Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android

Güliz Seray Tuncay
University of Illinois at Urbana-Champaign
tuncay2@illinois.edu

Soteris Demetriou
University of Illinois at Urbana-Champaign
sdemetr2@illinois.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign
cgunter@illinois.edu

ABSTRACT

In-app embedded browsers are commonly used by app developers to display web content without having to redirect the user to heavy-weight web browsers. Just like the conventional web browsers, embedded browsers can allow the execution of web code. In addition, they provide mechanisms (viz., JavaScript bridges) to give web code access to internal app code that might implement critical functionalities and expose device resources. This is intrinsically dangerous since there is currently no means for app developers to perform origin-based access control on the JavaScript bridges, and any web code running in an embedded browser is free to use all the exposed app and device resources. Previous work that addresses this problem provided access control solutions that work only for apps that are built using hybrid frameworks. Additionally, these solutions focused on protecting only the parts of JavaScript bridges that expose permissions-protected resources. In this work, our goal is to provide a generic solution that works for all apps that utilize embedded web browsers and protects all channels that give access to internal app and device resources. Towards realizing this goal, we built Draco, a uniform and fine-grained access control framework for web code running on Android embedded browsers (viz., WebView). Draco provides a declarative policy language that allows developers to define policies to specify the desired access characteristics of web origins in a fine-grained fashion, and a runtime system that dynamically enforces the policies. In contrast with previous work, we do not assume any modifications to the Android operating system, and implement Draco in the Chromium Android System WebView app to enable seamless deployment. Our evaluation of the the Draco runtime system shows that Draco incurs negligible overhead, which is in the order of microseconds.

Keywords

Android, WebView, access control, origin, JavaScript bridges, exploitation, JavaScript, HTML5

1. INTRODUCTION

Mobile application (or "app" for short) developers heavily rely on embedded browsers for displaying content in their apps and libraries. A previous study shows that 85% of the apps in the Google Play store contain at least one embedded browser (i.e., WebView on Android) [1]. Other than the natural use case of just displaying web content, there are some interesting ways to use these web containers in apps: advertisement libraries use embedded browsers to display ad content within apps, app developers can rely on embedded browsers to tightly couple web sites with similar functionality to the app in order to reuse web site's UI code and to provide fast and convenient updates. Additionally, hybrid frameworks (e.g., PhoneGap) rely on embedded browsers to enable app developers to write their apps purely with web languages (e.g., JavaScript and HTML) with the premise of ease of programming and portability to other mobile operating systems.

Even though they are extremely useful, these embedded browsers come with their own security problems. They are inherently given the ability to execute web code (i.e., JavaScript). Additionally, through the use of JavaScript bridges, they can allow web code to interact directly with app components (i.e., internal Java code). Indeed, these bridges are what hybrid apps rely on to allow access to system resources such as contact list, camera, Bluetooth, SMS etc. Obviously, the misuse of this functionality by malicious web domains can be detrimental to the user and to the app since an attacker, whose web domain (hence malicious code) was loaded into a WebView can exploit the existing bridges to collect information about the user and even change the app's behavior. The main problem here is that there is no means of performing access control on the untrusted code running within a WebView, any origin loaded into the WebView is free to use all the available JavaScript bridges. With the introduction of API level 17, Android made an attempt to mitigate the negative consequences of this problem (i.e., accessing Android runtime via Java reflection) by introducing mechanisms to allow the developer to specify which methods will be exposed to JavaScript. However, this does not eliminate the problem as the untrusted code loaded into a WebView still inherits the same permissions as the host app and can exploit just the exposed parts of the bridge to perform its malicious activities. Since the origin (as in same origin policy) information is not propagated through the bridge, the app developer has no control over this access attempt and cannot perform any access control based on the origin.

Prior research studies on security issues in WebViews and JavaScript bridges fall short in at least four significant ways. First, they have limited scope, since they mainly target hybrid apps and create solutions that work only for the hybrid frameworks [2]. Second, they are incomplete, since they focus only on protecting permission-protected resources (such as the camera and microphone) [2, 3], and disregard other cases where a foreign domain is inadvertently allowed to access sensitive information (such as a user's social security number). Third, they rely on whitelisting policies that always

CCS'16, October 24-28, 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4139-4/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2976749.2978322

block unknown domains and therefore deprive developers of the flexibility to make decisions based on user input. Fourth, they are *ad hoc* since they focus only on a subset of resource access channels and do not provide a uniform solution that works across all channels.

The current disorganized and complex nature of interactions between web origins and applications creates confusion for developers. From our inspection of the apps in the Google Play store, we observed that the danger of loading untrusted web origins and exposing resources to them is not very well understood by app developers. Developers mistakenly assume that targeting API versions that address some of the issues with embedded browsers (e.g., using API level 17 or higher on Android) will protect their apps from these vulnerabilities. When they seem to be aware of the danger, assuring protection seems to be burdensome, and they tend to make mistakes while trying to evade the problem by implementing navigation control logic or multiple WebViews with different levels of exposure. However, even taking the correct programmatic precautions does not completely eradicate the problem since there is no guarantee that a trusted web domain will consist only of trusted components. Indeed, it is quite common for web pages to use an `iframe` in order to display ad content, and once loaded, there is no means for a developer to protect the resources that were exposed to web content from these potentially malicious components. All of this creates the necessity for an access control mechanism targeting web code where developers are given the ability to specify desired access characteristics of web origins in terms of app and device resources. Developers should be allowed to specify what capabilities should be given to web origins with a fine granularity, and if they need user input to make decisions. This brings forth the need for a policy language, which developers can use to describe the expected behavior and use of resources by web origins, without having to rely on any complex programmatic structures, and the need for a mechanism that will take into consideration the developer policies to make access control decisions.

In this work, we systematically study the vulnerabilities that are caused by loading untrusted web domains in WebViews on Android. We show cases where top-selling Android apps suffer from these vulnerabilities. Based on the threats we identified, we designed an easy to use, declarative policy language called Draconian Policy Language (DPL) for developers to specify access control policies on resources exposed to web origins. DPL allows declaration of policies with different levels of trust (i.e., fully-trusted, semi-trusted, untrusted) for different origins. We implement a system called Draco for fine-grained access control of web code: Draco enables app developers and device manufacturers (OEMs) to insert explicit Draconian policies into their apps, and dynamically enforces these policy rules at runtime in an efficient manner. Our contributions can be summarized as follows:

1. We model web origin access and design a new policy language for app developers and device manufacturers to dictate how web origins should access resources.
2. We provide a fine-grained access control runtime system for web containers to make access control decisions based on origins and their expected behavior without requiring OS modifications.
3. We provide a real world implementation that works on Android devices and evaluate the overhead of our approach.

The rest of the paper is organized as follows. In section 2, we give background information on how the Android embedded web browser works. In section 3, we describe the problems caused by the lack of a uniform access control mechanism in WebView in more detail, show our analysis on the use of WebView APIs by the top free apps on Google Play Store, and present case studies of top-selling Android apps that suffer from this problem. In section 4, we present the Draco framework, which consists of a declarative policy language for controlling web code execution and a runtime system that enforces the policies in Chromium's Android WebView implementation. In section 5, we evaluate our implementation. In section 6, we present related work on privilege separation and WebView vulnerabilities. Finally in section 7, we conclude with a discussion of our future work.

2. BACKGROUND

We refer to the applications that utilize WebViews as *mobile web apps* [1]. In order to understand the vulnerabilities caused by embedded browsers in mobile web apps, we need to have an understanding of the functionalities provided by these browsers. For the rest of the paper, we will focus on Android WebView, which is the widely-used open source embedded browser that forms the basic building block for modern web browser applications on the Android platform. This web container allows app developers to display web content fetched from the local storage or from the web. Developers use WebViews to seamlessly integrate web content into their apps, without having to rely on a full-featured, heavy-weight web browser to render web content.

2.1 WebView Implementation

WebView was first introduced in the API level 1 of the Android platform. It inherits from Android `View` and has additional rendering capabilities for displaying web pages. In Android 4.3 (JellyBean) and earlier, WebView implementation is based on Apple's WebKit browser engine [4], which powers several web browsers such as Safari, Google Chrome and Opera. Starting from Android 4.4 (KitKat), the WebView implementation is instead based on Chromium [5], which is Google's widely-used, open-source browser project. Chromium uses Google's fork of WebKit, called Blink, as a rendering engine, and Google's high-performance V8 JavaScript engine.

Up until Android 4.4 (inclusive), the WebView implementation resided in the Android Open Source Project (AOSP) [6]; hence, any update to the WebView requires modifications to the operating system and can be pushed to users only with an OS update. With the introduction of Android 5 (Lollipop), WebView became a system app (called Android System WebView), presumably to ship updates quickly to the WebView code through Google Play. Apps that use WebViews load WebView code as a library into the app's process from the System WebView app.

2.2 WebView API

The WebView API allows app developers to load web content by calling the methods `loadURL()`, `loadData()`, `loadDataWithBaseURL()` and `postURL()` with a string argument that is the URL of the desired web content. JavaScript can be enabled on a WebView by calling `setJavaScriptEnabled()` on a `WebSettings` instance of a WebView. The source of JavaScript can be a file on the local storage or a remote domain. Additionally, the app can directly execute JavaScript by calling `loadURL()` with a string that starts with *"javascript:"* and is followed by the JavaScript code.

Navigation. Android developers have the option of controlling navigation within WebViews. Whenever the user clicks on a link in a page on a WebView, the developer can intercept this to make a decision on how this page should be loaded, or if it should be loaded at all. Developers have the option of allowing page loading from only certain domains, and open pages from untrusted domains in the web browser. This can be implemented by overriding the `shouldOverrideUrlLoading()` callback method and checking the domain of the page before it is loaded .

JavaScript interfaces. The WebView API allows inserting Java objects into WebViews using the `addJavaScriptInterface()` method. JavaScript loaded in the WebView can have access to application's internal Java code, giving web code the ability to interact more tightly with an app, and in some cases get access to system resources (e.g., hybrid frameworks). Mobile web apps commonly utilize JavaScript interfaces to meld web content with application code and provide users with a richer user experience compared to pure web apps.

Listing 1 shows how JavaScript interfaces can be used in applications. First, the app needs to register a Java object with a specific WebView instance and give this object a name. As shown in the example, this can be done by `addJavaScriptInterface(new MyJSInterface(),"InjectedObject")`. After this, JavaScript code running in the WebView can execute the methods of this object by using the name of the object and the name of the method, as in `InjectedObject.myExposedMethod()`.

Android API 17 introduced the use of `@JavaScript` annotation tag to export only the desired Java methods of a Java class to JavaScript, primarily to prevent reflection-based attacks, where an adversary can use Java reflection to get access to the Android runtime and then execute arbitrary commands via calling `InjectedObject.getClass().forName("java.lang.Runtime").getMethod("getRuntime",null).invoke(null,null).exec(cmd)`. The use of the annotations is illustrated in Listing 1, where only the annotated method is made accessible to JavaScript. Even though API level 17 addresses a critical problem, it does not completely eradicate all the issues with WebViews. WebView still provides no access control on the JavaScript interfaces; any domain whose content was loaded into a WebView is free to use all the exported parts of the exposed Java object.

Listing 1: JavaScript Interfaces in Android WebView

```
mWebView.addJavaScriptInterface(new MyJSInterface(),
    "InjectedObject");
//...
public class MyJSInterface {
  @JavaScriptInterface
  public void myExposedMethod() {
    // do some sensitive activity
  }
  public void myHiddenMethod() {
    // JavaScript cannot access me, do some other activity
  }
}
```

JavaScript event handlers. The WebView API allows developers to handle the `alert`, `prompt` and `confirm` JavaScript events, by registering the `onJsAlert()`, `onJsPrompt()` and `onJsConfirm()` Java callback methods, respectively. Whenever the JavaScript side calls any of these event methods, their respective handler will be called, if it is overridden. The developer is free to implement any logic in these event handlers. In fact, these event handlers are used in some hybrid frameworks to connect the web side to the local side.

Handling HTML5 API requests. The rise of HTML5 has brought in a set of APIs that can give web applications the ability to access device hardware via JavaScript. Some examples to these HTML5 APIs are `Geolocation` and `getUserMedia`, which enable access to GPS and to media devices such as camera and microphone, respectively. When a web domain requests access to one of these devices, the user should be prompted to grant access to this request. Starting from API level 21, Android WebView provides support for these HTML5 APIs and introduces mechanisms to grant or deny requests for accessing device hardware. In order to handle requests from web origins, the developer needs to make use of `onGeolocationShowPrompt` (for geolocation), and `onPermissionRequest` (for media devices) to grant or deny permission to the requests. In Listing 2, we show an example of how HTML5 geolocation permission can be handled on Android. Listing 3 shows how granting permissions for HTML5 request will be combined with Android 6.0's run time permissions. Evidently, handling HTML5 requests can get cumbersome when combined with Android 6.0's run time permissions.

Listing 2: Granting access to HTML5 geolocation requests

```
@Override
public void onGeolocationPermissionsShowPrompt(String origin,
GeolocationPermissions.Callback callback) {
  myCallback = callback; //myCallback is global
  //If the permission is not yet granted, ask for it.
  if (ContextCompat.checkSelfPermission(getApplicationContext(),
      Manifest.permission.ACCESS_FINE_LOCATION)
    != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(thisActivity, new
        String[]{Manifest.permission.ACCESS_FINE_LOCATION},
        MY_PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION);
  } else { // Permission is already granted
    callback.invoke(origin, true, false);
  }
}
```

Listing 3: Run time permissions on Android

```
@Override
public void onRequestPermissionsResult(int requestCode,
String permissions[], int[] grantResults) {
  switch (requestCode) {
    case MY_PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION: {
      if (grantResults.length > 0 && grantResults[0] ==
          PackageManager.PERMISSION_GRANTED) {
        // permission was granted, do your location task
        myCallback.invoke(myOrigin, true, false);
      } else { // permission denied
       // disable functionality depenging on this permission.
      }
    //Handle other permissions...
```

3. UNDERSTANDING THE PROBLEM

In this section, we will discuss the lack of access control in Javascript bridges in WebView. We will argue that even though the Android APIs for handling HTML5 requests provide the means to perform limited origin-based access control (i.e., only for a subset of the device resources), developers simply avoid leveraging that due to the cumbersome and complex nature of the permission handling APIs. Finally, we will present our case studies on two mature and popular free Android apps that suffer from the nonexistence of access control in WebViews.

3.1 Lack of Access Control in WebView

The vulnerabilities in WebViews have been investigated by previous work [1, 7, 8, 9, 10]. A recurrent and fundamental problem is that there is no way of performing access control on the foreign code executed within a WebView; any origin loaded into the WebView is free to use the exposed JavaScript bridges. In particular, since the origin information is not propagated to the app through the bridges, the app developer has no control over the behavior of foreign code and cannot make access decisions based on the real origin of the invocation. With the introduction of API level 17, Android addressed some critical problems of WebViews such as reflection-based attacks by introducing Java annotations into the WebView API to limit the extent of exposure. However, this does not completely solve the problem as the foreign code loaded into the WebView still has the same permissions as the host app, and it can exploit the exposed parts of the JavaScript bridges to perform malicious activities such as accessing system resources, getting the user's private information, and executing code that was meant for use only by the web domain of the developer.

In order for a JavaScript bridge to be exploitable, the app must load untrusted content into the associated WebView. An obvious way is by allowing the WebView to navigate to untrusted websites or to sites with untrusted content (e.g., `iframe`). Previous work shows that navigation to untrusted sites is common among applications: 34% of the apps that use WebViews do allow the user to navigate to third-party websites [1], and 42.5% of the apps that register a JS interface allow the user to navigate to third-party websites or to websites with untrusted content [8]. In order to verify these results, we picked three top-selling Android apps that demonstrate the common vulnerabilities identified by previous work: USPS, CVS Caremark, and JobSearch by Indeed. Through manually analyzing their code, we observed that developers do try to take precautions against the attacks on JS bridges by loading pages from untrusted domains in either the browser instead of the WebView (e.g., USPS app), or in separate WebViews with limited functionality which they create for this purpose (e.g., JobSearch app by Indeed). However, developers can make mistakes while implementing the navigation control logic. For example, in the USPS app, the developer checks if the loaded URL contains "usps.com" rather than checking if the host's domain name matches "usps.com", mistakenly allowing any non-USPS website that partially matches "usps.com" (e.g., musps.com, uusps.com). Additionally, developers might make wrong assumptions about the navigation behavior of the WebView. We have identified that the app developer might assume that the content provided to the WebView intrinsically does not allow navigation (i.e., it does not contain hyperlinks) and provide the user with functionalities that can break this assumption (e.g., allowing users to input hyperlinks) as in JobSearch app by Indeed, or they simply do not foresee that a specific WebView can be used by the user to navigate out of the trust-zone of the app by just following the links on the web pages as in the CVS Caremark app. We will examine the CVS Caremark and JobSearch apps in more detail later in this section.

Although it may look like correct implementation of navigation control would solve the JavaScript bridge exploitation issues (and fix the USPS app), we argue that it is simply an insufficient measure to protect JavaScript bridges. Even if developers implement all navigation behavior correctly and do not allow the user to navigate to untrusted web origins within the context of their apps, the pages from trusted domains might include untrusted components such as `iframes`, which also inherit the same permissions as the app and have access to all the exposed bridges. Thus, the system does not provide the necessary means for developers to completely protect their apps against attacks on the JavaScript bridges.

3.2 Prevalence

While the current design of JavaScript bridges by default grants access to a domain for all the exposed resources, for HTML5 APIs the access model is exactly the opposite; the default behavior is to deny all the requests by a domain for a permission unless the `onPermissionRequestResult` and `onPermissionsRequest` or `onGeolocationPermissionPrompt` methods are overridden by the app developer. In order to better understand how this difference between the JavaScript bridges and the HTML5 APIs affects the developers, we statically analyzed the top 1337 free Android applications from 21 Google Play categories selected at our discretion. Table 1 depicts the prevalence of WebViews and how often WebView APIs are used in these apps. Here, we distinguish ad and core WebViews (WebView in the core of the app) based on our comprehensive list of package names for ad libraries, and also give the cumulative result including both uses of WebView. In line with the previous work [1, 8], we have observed that WebView is a commonly-used component as around 92% of the applications in our dataset make use of it in their core application code (i.e., not used by advertisement libraries). Among the applications that include at least one WebView in their core code, 77% of them use JavaScript interfaces, and 70% use event handlers. However, it can be observed that there is a sudden drop in the numbers when it comes to the use of HTML5 APIs. This might be happening for two reasons. On the one hand, developers generally do not wish to grant access to permission-protected resources to external domains, and apps can operate without having to rely on external web origins. On the other hand, even though more than 85% of the apps in our dataset target API 21 or higher and are able to handle HTML5 API requests, they simply decide not to do so, possibly due to the complex request handling logic of the HTML5 APIs. Since the majority of the apps require API 21 or higher, they need to comply with the run time permissions introduced by Android 6.0. This means that each time they wish to grant access to a web domain, they also need to check if the app was granted the permission of interest by the user and, if not, they must prompt the user to grant it. On top of this, they also need to implement origin-based access control; hence, they need to maintain the necessary data structures and track user preferences. This process is unnecessarily cumbersome. If the system provides the necessary infrastructure to allow developers to uniformly declare their security policies, this would minimize effort and reduce the likelihood of errors, irrespective of the underlying channel (whether that is a JavaScript interface, an event handler or an HTML5 API).

3.3 Case Studies

Previous work has shown how applications built with hybrid frameworks suffer from JavaScript bridge vulnerabilities since hybrid frameworks rely on these bridges to give application code access to device resources like the camera, contact list and so on [2, 3]. However, the problem is not constrained to hybrid applications. In fact, an application that uses an embedded browser to display web content and needs enhanced communication between the web domain and the app, is susceptible to similar exploits. Here, we investigate the understudied JavaScript bridge issues that exist in non-hybrid applications. In particular, we present our analysis on two widely-deployed applications distributed through the Google Play store, which we found as suffering from JavaScript bridge issues. Indeed, we show that the exposure of these bridges to adversaries can be detrimental to users' privacy and can adversely affect the application's flow. We have disclosed these issues to the developers of those apps but—at the time of writing—haven't received a response.

CVS Caremark. CVS Caremark is one of the Android apps offered by the American pharmacy retail company CVS. It has been downloaded 100,000 times so far and currently has a rating of 3.6. It allows users to track their prescription history, get refills or request mail service for new prescriptions, and get information about drugs and their interactions. In order to help their users with their medical needs, the app requires them to register to the CVS system with their name, health care ID, and email address. The app additionally tracks some other personal information, including the user's phar-

Used Web Features	# (%) in core	# (%) in ad	# in both core and ad	Total # (%)
WebView	1226 (92%)	551 (41%)	544 (41%)	1233 (92%)
JavaScript enabled	1189 (89%)	495 (37%)	482 (36%)	1202 (90%)
JavaScript Interfaces	945 (71%)	395 (30%)	361 (27%)	979 (73%)
@JavaScriptInterface	587 (44%)	328 (25%)	182 (14%)	769 (58%)
onJsPrompt	857 (64%)	202 (15%)	172 (13%)	887 (66%)
onJsAlert	696 (52%)	259 (19%)	227 (17%)	923 (69%)
onJsConfirm	699 (52%)	214 (16%)	184 (14%)	883 (66%)
onGeolocationPermissionsShowPrompt	567 (42%)	169 (13%)	133 (10%)	700 (52%)
onPermissionRequest	32 (2%)	0 (0%)	0 (0%)	32 (2%)

Table 1: Prevalence of WebViews and use of WebView APIs (#: absolute number, %: percentage)

macy preferences and location. Furthermore, the app implements some functionality to check the login state of users, retrieve some internal database IDs, perform UI functionality such as displaying date pickers, and invoke the browser to load a given URL.

CVS Caremark app uses WebViews to render web content, and utilizes JavaScript interfaces to enable a tight communication between the app and CVS web servers. It registers two different interfaces with the WebView, of classes `WebViewJavascriptInterface` and `JavaScriptWebBridge`, and names the JavaScript objects associated with these interfaces *"native"* and *"WebJSInterface"* respectively. We observed that the *"native"* interface implements some of the main functionalities of the app (e.g., scanning prescriptions, registering users etc.) with access to device resources and exposes user's private information. Hence, it is highly possible that this interface was meant by the developer to be used for internal use only, that is by trusted CVS domains. On the contrary, the *"WebJSInterface"* is possibly meant to be used in a more generic context and by untrusted domains; hence, it exposes functionalities more conservatively. However, this attempt to protect the app and device resources by creating two interfaces is not useful, since both of these interfaces belong to the same WebView instance, which is used to load all URLs. Hence, the security of the app relies on implementing navigation control correctly, by not allowing the app to navigate to untrusted domains or to domains that might contain pages with untrusted elements. The app simply does not make any attempt to mitigate the problem by implementing navigation control to filter untrusted domains; hence, it is vulnerable to JavaScript bridge attacks. Even if navigation control was implemented correctly, this would not be enough to protect from these attacks since even a trusted origin may include elements that are of risky nature (e.g., `iframe`). We have successfully performed an attack on this JavaScript interface bridge by navigating to our attack URL which runs the code in 4. The attack domain was able to retrieve personal information (like the user's name, health care ID, email address, pharmacy preference, and location) as well as execute app functions such as using the camera on the victim device for barcode scanning functionality and retrieving images of user's prescription drugs.

Job Search by Indeed. Job Search is an app released by Indeed, a company that produces an employment-related search engine (*indeed.com*), to allow users to search for jobs on their Android devices. The app is downloaded 10,000,000 times and has a rating of 4.1.

Most of the content displayed to the user in the app is fetched from Indeed's web domain (*indeed.com*) and rendered in a WebView. This is done mainly to reuse the UI code of Indeed's web domain in order to reduce the app development effort and simplify maintenance of its deployment. Similarly to the CVS Caremark app, JobSearch creates and uses two types of WebView classes, one (of class `IndeedWebView` that extends the `WebView` class) for internal use and another (of class

Listing 4: JavaScript interface exploitation in CVS Caremark

```
function beEvil() {
   deviceInfo = native.getDeviceInfo()
   clientID = native.getBenefactorClientInternalId()
   geolocation = native.getGeoLocation()
   loginState = native.getLoginState()
   userName = native.getUserName()
   preferredPharmacy = native.getPreferredPharmacy()
   native.scanRx() // scan barcodes
   // get prescription barcode image
   prescriptionImage = native.getFrontRxImgData()
   data = constructData(deviceInfo, clientID, geolocation,
         loginState, preferredPharmacy, userName,
         prescriptionImage)
   // Send data to server
   b=document.createElement('img')
   b.src='http://123.***.***.***/?data='+ data
   native.setPreferredPharmacy("WhicheverPharmacyIWant!")
}
```

`ExternalWebView` that extends `IndeedWebView`) for showing external content such as job descriptions from untrusted domains. The app attaches a JavaScript interface (named *"JavaScriptInterface"*) to the internal WebView, while not exposing any such interfaces to the external one in an attempt to protect resources from external domains. The app also takes precautions to restrict navigation in the internal WebView by removing all the hyperlinks in the rendered text content; hence, it supposedly does not allow loading of external URLs in this WebView. However, the app also offers the user the choice of adding web sites as a part of their profile and allowing the user to navigate to these sites. This breaks the developer's no-load assumption on the internal WebView for pages from external domains, and thereby puts the exposed JavaScript interfaces at risk. In fact, we were able to access the JavaScript interface by navigating to our "malicious" website in the internal WebView. In this interface, the app offers the device's unique ID, enabling/disabling Google Now, checking if this device is registered with Indeed, getting user's registration ID, and registering the device with Indeed.

3.4 Adversary Model

Based on our observations on how JavaScript bridge vulnerabilities can be exploited, we assume a web adversary who owns web domains and/or ad content in which he can place malicious code. Mobile web apps render the malicious domains and malicious ad content through their embedded browsers. Such an app can reach a malicious domain when the user starts navigating through the embedded browser. Moreover, malicious ad content can be offered by the adversary to both the app's trusted domains and to other untrusted domains. We assume that the adversary can reverse engineer the victim app code to identify the exploitable JavaScript bridges. The adversary can achieve this by first downloading the victim app's apk using existing frameworks for crawling Google Play store, and then decompiling it with any of the existing dex decompilers (dex2jar [11], JD-GUI [12], apktool [13] etc.).

4. DRACO ACCESS CONTROL

Running code from untrusted origins in a WebView can be detrimental to users as the foreign code can compromise users' privacy and disturb their experience by exploiting the WebView's tight-coupling with the application code and device resources. A straightforward way to address this threat is to simply prevent the user from visiting untrusted web pages. However, there are cases where this is impossible to achieve since even trusted domains might embed untrusted components in their web pages. Hence, it is necessary to provide an access control mechanism for WebViews that can distinguish the source of foreign code that is being executed and grant access only if the source is trusted by the developer or by the user. To tackle this problem, in this work, we propose *Draco*, a fine-grained, origin-based access control system for WebViews, which consists of two major components: 1) an access control policy language that we call the *Draconian Policy Language* (DPL), which allows app developers to declare policy rules dictating how different components within a WebView should be exposed to different web origins, and 2) a runtime system we call *Draco Runtime System* (DRS), which takes policy rules on system and internal app resources (i.e., JavaScript bridges) as an input from the developer and enforces them dynamically when a resource request is made by a web origin.

4.1 Design Goals

Before we go into the depths of our policy language and the Draco runtime access control on WebViews, it is important to discuss what our design goals are and how they affect the architecture of our system. Previous work focuses on access control only on the permission-protected parts of the exposed bridges in hybrid frameworks. Our goal is to provide developers with a fine-grained access control model, which will enable them to express access control policies on *all* parts of *all* access channels for *all* use cases of WebViews (i.e., in hybrid and native apps). These channels are namely the *JavaScript interface*, the *event handlers*, and the *HTML5 API*. App developers should be given full control on all of the channels, that is, they should be able to specify which origins can access which parts of the channels and assign permissions to trusted origins. They should also be given the flexibility to delegate decisions to the user when needed. Draco should avoid modifications on any parts of the operating system and should be implemented as part of a userspace app. This would allow the system to be readily and immediately deployable, while enabling frequent updates that are disjoint from firmware updates. Additionally, Draco should be able to enforce policy rules efficiently and its policy language should be easy to understand and use for developers, as well as easy to extend if necessary.

4.2 Draconian Policy Language

Draco supports a declarative policy language that allows app developers to describe their security policies with respect to remote code origins. Here we present the *Draconian Policy Language* (DPL) and provide examples to demonstrate its expressiveness.

Grammar. We want to instantiate a capability-based access control scheme based on least privilege and allow specification of what resources each remote origin can access and how they can access them. By default, if a DPL rule does not exist to allow the web code to access any resource, then access is denied. We use the Backus-Naur Form (BNF) notation [14] for context-free grammar to describe the new policy language. Terminals are denoted by single-quoted literals.

Draco allows developers to write policy rules which dictate how sensitive resources can be accessed by web code. We define the syntax of a DPL rule as:

$$\langle \textit{policy rule} \rangle ::= \langle \textit{subject} \rangle \text{ `;' } \langle \textit{trust level} \rangle \mid \langle \textit{subject} \rangle \text{ `;' } \langle \textit{channel} \rangle \text{ `;' } \langle \textit{decision point} \rangle$$

Each Draconian policy rule is applied on a subject. The *subject* indicates the web origin whose web content was loaded in the WebView. Here, a remote origin is represented by a URI scheme (i.e., http or https), a hostname and a port number as in the same origin policy. We allow wild cards for origins in our language, in order to allow creation of rules that can be applied to any origin. Additionally, we allow wild cards for sub-domains in domain names (e.g., (*).mydomain.com) to enable rules that can assign all hosts under the same domain the same access characteristics.

$$\langle \textit{subject} \rangle ::= \text{`}{*}\text{'} \mid \langle \textit{protocol} \rangle \langle \textit{hostname} \rangle \langle \textit{port} \rangle$$

$$\langle \textit{protocol} \rangle ::= \text{`}\texttt{http://}\text{'} \mid \text{`}\texttt{https://}\text{'} \mid \varnothing$$

$$\langle \textit{hostname} \rangle ::= \langle \textit{subdomain} \rangle \langle \textit{domain name} \rangle$$

$$\langle \textit{domain name} \rangle ::= \text{string}$$

$$\langle \textit{subdomain} \rangle ::= \text{`}\texttt{(*).}\text{'} \mid \langle \textit{name} \rangle \text{ `}\texttt{.}\text{'} \mid \varnothing$$

$$\langle \textit{name} \rangle ::= \text{string}$$

$$\langle \textit{port} \rangle ::= \text{`}\texttt{:}\text{'} \langle \textit{port number} \rangle \mid \varnothing$$

$$\langle \textit{port number} \rangle ::= \text{integer}$$

A *trust level* is an abstraction that allows developers to instantiate default policies. Our system supports three trust levels:

$$\langle \textit{trust level} \rangle ::= \text{`}\texttt{trustlevel}\text{' `<' } \langle \textit{trust level options} \rangle \text{ `>'}$$

$$\langle \textit{trust level options} \rangle ::= \text{`}\texttt{trusted}\text{'} \mid \text{`}\texttt{semi-trusted}\text{'} \mid \text{`}\texttt{untrusted}\text{'}$$

A *trusted* subject is allowed to access all resources whereas an *untrusted* subject is never allowed to access any resources through any channels. A *semi-trusted* domain can access exposed functionality but only through user interaction. At this point it should be clear that our policy language allows for essentially whitelisting domains. However, this is still not expressive enough. Consider for example the case that we want to allow a subject to access the exposed JavaScript interfaces but not run HTML5 code that can unilaterally access resources. Towards this end, the second part of the DPL rule definition allows for such fine-grained declarations. In particular an app developer can specify which *channel* should be protected. A *decision point* dictates whether such a policy rule should be enforced transparently to the user or only when the user agrees to it. If left empty, then "system" is assumed which forces the system to enforce the rule transparently to the user. If "user" is chosen then the system delegates the enforcement decision to the user at the time of the access attempt. DPL also allows app developers to provide a description message for the user. This can be useful in cases the DPL rule governs resources at a very fine-granularity (e.g. at the method level) which might be challenging for the user to understand.

In such cases a semantically meaningful message provided by the app developer could help the user better perceive the context.

⟨*decision point*⟩ ::= ‘`decisionpoint`’ ‘<’ ⟨*decision maker*⟩ ‘>’ ⟨*description*⟩

⟨*decision maker*⟩ ::= ‘`system`’ | ‘`user`’ | ∅

⟨*description*⟩ ::= ‘<’ ⟨*text*⟩ ‘>’ | ∅

⟨*text*⟩ ::= string

The *channel* definition is more intricate: Draco needs to allow greater levels of rule expressiveness to enable developers to dictate fine-grained policies. Every channel has its own idiosyncrasies and exposes resources in different ways. This obviates the need for allowing different specifications for each channel. Thus, an app developer should be able to choose the channel they want to protect:

⟨*channel*⟩ ::= ⟨*event handler*⟩ | ⟨*html5*⟩ | ⟨*jsinterface*⟩

Our access control follows a least privilege approach: by default everything is forbidden unless there is a rule to allow something to happen. In particular, for the `event handler` channel, our policy allows app developers to specify how the whole channel can be accessed by the subject, but also—need be—to define which permission-protected APIs can be utilized by each event handler method. This is reflected in the policy language as follows:

⟨*event handler*⟩ ::= ‘`alloweventhandler`’ ‘;’ ‘<’ ⟨*eh methods*⟩ ‘>’ ‘;’ ‘<’ ⟨*permission list*⟩ ‘>’

An event handler method list (*eh method list*) can be a single event handler method, or a list of event handler methods. Furthermore, developers should be allowed to specify a list of permissions for the exposed event handler methods:

⟨*eh methods*⟩ ::= ‘`all`’ | ⟨*eh method list*⟩

⟨*eh method list*⟩ ::= ⟨*eh method*⟩ | ⟨*eh method*⟩ ‘,’ ⟨*eh method list*⟩

⟨*eh method*⟩ := ‘`onJsHandler`’ | ‘`onJsPrompt`’ | ‘`onJsConfirm`’

⟨*permission list*⟩ ::= ⟨*permission*⟩ | ⟨*permission*⟩ ‘,’ ⟨*permission list*⟩ | ∅

where a <*permission*> can be any of the Android permissions.

Similarly, for the *html5* channel one can specify the WebKit permissions that web code can make use of:

⟨*html5*⟩ ::= ‘`allowhtml5`’ ‘;’ ‘<’ ⟨*HTML permission list*⟩ ‘>’

⟨*HTML permission list*⟩ ::= ⟨*HTML permission*⟩ | ⟨*HTML permission*⟩ ‘,’ ⟨*HTML permission list*⟩

⟨*HTML permission*⟩ ::= ‘`VIDEO_CAPTURE`’ | ‘`AUDIO_CAPTURE`’ | ‘`GEOLOCATION`’ | ‘`PROTECTED_MEDIA_ID`’ | ‘`MIDI_SYSEX`’

Lastly, for the *jsinterface* channel, our policy language allows developers to describe how every Java class and Java method exposed to JavaScript can be accessed by the subject:

⟨*jsinterface*⟩ ::= ‘`allowjsinterface`’ ‘;’ ⟨*class methods*⟩ ‘;’ ‘<’ ⟨*permission list*⟩ ‘>’

⟨*class methods*⟩ := ⟨*class name*⟩ ‘<’ ⟨*methods*⟩ ‘>’

⟨*class name*⟩ ::= string

⟨*methods*⟩ ::= ‘`all`’ | ⟨*method list*⟩

⟨*method list*⟩ ::= ⟨*method name*⟩ | ⟨*method name*⟩ ‘,’ ⟨*method list*⟩ | ∅

⟨*method name*⟩ ::= string

Expressiveness. As with any language, there exist an intrinsic tradeoff between the usability of the policy language and its expressiveness. On the one hand, a usable policy language is of low complexity but at the same time limited on the policies it can express. On the other hand, a complex policy language can express more fine-grained rules. DPL strikes a careful balance between the two by aiming to express selected set of useful policies at the method level with one-line, concise rules.

Consider for example the case where a developer of a low risk application would like to allow their web service (*“mydomain.com”*) to run code within the WebView of the host app. In such cases, the developer could simply provide a rule as follows:

```
https://mydomain.com;trustlevel<trusted>
```

Given only this rule, the system forbids any web code of origin other that *“mydomain.com”* to access any exposed functionality from the host app. At the same time, the trusted *“mydomain.com”* can benefit from all the exposed features.

In the aforementioned vulnerable case of the CVS Caremark app (see Section 3), it is evident that the app developers wanted to allow the CVS domains to access a rich JavaScript interface (*WebViewJavascriptInterface*) and other domains to access a more conservative *JavaScriptWebBridge* interface. This could be simply described by the app developer and enforced by DRS providing the following DPL rules:

```
https://www.caremark.com;allowjsinterface;
      WebViewJavascriptInterface<all>;decisionpoint<system>

*;allowjsinterface;JavaScriptWebBridge;decisionpoint<user>
```

where “*” is a wildcard that can match any origin. The former rule allows only CVS domains to access the sensitive APIs, whereas the non-sensitive app functionality can be exposed to all domains after user approval with the latter rule.

In the case of the “Job Search” app by Indeed, the developer could simply provide the rule:

```
(*).indeed.com;allowjsinterface;JavaScriptInterface;
      decisionpoint<system>
```

This will allow only code from *“indeed.com”* to use the exposed JavaScript interfaces. The developer does not need to worry about implementing two different WebViews, one for secure domains and one for untrusted domains. Furthermore, navigation will not be an issue as the system transparently allows only the *“(*).indeed.com”* domains to access sensitive APIs.

In fact, we identified by looking at the decompiled application code that the “Job Search” developers have written around 550 lines of code, aiming to achieve separation between trusted and untrusted domains by using a second fully-developed WebView (along with custom-built Activity, WebViewClient, WebChromeClient classes), and yet the app was still vulnerable. In contrast, one line of code with the Draconian Policy Language is enough to secure the app with Draco. Additionally, even though DPL allows the construction of very fine-grained policies, it can be seen from these examples that simple and easy-to-construct policy rules can be sufficient in many practical cases. Consequently, our system can provide strong protection to apps and minimize developers' efforts with an easy to use policy language.

MyStore example. To demonstrate more fully the expressiveness of DPL, we consider a more elaborate scenario. Let us assume that *MyStore* is a large retail company that aims to incorporate Eddystone [15] Bluetooth Low Energy (BLE) [16] beacons on product shelves in its stores [17]. These beacons broadcast URLs for the product they advertise using the BLE protocol. *MyStore* also provides its clients with a shopping app, namely *MyStore App*, which scans for the BLE beacon advertisement messages and displays the web page of the advertised products in a WebView. The advertised websites provide further information about the product such as description, images, reviews etc. These web pages can belong to the web domain of *MyStore* (*mystore.com*), or to the *MyStore* suppliers that partner with *MyStore* to use store beacons. *MyStore App* collects a user's profile and preferences, allows her to scan product barcodes, and also acquires the location of the user's mobile device to perform analytics in order to better their services.

MyStore App is a mobile web application that uses components from *mystore.com* via the help of the WebView embedded browser, and exports device resources and app functionalities through Java Script bridges. In particular, it exposes the `MyInterface` Java class, which features the following functions: `getAge()`, `getGender()`, `getStoreLocation()`. *MyStore* wants to allow *mystore.com* to access all the JavaScript Interfaces and resources of *MyStore App*. At the same time, it wants to allow its partners' web domains to access the location of the store for them to know where their products are being seen. Furthermore, it wants to provide its partners the opportunity to get the user's age and gender. However, providing this information might entail privacy concerns. Thus, the store wants to allow their partner to access this information only if the user agrees, which will not violate the user's expectations and thus minimize the privacy risk. Enforcing this complex interactions can be extremely challenging with the current state of affairs on the Android platform, since it might require implementing multiple WebViews with different levels of exposure and duplicating method definitions. However, it becomes much easier when Draco is used. Consider for example the following Draconian policy rules:

```
mystore.com;trustlevel<trusted>

partner.com;allowhtml5;permission<GEOLOCATION>

partner.com;allowjsinterface;MyInterface<getLocation>;
      decision<system>

partner.com;allowjsinterface;MyInterface<getAge,getGender>;
      decision<user><"Access to age and gender">
```

The first rule allows *mystore.com* to access all exposed resources on all channels. The second and third rule allow the trusted *partner.com* domain (e.g., partner companies) to access the location through either HTML5 APIs or exposed interfaces of the *MyStore App*. Finally, the last rule allows *partner.com* to access the exposed functions that provide the user's age and gender only if the user agrees.

4.3 Draco Runtime System

Draco's other major component is its runtime system, namely the *Draco Runtime System* (DRS). A key aspect of the design of DRS is to avoid any modifications in the Android OS to make DRS easier to both deploy and update. In particular, DRS is built on top of the WebView system app on Android. This requires making modifications only in the Chromium source code [5], which is the provider of the WebView implementation on the Android platform.

High-level architecture. Keeping our design goals in mind, we implemented DRS in Chromium [5], which is a system library residing in the Android WebView system app, providing the WebView implementation to the Android apps. Figure 1 illustrates our design in more detail. The app developer provides DPL rules to the WebView programmatically through their Android app. DRS features a Policy Manager class (i.e., `PolicyManager`), which parses the policy rules and inserts them into a policy map data structure. We also implemented a unit for decompiling the app and statically analyzing it to determine the permissions necessary to successfully execute the methods in exposed Java class methods and event handlers. This is necessary because we allow the developer to assign subjects a set of permissions they are allowed to use for each channel and do not assume any cooperation other than entering policy rules. In particular, in the case of JavaScript interfaces and event handlers, we need to know beforehand what permissions the requested method uses in order to determine if the subject under investigation is granted all of the permissions required by that method. DRS employs an *Information Unit* which hosts and manages the two data structures corresponding to policies (policy map) and the permissions used by class methods and event handlers (permission map). During invocation on any of the three aforementioned channels, DRS intercepts the invocation and checks if any one of the policy rules allows the subject to execute or access the requested part of the channel, or if the user needs to be prompted for a decision. If the request is allowed according to the developer policies or by the user, DRS lets the invocation go through, otherwise it gracefully blocks the request.

Note that it could be the case that the app itself runs web code loaded from its local files. For example, the `loadUrl()` method can be invoked with "file://" which loads a html file from the local storage or with "javascript://" which invokes the inline JavasSript code with no origin. Since the subject of this execution is the app, we treat this as trusted. However, there could be cases where web code can run with undefined origin (e.g., by loading code with the `eval` JavaScript function). Our system completely blocks such attempts.

Parsing module. Draco allows developers to enter policy rules into the WebView without having to modify the source code of the Android Open Source Project (AOSP). We considered extending the Android Manifest file with policies (as was done in previous work), as well as using annotation tags as policy rules on the exposed Java code. However, both of these approaches require changes to be made on the Android platform itself; the former requires changing the parsing logic in the *Package Manager*, and the latter requires changing the WebView APIs.

In our work, we exploit the existing `loadUrl` method in the WebView API, and piggy-back rules into the WebView. As we explained before (see Section 2), the `loadUrl` method takes a URL string as an input argument and loads this URL. It can also execute JavaScript code if the given string starts with the *"javascript:"* tag. We extend the functionality of `loadUrl` within the WebView system app, to capture strings that start with the *"policyrule:"* tag which indicate a Draconian policy rule for the WebView. DRS's *Policy Manager* class is implemented in the facade Java layer of Chromium, and is responsible of parsing DPL rules and inserting them into the data

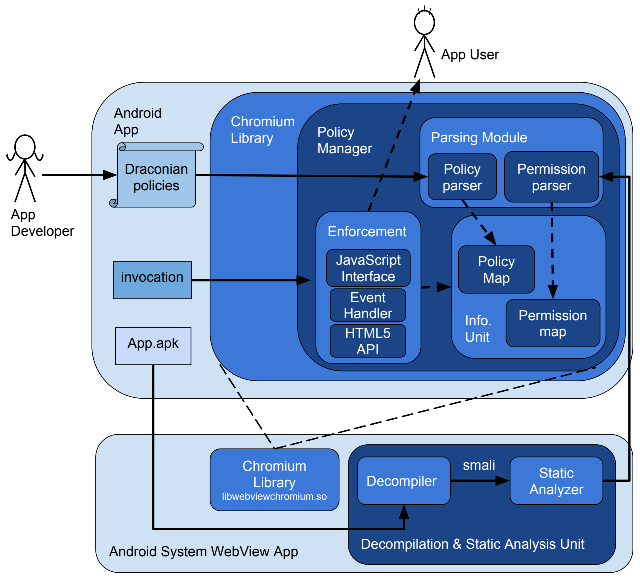

Figure 1: Draco Runtime System (DRS) architecture.

structures we utilize for enforcement. This class uses the Java Native Interface (JNI) to talk to its native "back end" that performs these functionalities in C++. This structure makes it easier to communicate the policies to the Chromium implementation that performs most of its main functionalities in native code (C++).

App decompilation and static analysis unit. For the enforcement of permission-based DPL rules that regulate the use of sensitive APIs in the JS interface and event handler channels, we need to determine the permissions used in a given class and its methods for JavaScript interfaces, and permissions used in the event handler callback methods for the event handlers channel. Such a permission-based rule might be expressed informally as follows: "Origin X can access a method which makes sensitive API calls that require permissions Y, Z only if it's given these permissions by the developer.". Doing this without the help of the developer (e.g., by submitting the list of methods with the used permissions) requires static analysis on the application code to retrieve the permissions used in the JavaScript interface classes and their methods, and in the event handlers. DRS uses apktool [13], which works with a success rate of 97.6% [18], to decompile the app into smali bytecode, then finds the exposed JavaScript interface classes and the event handlers declared in DPL rules, and processes them to find the permissions that are required by them. This requires determining the sensitive API calls made in the class methods and the event handlers. DRS utilizes the API-to-permission mapping released in PScout [19] and searches for the API calls specified in this mapping in the app's decompiled smali code. This provides DRS with a list of used sensitive API calls and their corresponding permissions required by each class method and event handler. DRS performs this process in the background, only during app installation and update time and saves the results in a file in the app's data folder to avoid repetition of this process.

Information unit. At the core of the PolicyManager lie two components: the *policy map* and the *permission map*. The *Policy map* uses a hash map to keep track of the developer DPL rules (as key value pairs). It uses a string for its key, and a vector of strings for the value. The key consists of the subject, channel name, and type of the channel option, namely class name (for interfaces), function (for event handlers), or permission (all channels). In case the channel option is a JavaScript Interface class name, the vector of strings (value for that key) consists of the name of the allowed methods for that interface. As for permission for the channel option, this vector will be the name of the allowed permissions. The *Permission map* is used for tracking the used permissions in a given class and its methods for JS interfaces, and for tracking permissions used in event handler methods. The list of permissions for these methods will be retrieved from a permission file created by the aforementioned static analysis unit.

Enforcement. Chromium works based on a multi-process architecture (Figure 2). Each tab (called Renderer in Chromium jargon) in the browser is a separate process and talks to the main browser process through Chromium inter-process communication (IPC). Even though WebView is based on Chromium, it does not inherit this multi-process architecture. Instead, it keeps the same code structure but adopts a single process architecture, due to various reasons including the difficulty of creating multiple processes within an app in Android, concern for memory usage in resource-limited environments, synchronization requirements of Android Views, as well as other graphics related issues. DRS policy enforcement is implemented in the native part (C++) of the `Browser` component of the Chromium code 2. This is because this is the point of invocation for all of the channels and most of the necessary information to perform access control already resides in this component.

• *JavaScript interfaces.* For the JavaScript interfaces, the origin information is not passed to the Browser component with the creation of a bridge object. Hence, in the Renderer, we get the security origin (as in same origin policy) of the calling frame and propagate this information to the Browser. When the invocation happens, we know the name of the class method the origin wants to execute, however, we have no way of knowing the class name of the object this method belongs to since C++ provides no way of retrieving the name of a Java object's class in run time. That is why, the retrieval of the class name needs to be done in the Java facade layer in Chromium in order to be communicated later to the native layer. This is done after the call to `addJavaScriptInterface` (so that we do not have to change this API method) and before the object moves to the native side (so that we still have class name information). Given the channel name, class name, method name and the web origin, the *Policy Manager* is able to make an access control decision for this origin using the aforementioned data structures to check if the required policies exist for this domain to execute the method. The *Policy Manager* first checks if the origin has access to the given method of the class, and if that is the case, it performs the permissions checks to see if the permissions given to the origin would be a superset of the permissions used by the requested method. If the origin passes both checks, then it is allowed to perform the invocation of the requested method. However, if the decision point set in the respective DPL rule is set to "user", then the user is prompted with the description provided for this rule to make the access control decision.

• *Event handlers.* For the event handler channel, all the information necessary for enforcement exists in the Browser component. Hence, given the channel name, origin, and the name of the event handler method, the *Policy Manager* can enforce the policy. The enforcement logic is similar to that for JavaScript interfaces. The *Policy Manager* first checks if the origin is allowed to execute the event handler method for a given JavaScript event. If so, it then performs permission checks on the event handler to see if the origin is granted the permissions to execute the handler. If that is the case, then the origin is granted access to this event handler.

• *HTML5 API.* For this channel, DRS intercepts the entry point of the `onGeolocationShowPrompt` callback for geolocation and the `onPermissionRequest` callback for other HTML5 resources in the

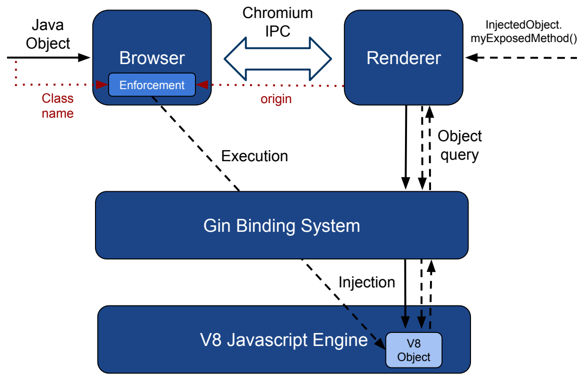

Figure 2: Enforcement in Draco implemented in the Browser component in Chromium for all channels. For JS interfaces, Java objects are inserted as V8 objects into Chromium. Draco needs class name and origin information from outside the context of Browser component.

native part of Chromium code. We retrieve the resources from the request object in Chromium and for each resource ask the Policy Manager whether there is a rule allowing the origin to use that permission, or ask the user if the developer specifies in the DPL rule for the user to be consulted. If the origin is allowed by a DPL rule (and by the user if needed), DRS allows the origin to go through with the invocation, else DRS gracefully blocks the request.

5. SYSTEM EVALUATION

In this section, we evaluate the effectiveness and performance of Draco on commercial off-the-shelf (COTS) devices.

5.1 Effectiveness

In Section 3, we described attacks on the CVS Caremark and Job Search apps. To evaluate the effectiveness of Draco, we enhanced the apps with DPL rules as these are described in Section 4.2. We have installed the apps on a Nexus 5 phone running Android 6.0. We have also updated the WebView system app to a version enhanced with the Draco Runtime System. In both cases, we found that Draco successfully blocks all illegitimate access attempts by spurious domains. At the same time, the legitimate domains can function properly and the app remains fully functional.

5.2 Performance

As explained before, DRS consists of: 1) a static analysis unit for determining the permissions used by class methods and event handlers, 2) a parsing module that dissects the policy rules entered by the developer and permissions used by class and event handler methods and inserts them into our data structures, and 3) an enforcement unit that intercepts invocations on JS bridges and invocations made via HTML5 APIs to ensure that the origin making the request is granted the access right by the developer and to block the request otherwise, or to prompt the user for granting permissions. We will analyze the performance of each component separately. We conduct all of our experiments on a LG Google Nexus 5 smartphone, which runs Android 6.0 (Marshmallow) and is equipped with 2.26GHz quad-core Qualcomm Snapdragon 800 processor and 2GB RAM.

App decompilation and static analysis unit. We first require the decompilation of class files associated with Draconian policy rules into smali bytecode. Currently, we are performing this off-line (not on Android); however, there are existing tools that can decompile apps on the Android platform. For example, apktool [13] decompiles

Static Analysis cost	average (s)	standard deviation
small class (5 methods)	3.004	0.054
medium class (10 methods)	5.940	0.114
large class (15 methods)	8.766	0.167

Table 2: Runtime for static analysis on Nexus 5

all the resource and class files in the target app between approximately 60 to 90 seconds, depending on the size of the app. Another tool, DexDump [20], can perform app decompilation much more efficiently since it parses classes on demand. We hope to borrow from their techniques and use them in our future implementation.

After decompiling the target app, we statically analyze the classes associated with Draconian policy rules to determine which permissions are used by each exposed method. Table 2 shows the performance results of permission extraction of class methods for three cases: 1) a small class with 5 methods (506 smali instructions), 2) medium-sized class with 10 methods (1106 smali instructions), and 3) a larger class with 15 methods (1706 smali instructions). Here, the total number of smali instructions in methods is the dominating factor for the performance since for each instruction we perform a lookup in our data structure for PScout mappings.

Policy parsing module. The Parsing module runs each time the app is launched by the user in order to populate the in-memory policy and permission maps. As explained before, there are two subcomponents of the parsing module; the *policy parser*, which simply parses the policy rule inserted by the developer, and the *permission parser*, which parses the output of the static analysis to identify the permissions used by class methods and event handlers for the classes and event handlers declared in the DPL rules. Both of these components incur minimal overhead: in total, we identify the run time of the parsing module to be in the order of miliseconds as shown in Table 3. This cost is negligible compared to the launch time of Android apps, which is expected to be in the order of seconds [21].

• *Policy parsing.* The complexity of the inserted DPL rule can change the total run time of the parsing operation. In order to investigate how rule complexity affects performance, we have considered three types of rules: 1) a *simple* DPL rule that involves 5 class methods, 2) a *large* DPL rule that involves 15 class methods, and 3) a *semantically large* DPL rule that involves all class methods (i.e., uses *all* tag). Listing 5 shows how these rules can be added to the app. We only investigate the performance for the JavaScript interface channel; however, the results are comparable for other channels since the construction of the policy rules are similar. We run each experiment 10 times and report average run time and standard deviation. Table 3 shows the results for the three types of policy rules (case 1,2,3). First we observe that the parsing overhead, is negligible, in the order of a few miliseconds. Additionally, as the number of class methods increases (from 5 to 15), the run time slightly increases. This is simply because number of insertion operations performed is proportional to the number of class methods in the rule($\mathcal{O}(n)$). However, for a policy that addresses all class methods, the run time is even smaller than that of a simple policy. This is because *all* tag semantically means that all class methods are involved, and when it is used, we do not require insertion of all the class methods into the policy map one by one.

• *Permission parsing.* The performance of permission parsing is affected by the the number of permissions used by the app, and the number of sensitive API calls (i.e., required permissions for the sensitive API call) used in each method. It has been shown that on average Android apps use five permissions [22]. Therefore, we

Listing 5: Small, large, semantically large rules for JS interfaces

```
// small policy rule
mWebView.loadUrl("policyrule;allowjsinterface;
https://mydomain.com;GeoWebViewActivity$JsObject<classMethod1,
classMethod2,classMethod3,classMethod4,classMethod5>");

// large policy rule
mWebView.loadUrl("policyrule;allowjsinterface;
https://mydomain.com;GeoWebViewActivity$JsObject<classMethod1,
classMethod2,..., classMethod14, classMethod15>");

// (semantically) large policy rule
mWebView.loadUrl("policyrule;allowjsinterface;
https://mydomain.com;GeoWebViewActivity$JsObject<all>")
```

Parsing cost	average (ms)	standard deviation
(1) small policy	1.874	1.248
(2) large policy	2.453	0.811
(3) semantically large policy	1.633	0.847
(1) w/ small permission file	2.428	0.820
(1) w/ large permission file	8.434	2.269

Table 3: Runtime for policy rule parsing and insertion on Nexus 5

consider two cases: 1) a small permission file (created by the static analysis unit) for a class with five methods where each method uses five or less permissions, 2) a large permission file with 20 class methods and five permissions for each method. We consider the second case to be not very likely to occur and use it only as an upper bound on the performance for permission parsing. For both of these cases, we use a simple policy rule (including 5 methods), with the addition of the permission list that contains all the five permissions the app (without using *all* tag) grants to the given subject. Table 3 shows the results of policy parsing (which include permission parsing). As can be observed from the table, the run times are still in the order of milliseconds, with the total run time being higher for when the permission file that contains many methods that use all of the app permissions.

Enforcement. It is important for the enforcement to be efficient since this an action that is expected to be performed frequently during the lifetime of an app. Thus, any delay can affect the app's run time performance and degrade user experience. Here, we take a closer look at the performance of the enforcement unit for the JavaScript interface and HTML5 channels. We do not present our results for the event handler channel since they are intrinsically similar to those of the JavaScript interface channel. For the former cases, we report the average time it takes (and standard deviation) to allow and disallow an origin.

• *JavaScript interface channel.* We take the same approach as in our evaluation for policy parsing, and perform enforcement corresponding to small (5 methods), large (15 methods) and semantically large (all methods) policy rules. For each case, we assume the origin wants to access the method that is the last method in the provided method list so that we get an upper bound on the run time (since we perform linear search in vector that contains the methods associated with a policy rule). Table 4 shows the results for the JavaScript interface channel. Evidently, the enforcement overhead is negligible (in the order of microseconds).

• *HTML5 API channel.* For the HTML5 API channel, we consider two cases: 1) an access control decision is made solely by the system, and 2) the user is prompted to make a decision on the use of

Enforcement cost	average (ms)	standard deviation
small policy (allow)	0.356	0.260
small policy (block)	0.243	0.051
large policy (allow)	0.965	1.214
large policy (block)	0.551	0.124
semantically large policy (allow)	0.146	0.0252

Table 4: Runtime for enforcement on JavaScript interface channel on Nexus 5

Enforcement cost	average (ms)	standard deviation
system (allow)	0.282	0.093
system (block)	0.130	0.029
user (allow)	0.326	0.116
user (block)	0.286	0.076

Table 5: Runtime for enforcement on HTML5 channel on Nexus 5

permissions. Table 5 shows the time taken by the system for both of these cases for the HTML5 API channel. We do not show the time the user takes to grant or revoke access to permission-protected resources. Naturally, this will be at least in the order of seconds with a large variation, and is many orders of magnitude larger than the purely-system based access control decision. Again we observe sub-millisecond delays highlighting the efficiency of DRS enforcement.

6. RELATED WORK

Previous work has discussed the problem that foreign code governs the same privileges as the host application in different contexts.

Protecting against third-party libraries and other inter-module threats. Third-party libraries governing the same permissions as the host app has been shown to be problematic by the previous work. Working towards solving this issue, AdSplit [23] suggests separation of ad components from the core app and running them in their own processes for protecting against the malicious activities that can be performed by potentially-malicious ad libraries. In [24], the authors discuss the vulnerabilities due to the nonexistence of origin-based protection on the Android system. More specifically, they show that third-party libraries make host apps vulnerable to cross-origin attacks on the app-to-app channels such as intent and the scheme mechanism. Their solution, Morbs, gives developers a means to express new policies about how two apps can communicate, and it labels messages between apps with their origins so that the developer-written permissions can be enforced at run time. One short-falling of this solution is that it works for only app-to-app communication channels. FlexDroid [25] gives developers a way of creating fine-grained access control policies on the system resources for third-party libraries based on Android permissions. To enforce the policies, they examine the Dalvik call stack at run time to identify the origin of the call and its associated permissions. Case [26] takes another approach and instruments apps with a module that can mediate access between the submodules (which can even be in the granularity of a Java class) of the app. These solutions are not limited to only app-to-app channels as Morbs and can protect an app against inter-module threats; however, they do not provide protection against arbitrary foreign content that can be loaded within a single in-app module (e.g., via web containers).

Analysis, attacks, and defenses for WebViews. Vulnerability of WebViews has been extensively discussed by previous work [7, 8, 1, 9, 10]. In [7], the authors present several classes of attacks that can be launched against apps that use WebViews. Chin et al. present a static analysis tool that can identify whether an app is vulnerable to WebView attacks [8]. Mutchler et al. present a large-scale analysis on mobile web applications, and present the trend of vulnerabilities in these applications. None of these work implement any defense mechanism targeting WebViews [1]. In [27], the authors present an access control mechanism for WebViews. Their approach uses static analysis to identify the use of security-sensitive APIs in the exposed Java class, and notifies the user if any such use is found. The user is then prompted to allow or completely block the binding of the Java object. The main drawback of this approach is that after the user allows the binding, they do not provide any origin-based access control, so all the origins still have the same access rights. Additionally, their focus is only on the permission-protected resources.

WebView-related attacks on hybrid frameworks and bringing origin-based access control. Georgiev et al. discuss the nonexistence of origin-based access control in hybrid frameworks and propose a capability-based approach (NoFrak), where app developer whitelists origins that are allowed to access system resources [2]. The drawback of their approach is that it works only for the Phonegap framework even though the aforementioned problem is not even specific to hybrid frameworks. Additionally, the solution is not fine-grained since a whitelisted origin get access to *all* resources of the app. In [3], the authors propose fine-grained access control system for hybrid apps, which allows developers to add origin permissions to the manifest file and associate iframes with permissions, and enforces the developer rules in the operating system. One drawback of this solution is that the web developer has to be compliant and include the permission tag along with the desired permissions in the iframes; otherwise, the frame just governs all the permissions the main page is given to. Furthermore, even though this solution provides a more fine-grained access control than NoFrak, it focuses on only protecting permission-protected resources, and hence is not enough to fully protect the app and its user as we have previously shown. Moreover, neither of these solutions give developers the flexibility to consult with the user on how to handle requests. In [28], the authors present code injection attacks on hybrid apps. Even though they mainly target hybrid frameworks, the attack shown can be applied to all mobile web applications in general.

Fixing Web-based system apps. Georgiev et al. show that Web-based system applications also suffer from similar problems, and introduce POWERGATE, which provides access control on native objects in the system by enforcing the policy rules created by the developer [29]. Here, their focus is on native-access APIs provided to the application by the platform, and not on the resources exposed by the use of JavaScript bridges.

7. CONCLUSION AND FUTURE WORK

In this work, we investigate the understudied JavaScript bridge vulnerabilities for native mobile web applications that use embedded web browsers (WebView) to show content. We show cases where highly-downloaded vulnerable Android apps inadvertently expose their internal resources to untrusted web code. By investigating the use of WebView APIs by app developers, we identify the need for a unified and fine-grained access control mechanism on WebView. Hence, we propose Draco, a unified access control framework that allows developers to declare access rules for the exposed resources with fine granularity and enforces these access policies at runtime. Draco's declarative policy language can be used by app developers to create policy rules that specify their trusted or semi-trusted origins with capabilities defining their access coverage on the three access channels (JavaScript inteface, event handlers, HTML5). Draco Runtime System then enforces these policy rules in an effective and efficient manner. This approach also saves developers from implementing burdensome programming measures (i.e., navigation control, multiple WebViews with different levels of exposure) in an attempt to prevent exposed resources from web domains. Draco is easily deployable since it does not require Android OS modifications, but only enhancements in the Android System WebView app. In future work, we plan to investiage the use of server credentials for authorization, and explore efficient infrastructures for credential management, credential distribution and revocation.

Acknowledgments. This work was supported in part by NSF CNS 12-23967, 14-08944, and 15-13939. The views expressed are those of the authors only.

8. REFERENCES

[1] P. Mutchler, A. Doupé, Kruegel C. Mitchell, J., and G. Vigna. A large-scale study of mobile web app security. In *MoST*, 2015.
[2] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS*, 2014.
[3] X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in android. In *Information Security*. 2015.
[4] Webkit: Open source web browser engine. https://webkit.org/.
[5] The chromium project. https://chromium.org/.
[6] Android open source project. https://source.android.com/.
[7] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*. ACM, 2011.
[8] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications*. 2013.
[9] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: Webview exploitation. In *LEET*, 2013.
[10] D. Thomas, A. Beresford, T.s Coudray, T. Sutcliffe, and A. Taylor. The lifetime of android api vulnerabilities: case study on the javascript-to-java interface. In *Security Protocols XXIII*. 2015.
[11] Dex2jar. https://github.com/pxb1988/dex2jar.
[12] Jd-gui. http://jd.benow.ca/.
[13] Apktool decompiler. http://ibotpeaches.github.io/Apktool/.
[14] D. McCracken and E. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*.
[15] Eddystone ble beacons. http://bit.ly/1WMaylQ.
[16] Bluetooth low energy. http://bit.ly/1Rw9grs.
[17] 15 companies using beacon technology. http://bit.ly/16qwASy.
[18] The apktool's failed app list. http://bit.ly/2aUyE9T.
[19] K. Au, Y. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *CCS*, 2012.
[20] Dexdump. http://bit.ly/1NBg7QM.
[21] Cold start times: Analysis of top apps. http://bit.ly/1TFTtb0.
[22] Key takeaways for mobile apps. http://pewrsr.ch/1M4LqyY.
[23] S. Shekhar, M. Dietz, and D. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX*, 2012.
[24] R. Wang and X.and Chen S. Xing, L.and Wang. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS*, 2013.
[25] J. Seo, D Kim, D Cho, T. Kim, and I. Shin. Flexdroid: Enforcing in-app privilege separation in android. 2016.
[26] S. Zhu, L. Lu, and K. Singh. Case: Comprehensive application security enforcement on cots mobile devices. In *MobiSys*, 2016.
[27] Y. Jing and T. Yamauchi. Access control to prevent malicious javascript code exploiting vulnerabilities of webview in android os. *IEICE TRANSACTIONS on Information and Systems*, 2015.
[28] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *CCS*, 2014.
[29] M. Georgiev, S. Jana, and V. Shmatikov. Rethinking security of web-based system applications. In *WWW*, 2015.