, the render module reads the context of V8, and binds a pre-defined callback function f to V8 as "*postMessageToNativeFrame()*".
  In run-time, when *postMessageToNativeFrame()* is called in the main web frame, f follows. Then, in f, multiple level origin information is collected. The origin of the main web frame "frameOrigin" is obtained by identifying the mainframe object in the frame tree and retrieving the last-loaded URL from the mainframe object. It can be done by calling *"frame_tree()->GetMainFrame()->last_committed_url().GetOrigin().spec()"*. The origin of the nested script "ScriptOrigin" can be retrieved from the last node of the frame stack (i.e., v8::StackTrace::CurrentStackTrace()). The flag "isProxyInvolved" is configured by checking if a message handler is called, which is done by analyzing the above frame stack. Currently, only the global message handler "onmessage" is supported. We leave supporting other message handlers as our future work.
  Later, the render module packs all above origin information together with the message content and sends them to the content module. Finally, developers' callback function "Callback.onMessage()" is called with multiple level origin information and the message content.

D. Evaluation

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

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

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

Target Item	***APIs***	***Average Cost Time*** (milliseconds)
The official API ($N \to W$)	*postWebMessage()*	2.63
OSV-Free $N \to W$	*postMessageToMainFrame()*	2.23
OSV-Free $W \to N$	*postMessageToNativeFrame* $\to$ *receiveMessageFromMainFrame()*	2.08

Table IV: The Performance of OSV-Free APIs

2) Effectiveness: To check OSV-Free's effectiveness, we use OSV-Free to patch two vulnerable frameworks: the Facebook React Native framework and the Google Print lib. We found that the vulnerabilities could be eliminated. In $N \to W$, only the specified target origin can receive the message. When a message is received, its source origin is the native frame's origin. In $W \to N$, the target origin is implied in the function *postMessageToNativeFrame()*, while the source origin of the received message provides rich and correct origins.

3) Compatibility: To confirm OSV-Free's compatibility, we installed and successfully verified OSV-Free APIs in several popular Android versions (5.0+). These tested versions collectively occupy ~80% distribution of the Android market [17].

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

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

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

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

VI. Related Work

A. Regular postMessage Security

In past years, several detection and defense solutions for regular postMessage were proposed. However, all of them are incompetent to detect or defend against OSV. Barth et al. [22] conducted a systematic study of the frame isolation and communication, and enhanced postMessage. However, it could not prevent postMessage from being misused, and also did not support hybrid postMessage. Saxena et al. [7] highlighted the client-side validation vulnerability (CSV) in postMessage and proposed the detection tool "FLAX". Weissbacher et al. [8] applied the dynamic invariant detection technique in defending against CSV. Son et al. [6] conducted a systematic study of CSV on a large number of popular websites, and also proposed novel defense solutions to defend against CSV. Guan et al. discovered DangerNeighbor attacks on postMessage, and designed a deployable defense solution. However, they were only available to vet or protect the message receivers of $N \to W$, and could not eliminate OSV by making up the lost origins. Furthermore, since the source origin is not always provided due to V_2, their effectiveness may be impacted.

B. Android WebView Security

Recently, WebView security has attracted significant attention from researchers. Luo et al. [23] explored the potential attack vectors in WebView. Mutchler et al. [3] conducted a systematic study on a large number of hybrid apps. Wang et al. [24] studied the Intent abuse problem in hybrid apps. Georgiev et al. [4] conducted a systematic study on web-mobile bridges. Tuncay et al. [9] demonstrated the potential attacks on web-mobile bridges. Jin et al. [25] disclosed new attack channels for code injection attacks in WebView. Wu et al. [26] studied file:// based attacks. Rastogi et al. [27] discovered web-mobile bridges might be exploited by malicious content. Li et al. [28] disclosed a novel cross-app infection attack on WebView. Yang et al. [29] discovered a novel event oriented attack.

Several static analysis based approaches were proposed to vet hybrid apps. However, they were not suitable to detect OSV, since they failed to fill the semantic gap between the web and native layers. Furthermore, they all could not track origins, since the real data was missing. Chin et al. [30] statically analyzed WebView vulnerabilities that result in illegal authorization and file-based attacks. Yang et al. [10] and Hassanshahi et al. [14] proposed static analysis tools to vet hybrid apps armed with web-mobile bridges.

Other generic detection tools were also circumscribed to detect OSV. For example, Flowdroid [31] and Taintdroid [32] statically and dynamically applied taint analysis in the native layer. However, both could not fill the web-mobile gap.

Several defense solutions, such as NoFrak [4], Draco [9], MobileIFC [11], WIREframe [12], and HybridGuard [13], were designed to provide protection for WebView and web-mobile bridges. NoFrak and MobileIFC extended SOP into the native layer. Draco and HybridGuard enforced security policies for $N \to W$ by instrumenting either the chromium provide library, or JavaScript code. WIREframe provided bidirectional protections by directly instrumenting apps. However, as discussed in Section I, all of them were not suitable to protect hybrid postMessage.

VII. Discussion

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

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

Other ways to defend against V_4. Developers may retrieve the origin of the main frame through other ways, such as the native API $WebView.getUrl()$, which provide the URL for the current page. However, the API may fail and return NULL [33]. Developers may also maintain the status of current URL using event handlers [33]. However, this approach may also fail, since event handlers may not be successfully triggered [34].

VIII. Conclusion

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

Acknowledgment

We thank all framework/library/app developers, especially the Facebook security team, for helping us confirm the OSV issues. This material is based upon work supported in part by the National Science Foundation (NSF) under Grant no. 1314823 and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] "Web messaging standard," https://html.spec.whatwg.org/multipage/web-messaging.html.
[2] "Same origin policy," https://en.wikipedia.org/wiki/Same-origin_policy.
[3] P. Mutchler, A. DoupÃ, J. Mitchell, C. Kruegel, G. Vigna, A. Doup, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *MoST*, 2015.
[4] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *NDSS*, 2014.

[5] S. Farhang, A. Laszka, and J. Grossklags, “An economic study of the effect of android platform fragmentation on security updates,” in *ariv:1712.08222*, 2017.

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

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

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

[9] G. S. Tuncay, S. Demetriou, and C. A. Gunter, “Draco: A system for uniform and fine-grained access control for web code on android,” in *CCS*, 2016.

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

[11] K. Singh, “Practical context-aware permission control for hybrid mobile applications,” in *RAID*, 2013.

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

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

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

[15] “Adding a security warning about osv in the facebook react native framework,” https://github.com/facebook/react-native-website/pull/113.

[16] “Android webview message ports implementation,” https://developer.android.com/reference/android/webkit/WebMessagePort.html.

[17] “Android version distribution: Nougat and oreo up, everything else down,” https://www.androidauthority.com/android-version-distribution-748439/.

[18] “Ui/application exerciser monkey,” https://developer.android.com/studio/test/monkey.html.

[19] “An interactive tls-capable intercepting http proxy for penetration testers and software developers,” https://github.com/mitmproxy/mitmproxy.

[20] J. Schwenk, M. Niemietz, and C. Mainka, “Same-origin policy: Evaluation in modern browsers,” in *USENIX Security*, 2017.

[21] “The chromium projects,” https://www.chromium.org/.

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

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

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

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

[26] D. Wu and R. K. C. Chang, “Indirect File Leaks in Mobile Applications,” in *MoST*, 2015.

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

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

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

[30] E. Chin and D. Wagner, “Bifocals: Analyzing webview vulnerabilities in android applications,” in *WISA*, 2013.

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

[32] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” in *OSDI*, 2010.

[33] “Webview.geturl() returns null,” https://stackoverflow.com/questions/13773037/webview-geturl-returns-null-because-page-not-done-loading.

[34] “Android webview not calling onpagefinished when url redirects,” https://stackoverflow.com/questions/10592998/android-webview-not-calling-onpagefinished-when-url-redirects.