Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation

Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin
and Gautam Nagesh Peri
Department of Electrical Engineering & Computer Science, Syracuse University,
Syracuse, New York, USA
{xjin05, xhu31, kying, wedu, heyin, nperi}@syr.edu

ABSTRACT

Due to the portability advantage, HTML5-based mobile apps are getting more and more popular. Unfortunately, the web technology used by HTML5-based mobile apps has a dangerous feature, which allows data and code to be mixed together, making code injection attacks possible. In this paper, we have conducted a systematic study on this risk in HTML5-based mobile apps. We found a new form of code injection attack, which inherits the fundamental cause of Cross-Site Scripting attack (XSS), but it uses many more channels to inject code than XSS. These channels, unique to mobile devices, include Contact, SMS, Barcode, MP3, etc. To assess the prevalence of the code injection vulnerability in HTML5-based mobile apps, we have developed a vulnerability detection tool to analyze 15,510 PhoneGap apps collected from Google Play. 478 apps are flagged as vulnerable, with only 2.30% false-positive rate. We have also implemented a prototype called NoInjection as a Patch to PhoneGap in Android to defend against the attack.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection; D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Code inspections and walk-throughs*

General Terms

Security

Keywords

HTML5-based Mobile Application; Code Injection; Static Analysis

1. INTRODUCTION

Smartphones have grown to become very popular over the years. A report from comScore [19] in February 2013 shows that smartphones surpass 50 percent penetration and break into "late majority" of adopters. All these facts have led many developers to switch to develop mobile applications (apps). However, to support different platforms, developers may need to develop different versions of apps based on the language used in the system, such as Java in Android and Object C in iOS. Using HTML5-based techniques to develop mobile apps provides a good solution to overcome this limitation. Unlike native apps, these apps are built on standard web technologies such as HTML5, CSS and JavaScript, which are universally supported by all mainstream mobile systems. Porting such apps from one platform to another is greatly simplified [10, 33]. With the increasing support for HTML5, HTML5-based mobile apps are becoming more and more popular [7, 9, 17]. A survey from Evans Data shows that among the 1,200 surveyed developers, 75% are using HTML5 for app development [1].

Unfortunately, the web technology has a dangerous feature: it allows data and code to be mixed together, i.e., when a string containing both data and code is processed by the web technology, the code can be identified and sent to the JavaScript engine for execution. This feature is made by design, so JavaScript code can be embedded freely inside HTML pages. This feature has led to the widespread code injection attack, called Cross-Site Scripting (XSS), on web applications. Built upon the same technology as web applications, HTML5-based mobile apps are subject to the similar code injection attacks.

However, we have found out that code injection attacks against HTML5-based mobile apps are significantly different from the XSS attack in terms of code injection channels: Web applications only have one channel for code injection; that is through the web server (or web site), which is why it is called "cross-site". HTML5-based apps have many channels for code injection, because mobile apps interact with other entities via many data channels, including barcode, SMS, file system, Contact, Wi-Fi, NFC, etc. *We have found that all these channels can be used for attacks. More specifically, we have found that a vulnerable HTML5-based app can be compromised by simply scanning a 2D barcode, reading data from Contact list, displaying information from MP3 music, or even scanning for Wi-Fi access points. These attacks affect all major mobile platforms, including Android, iOS, Windows Phone, etc.* We have conducted a systematic study to identify all the potential attack channels. We have built demonstration videos to show the feasibility of the attacks [5].

To evaluate the prevalence of this vulnerability in Android apps, we have developed a detection tool to conduct large-scale static analysis on Android apps. An app is vulnerable when it reads from a code injection channel and renders the input using an unsafe API. We transform the detection problem into an equivalent data-flow analysis problem that seeks to identify a data flow from a code injection channel to an unsafe rendering API. Our tool first models the APIs reading data from code injection channels and then con-

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660275 .

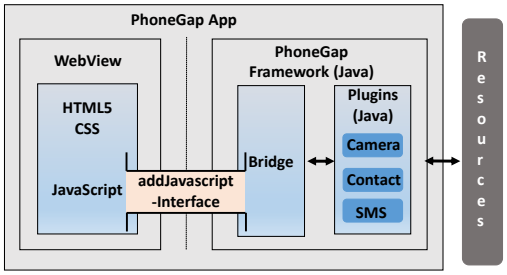


Figure 1: PhoneGap Architecture

structs a JavaScript program slice that is relevant to reading and processing the data from each of the code injection channels. Finally it performs static taint analysis on the slice to detect the presence of a dangerous information flow from the code injection channel to an unsafe rendering API.

We ran our detection tool on 15,510 PhoneGap apps (PhoneGap is the popular framework that is used to develop HTML5-based mobile apps) downloaded from Google Play. The average execution time of our tool on an app is 15.38 seconds, which is fast enough to process a large volume of apps. Among all tested apps, 478 were found to have code injection flaws. Our manual verification shows a false-positive rate of 2.30%. We further investigated these vulnerable apps, and found that many of these apps have access to users' private information or can conduct privileged operations, such as sending SMS. Once malicious code is injected into these apps, great damage can take place. We have been actively reporting our findings to the vulnerable app developers. Some developers have taken our reports seriously and fixed the apps by following our suggestions.

Our work makes three folds of contributions. First, we have found a new class of code injection attacks on mobile systems. We have conducted a systematic study of this attack. Second, we have developed a detection tool that can automatically scan HTML5-based apps to identify potential code injection flaws. Third, we have also conducted a systematic study on different mitigation techniques and implemented a prototype called NoInjection as a patch to the PhoneGap framework in Android to address the problem.

The rest of the paper is organized as follows: Section 2 gives a brief overview of WebView and the PhoneGap framework. Section 3 explains how the attack works. Section 4 talks about the detection tool. Section 5 discusses potential mitigation and presents our prototype to address the attack. Related works are surveyed in Section 6 and the paper is summarized in Section 7.

2. BACKGROUND

Most popular mobile OSes, such as Android, iOS, and Windows Phone, do not support JavaScript and HTML natively, so in order to display HTML5-based user interface and execute JavaScript code, an application needs to embed a web browser component. This component is called WebView in Android, UIWebView in iOS, and WebBrowser in Windows Phone. Without loss of generality, we refer to WebView in the context of Android throughout the paper.

Overview of WebView. WebView is an essential component in mobile platforms, enabling smartphone and tablet apps to embed a simple but powerful browser inside them. Since WebView is designed to display web contents, which usually come from external sources and are not trusted, a sandbox is implemented inside WebView. This sandbox basically isolates the JavaScript inside WebView from the system, so JavaScript cannot access the system resources, such as files, device sensors, cameras, etc. Such a sandbox is appropriate for web contents running inside a browser, but it will be too restrictive for mobile applications.

WebView provides an API `addJavascriptInterface()`, which allows an app to add a bridge between JavaScript code inside and native Java code outside. In this way, once the app has the required permissions, JavaScript inside WebView can access mobile resources by invoking the outside native code, which is not restricted by WebView's sandbox. Developers can write their own native code, but that would reduce the portability. The most common practice is to use a third-party middleware for the native-code part, leaving the portability issue to the developers of the middleware. Several middlewares have been developed, including PhoneGap [14], RhoMobile [16], AppMobi [3], Mosync [12], Appcelerator [2], etc. Due to the page limitation, we only provide a brief introduction on PhoneGap, which is the most popular framework used to develop HTML5-based mobile apps. Our studies also focus on PhoneGap apps. However, our attack, detection, and mitigation can also be applied to other frameworks.

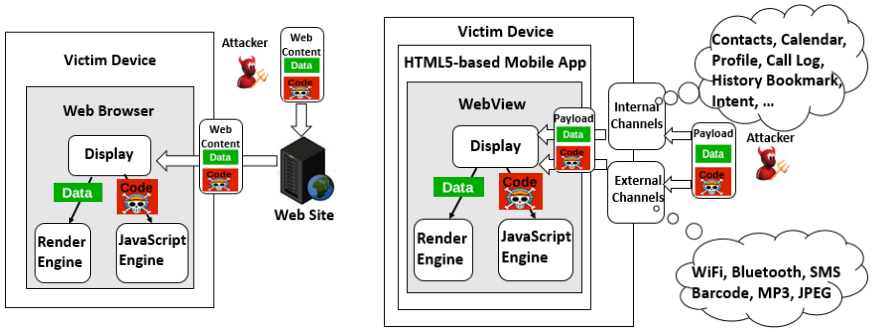
PhoneGap. PhoneGap is a middleware framework that can be used to create mobile applications using the web technologies, including HTML5, CSS and JavaScript. Applications developed using the PhoneGap framework can be easily ported from one mobile platform to another, as long as the second platform is supported by PhoneGap. The PhoneGap framework consists of two parts: the bridge part and the plugin part. The bridge part connects JavaScript code and Java code (native) through an interface called "cordova"; JavaScript code inside WebView can invoke the Java code outside using this interface. The plugin part is used to directly access different types of resources, such as Camera, SMS, Contacts, etc. PhoneGap provides 16 official plugins, but if an app's needs cannot be met by these plugins, developers can either write their own plugins or use third-party PhoneGap plugins.

The plugins are mainly written in the native language, so they are not restricted by the WebView's sandbox. However, to make it more convenient to use, many plugins provide companion JavaScript libraries. When the app's JavaScript code needs to access system resources, it calls the API provided in the plugin JavaScript libraries, which will then use the "cordova" interface to invoke the Java code inside bridge, and eventually cause the invocation of the corresponding plugin. When the plugin finishes its job, it will return the result back to the page, again through the PhoneGap bridge. That is how JavaScript code inside WebView's sandbox can access system resources. Figure 1 shows the PhoneGap architecture in Android.

3. CODE INJECTION ATTACK

Writing mobile applications using the HTML5-based web technology makes applications portable across different mobile platforms. However, it is well known that the web technology has a dangerous feature: it allows data and code to be mixed together, i.e., when a string containing both data and code is processed by the web technology, the code can be identified and sent to the JavaScript engine for execution. This feature is made by design, so JavaScript code can be embedded freely inside HTML pages. Unfortunately, the consequence of this feature is that if developers are not careful, unexpected code inside data can be automatically and mistakenly triggered. If such a data-and-code mixture comes from an untrustworthy place, malicious code can be injected and executed inside the victim application. This is exactly how the Cross-Site Scripting (XSS) attack works.

In a typical XSS attack, attackers insert JavaScript code into the data field (such as a form). If the server does not stripe off the in-



(a) XSS attack on Web Applications (b) Code Injection Attack on HTML5-based Apps

Figure 2: Code Injection Attacks on Web Applications and HTML5-based Mobile Apps

serted code, when this piece of code-mixed data is displayed to the victim, the injected JavaScript code will be executed on the victim's browser, and of course with the victim's privilege. XSS currently ranks as the No. 3 most popular web attack on the OWASP 2013 Top-Ten list [13].

Built upon the same technology as web applications, HTML5-based mobile apps have inherited the XSS vulnerability. However, as we will show in this paper, this is not a simple extension of XSS attacks: HTML5-based mobile apps have a much broader attack surface than that for web applications.

In XSS attacks, to achieve the damage, the malicious JavaScript code has to reach and run from the victim's browser. Since web applications only interact with web servers (i.e., web site), there is no direct way for attackers to interact with the victim. Therefore, to inject malicious JavaScript code into the victim's browser, attackers have to use the site for their code to reach the victim's browser. That is why the attack is called "cross-site". Here, "site" is the code injection channel, the only injection channel. Figure 2(a) depicts the XSS attack.

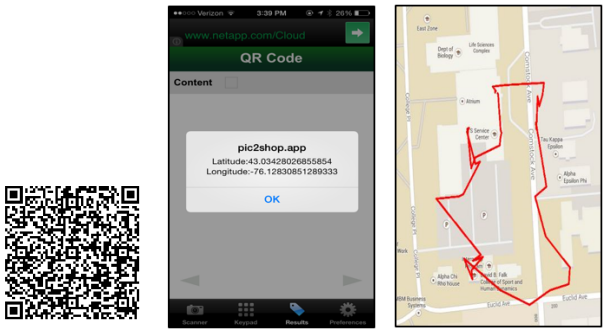
HTML5-based mobile apps have a much broader attack surface. Unlike web applications, HTML5-based apps are like other types of mobile apps: they are supposed to interact with many forms of entities, such as other apps, 2D barcode, Wi-Fi access points, other mobile devices, data sent by others or downloaded from external resources, etc. Each of these interaction points is a potential attack surface. Consequently, attackers are not limited to the "site" channel anymore. This is a major difference between XSS and the code injection attack on HTML5-based mobile apps. Figure 2(b) illustrates the basic idea of the attack.

Moreover, compared to XSS attacks, attacks on HTML5-based apps can cause much greater damage. In XSS, the web browser is sandboxed, so the injected JavaScript code cannot freely access local resources. In mobile apps, the injected code can do more damage through the interface provided by the middleware.

In the rest of this section, we first use one special channel (the camera channel) to illustrate how HTML5-based apps can be attacked using the XSS-like code injection technique. After that, we provide a systematic study to analyze all the possible channels that can be used for code injection; we will also discuss the conditions that make an HTML5-based app vulnerable and the damage that can be achieved by attackers.

3.1 The Attack

To demonstrate how the XSS-like code injection works against HTML5-based apps, we use a real app as an example. This app is called `pic2shop` (`http://www.pic2shop.com/`), and it is a quite popular barcode scanner app based on the HTML5 technology. The app runs in Android, iOS, and Windows Phone; it is vulnerable in all three platforms. In Android, the number of downloads for this app is in the range of 100,000 to 500,000. We do not have the download data for the other two platforms, but in iOS, the app has received 45,416 reviews for all its versions.



(a) QR Code (b) Attacked (c) Being tracked

Figure 3: An attack example

We made a QR code (Figure 3(a)), but it is not a typical one; it contains the following HTML tag and JavaScript code:

```
1  <img src=x onerror=
2  navigator.geolocation.watchPosition(
3  function(loc){
4    m='Latitude:'+loc.coords.latitude+
5    '\n'+'Longitude:'+loc.coords.longitude;
6    alert(m);
7    b=document.createElement('img');
8    b.src='http://128.***.213.66:5556?c='+m })>
```

The above code uses `Geolocation.watchPosition()` to steal the device's geolocation. The API registers a handler function that will be called automatically each time the position of the device changes. From the code, we can see that when the handler function is invoked, the location information is stored in the variable `loc`, and displayed at Line 6. At Lines 7 and 8, `loc`'s content is sent to an outside computer [1].

We scan the QR code using `pic2shop`. Immediately, a window pops up, displaying the user's current GPS location (Figure 3(b)). This indicates that the JavaScript code in the QR code has been injected into the device and successfully triggered. We then put the phone in the pocket, and took a walk in the campus. The phone kept sending its locations back to the "attacker", who simply plotted them on Google Map (Figure 3(c)).

[1] In this QR code, we intentionally used an invalid IP address `128.***.213.66` to ensure that the location will never be sent out even if readers scan the QR code using a vulnerable app.

From this example, it is not difficult to see how the code gets into the victim's device, but it is not clear how the code gets triggered. There is another critical condition for the above attack to work: the QR code needs to be displayed by the app. If app developers are not careful and choose a wrong way to display the QR code, they may end up executing the JavaScript code inside the QR code. This is exactly the case in `pic2shop`.

Let us summarize the key characteristics of the vulnerability exploited by the above attack. For an HTML5-based app to be vulnerable to the XSS-like attack, it needs to satisfy the following two conditions:

- The app needs to use a channel to get data from outside (outside of the app or device). In Section 3.2, we present a systematic study of these potential channels.
- The data from outside need to be displayed inside the HTML5 page. There are many ways to display data; some are safe, and some are not. Section 3.3 will discuss this in details.

3.2 Code Injection Channels

Any channel that mobile apps use for getting data from outside of the program can be used for code injection, and some channels are more obvious than the other. We have conducted a systematic study on these channels, and have confirmed that they can indeed be used for the attack. We divide them into two categories: the external channel and the internal channel, referring to whether the data come from outside of the device or from inside.

3.2.1 External Code Injection Channels

Just like other apps, HTML5-based mobile apps need to interact with the outside world, including the environment, users, other devices, etc. These interaction channels, intended for data, can be used for code injection. We can further divide them into three categories. The web channel is not included, because that is the channel used by the traditional XSS attacks.

Data Channels Unique to Mobile Devices. Other than getting data from the Internet, Wi-Fi, and Bluetooth, mobile devices also get data from many channels that are not very common in traditional computers. For example, most smartphones can scan `2D barcodes` (using camera), receive `SMS messages`, and some smartphones can read `RFID tags (NFC)`. These data channels make it very convenient for users to get information from outside, so they are being widely used by mobile applications. In our studies, we find out that if these mobile applications are developed using the HTML5-based technology, all these data channels can be used for injecting code. Malicious JavaScript code can be embedded in the contents of 2D barcodes, RFID tags and SMS messages. See our attack demo using SMS and 2D barcode in [5].

Metadata Channels. A very popular type of apps on mobile devices is the media-play apps, such as audio players, video players, and picture viewers. The main functionality of these apps is to view media files, which are often downloaded from the Internet or shared among friends. Since they mostly contain audio, video, and images, it does not seem that they can be used to carry JavaScript code. However, most of these files have additional fields called metadata, and they are good candidates for code injection. `MP3`, `MP4`, and `JPEG files` are standard formats for multimedia files, and they all contain metadata fields, such as title, artist, album, etc. Attacker can inject malicious code into these metadata fields. When these metadata are displayed—as they often are—in vulnerable HTML5-based apps, the malicious code will get executed. See our attack demo using MP3 in [5].

ID Channels. This type of channels is less obvious. When a mobile device needs to establish a connection (e.g. Wi-Fi or Bluetooth) with an external entity, it first conducts a scan to find the IDs from the nearby devices. These IDs are often displayed to the users, so they can choose the right one to connect. These IDs, provided by the untrusted external entity, can be used as a code injection channel, so code injection can happen before the connection is actually established. In other words, by simply scanning for Wi-Fi access points or Bluetooth devices, an HTML5-based app can be attacked.

To launch the attack, attackers can configure an Android phone so it functions as a WiFi access point; they then embed their malicious JavaScript code in the SSID of the access point. When an HTML5-based app scans for Wi-Fi access points, the SSID will often be displayed to the user, so the JavaScript code inside the SSID can be potentially triggered. Similarly, the attacker can also turn a mobile device into a Bluetooth device, embed a malicious JavaScript code in its device name, and broadcast the name to nearby devices. Any mobile device that is trying to pair with a Bluetooth device (not necessarily the attacker's one) can be potentially affected.

This type of channels does pose a challenge because of their size limitation. For example, SSID is limited to 32 bytes, which is not enough to include a meaningful piece of JavaScript code. To resolve this issue, attackers can break up the malicious JavaScript code into multiple pieces, each within the 32-byte limit. For example, if the entire code consists of "P_1 P_2 P_3", it can be broken into the following four pieces: "`A=`P_1", "`B=`P_2", "`C=`P_3", and "`eval(A+B+C)`". The attacker can periodically change the value of the SSID to one of these four pieces. If somebody is scanning, he/she will end up getting all the four pieces. The execution of these four pieces is equivalent to executing "P_1 P_2 P_3". See our attack demo using Wi-Fi access points in [5].

3.2.2 Internal Channels

In addition to interacting with the outside world, mobile apps often interact with other apps on the same device. Therefore, if a malicious app is installed on the victim's device but its privilege is limited, this app can inject malicious JavaScript code into vulnerable HTML5-based apps that have more privileges, hence achieving privilege escalation. We have conducted a systematic study on these internal channels, and we divide them into three categories (our discussion only focuses on Android, but similar consideration can be made to iOS and Windows Phone).

Content Provider. *Content Provider* is typically used by Android apps to store data that can be shared among apps. Because of its data-sharing nature, Content Providers are ideal candidates for code injection. A typical Android system usually comes with the following Content Providers: `Contact`, `Calendar`, `User Dictionary`, `Call Log`, `Browser`, `Sync Adapter`, and `Profile`. Take `Calendar` as an example. The attacker can inject malicious JavaScript code into a Calendar event; when a vulnerable HTML5-based app tries to display this Calendar event, the malicious code may be triggered.

File System. Many apps also directly interact with the file system, including the SD card that is a public space for apps to store data, and these data are accessible to other apps. Therefore, file system is a fertile ground for code injection. Obviously, code can be placed into the file content, so if a victim HTML5-based app tries to display the content, the code may be triggered. Furthermore, we have confirmed that JavaScript code can also be placed in file names, so if a vulnerable HTML5-based app just tries to display the file names, the code can be triggered. If the size of the name is

not enough, we can use the same technique as the one used for access points, i.e., putting the code into multiple file names. It should be noted that this attack can also be launched by an external party, i.e., the file that has JavaScript code in its name can be downloaded from an external site owned by attackers.

Intent. Another way for an Android app to interact with other apps is to use *Intent*. Through intent, an app can pass data to other apps. This can obviously be used for code injection: the attacking app can place malicious JavaScript code in the intent, which then triggers the target app. If the target HTML5-based app tries to display the data included in the intent, the JavaScript code may be triggered. We have confirmed this attack in our study.

3.3 Triggering Injected Code

For the injected code to be triggered, the data that embed the code need to be displayed. In the traditional XSS attack against web applications, the injected code, mistakenly treated as data, is already placed (by the server) in a web page, so when the page is displayed, the code will be triggered.

In the attack against HTML5-based apps, the code injected through data channels has not yet become part of the HTML page; it needs some "help" from the victim app. This "help" is achieved when the victim app displays the data coming from those channels. There are many ways to display data inside an HTML page. A typical way is to use the DOM (Document Object Model) display APIs and attributes, such as `document.write()`, `innerHTML` (attribute), etc. Many apps also use jQuery APIs, such as `html()` and `append()`. These APIs eventually call or use the DOM display APIs and attributes.

Selecting which APIs to use is very critical, because some of these APIs/attributes are safe to use without causing the code to be executed, but a number of APIs/attributes are not safe: if there is JavaScript code inside the data, the code will not be displayed, but will instead be sent to the JavaScript engine to be executed.

To understand how HTML5-based apps use various APIs and attributes to display information, we downloaded 15,510 HTML5-based (PhoneGap) apps from the Google Play (we thank Martin Georgiev and Vitaly Shmatikov [27] for providing us with the list of the apps). We wrote a tool to study how DOM/jQuery APIs or attributes are used to display data. They are categorized as *safe* and *unsafe*. Unsafe means injection code will be executed, safe means the otherwise. In Table 1, the column "`Occurrence Percentage`" shows how often a particular API is used among all the usage cases (an app can use the same API for many times, and each time counts as one occurrence). The column "`App Percentage`" shows the percentage of the apps that use a particular API for at least once.

From the table, we can see that the use of unsafe APIs/attributes is pervasive: 53% of the usages are unsafe. The unsafe `innerHTML` attribute alone is used by 91% of apps. If we look at the apps that use at least one unsafe APIs/attributes at least one time, the percentage becomes 93%.

There are other ways to display data in HTML5-based apps. Similarly, some are safe and some are not. We list some examples here; a full-scale study is outside the scope of this paper and will be pursued in our follow-up work. (1) jQuery is the most common JavaScript library, but there are other JavaScript libraries, such as MooTools [11], Prototype [15], etc. They come with their own displaying APIs. (2) We have seen apps that directly use `WebView`'s Java APIs to inject data into WebView for displaying purposes. `WebView.loadDataWithBaseURL()` is an example of such APIs. This approach is not safe.

DOM APIs & Attributes	Safe or Unsafe	Occurrence Percentage	App Percentage
document.write()	✗	0.79%	12.95%
document.writeln()	✗	2.27%	2.94%
innerHTML	✗	14.22%	90.90%
outerHTML	✗	1.55%	54.41%
innerText	✓	2.15%	62.01%
outerText	✓	0.003%	0.13%
textContent	✓	3.50%	65.97%
value	✓	14.43 %	83.11%
jQuery APIs			
html()	✗	14.02%	66.42%
append()	✗	15.67%	71.04%
prepend()	✗	1.14%	22.36%
before()	✗	1.17%	54.88%
after()	✗	0.06%	14.89%
replaceAll()	✗	1.68%	56.78%
replaceWith()	✗	0.01%	0.48%
text()	✓	14.78%	62.05%
val()	✓	11.95%	62.82%

Table 1: APIs and Attributes used for displaying data. (✓ means they are safe against code injection; ✗ means unsafe.)

3.4 Achieving Damage

Because mobile apps have more privileges than web applications, the damage caused by the code injection attack on HTML5-based mobile apps is more severe than those caused by the XSS attack on web applications. We summarize the potential damage here.

First, the injected malicious code can directly attack the device through the "windows" that are opened to the code inside WebView. Normally, JavaScript code cannot do much damage to the device due to WebView's sandbox, but to enable mobile apps to access the system and device, many "windows" have been created. These "windows" include the HTML5 APIs (such as the Geolocation API) and all the PhoneGap plugins that are installed in this app. PhoneGap provides 16 plugins, including `Contact`, `File` `Device` plugins, etc.; they allow the malicious code to access system resources. Before Version 3.0 all these plugins are built into the framework, which means even if an app does not use them, they are always available to the app and can be used by the injected malicious code. [2] Besides these traditional plugins, many PhoneGap apps also include additional third-party plugins, such as `Barcode` `Scanner`, `SMS`, and `Facebook` plugins. These plugins can also be used by the malicious code.

Second, the injected malicious code can be further injected into other vulnerable PhoneGap apps on the same device using the internal data channels. Data sharing among apps is quite common in mobile devices. For example, the Contact list is shared, so when an app is compromised by an external attacker via the attack, the malicious code can inject a copy of itself into the Contact list. When another vulnerable PhoneGap app tries to display the Contact entry that contains the malicious code, the code will be triggered, and this time, inside the second app.

Third, the injected malicious code can turn the compromised device into an attacking device, so it can use the same attacking technique to inject a copy of itself into another device. For example, if the compromised app has the permission to send SMS messages, the malicious code can create an SMS message containing a copy of itself, and send to all the friends on the Contact list; it can also add the code in the metadata field of an MP3 file, and share the

[2]PhoneGap was only recently updated to Version 3.0 on October 17th, 2013, so most of the existing PhoneGap apps are using Version 3.0 or earlier versions.

file with friends; it can also pretend to be a Bluetooth device with malicious code set in the name field, waiting for other devices to display the name inside their vulnerable apps. The more PhoneGap apps are installed on devices, the more successful the propagation can be, and the more rapidly the malicious code can spread out.

4. VULNERABILITY DETECTION

We would like to automatically scan HTML5-based mobile apps that are vulnerable to code injection attacks, so the discovered vulnerable apps can be reported to and fixed by the developers. It is well known that dynamic testing has limited code coverage. Therefore, we need to perform static analysis to discover viable program execution paths for launching code injection attacks.

According to the attack characterization described in Section 3, the problem of detecting this code injection vulnerability is equivalent to a data-flow analysis problem. Data coming from untrusted channels, through certain data propagation, is then used in unsafe APIs.

4.1 Vulnerable App Example

To help describe our static analysis technique, we present an example in Figure 4. The sample code shows a normal HTML5-based app that uses the PhoneGap barcode plugin to display the scanned barcode. When the activity starts, the app listens to the `deviceready` event and calls `onDeviceReady` (Line 2). The `OnDeviceReady` function invokes the `barcodeScanner` plugin and displays the result on a DOM element called `"display"`. An attacker can embed malicious code in a QR code, so when the victim uses this app to scan the QR code, the malicious code can get executed inside the app.

```
1  document.addEventListener("deviceready",
       onDeviceReady, false);
2  function onDeviceReady() {
3    window.plugins.barcodeScanner.scan(0,onSuccess,
       onError);
4  }
5  function onSuccess(result) {
6    $("#display").html(result.text);
7  }
8  function onError(contactError) {
9    alert('onError!');
10 }
```

Figure 4: A vulnerable app example

4.2 Challenges

We face several unique challenges in analyzing HTML5-based mobile apps.

C1: Mixture of application and framework code. An HTML-based app consists of its own HTML pages and JavaScript code, and a large body of JavaScript libraries, Java code, and native code that belong to the framework (e.g., PhoneGap [14]). It is too expensive and nearly infeasible to analyze all these different code modules in different programming languages as a whole. A realistic solution is to properly model the interface of the framework code and concentrate on the JavaScript code and HTML pages that belong to the app.

C2: Difficulties in static analysis on JavaScript. It is well recognized that static analysis on JavaScript code is difficult, due to its dynamic nature, including prototype-chain property lookups, lexical-scoping rules for variable resolution, reflective property accesses, function pointers, and the fact that the properties and prototype chain of any object can be modified [44]. Furthermore, JavaScript code in HTML5-based apps is often written in an asynchronous manner, as shown in the vulnerable app example. It is surprisingly difficult to even construct a complete call graph and identify all the entry points.

C3: Dynamically loaded content. Some HTML5-based apps dynamically load pages and JavaScript code from remote servers. Therefore, if our static analysis only focuses on the pages and the JavaScript code that come with the app installation packages, we will not be able to detect vulnerabilities in these dynamically loaded content. To address this issue, we would have to run these apps, such that the dynamically loaded content can be retrieved and included into our static analysis.

4.3 Our Solution

Given a PhoneGap-based mobile app, our vulnerability discovery takes the following steps. We first perform rewriting on the app's JavaScript code to properly model the APIs provided by the PhoneGap framework. In this way, we can concentrate our analysis target on the app's own JavaScript code. We then construct a JavaScript program slice that is relevant to reading untrusted input from PhoneGap APIs and further processing the input. In this step, we aim to overcome the challenges in entry point identification and asynchronous calls. In addition, we can further narrow down our analysis target to a much smaller body of code. In the end, we perform static taint analysis on the slice to detect dangerous information flow from the untrusted input to one of the unsafe APIs. We developed a tool based on WALA [18] in about 2200 lines of Java code.

```
1  cordova = {
2        exec:function exec(suc, error,
             pluginName, operator, args){
3              var channeldata = "fakedInput";
4              suc(channeldata);
5  }}
6  window = { plugins:{ barcodeScanner:{
7       scan: function scan(mode,win,err){
8         cordova.exec(win, err,
              "BarcodeScanner", "scan", [mode]);
9  }}}}
10 document = {
11       addEventListener:function
             addEventListener(evt, handler,
             capture){
12       handler();
13 }}
```

Figure 5: Modeling APIs in cordova library

Framework modeling. We model the framework APIs using small code snippets; Figure 5 presents some of them. The JavaScript API `cordova.exec` is implemented in the native code, and when the native execution is accomplished, it will either call `suc` callback when the operation is successful, or call `error` when a failure is encountered. We model this function as it would call these two callbacks immediately. In addition, we also model the APIs that introduce inputs from untrusted channels. For example, the API `barcodeScanner.scan` is used to read barcode. We add a small code snippet for it, so it calls `cordova.exec` and returns a barcode input. Similarly, we model the APIs for the other channels. Table 2 presents a list of APIs that we model. In addition, we also model the API `addEventListener`, which is used to register an event callback. The PhoneGap framework overrides the implementation of this API to support registering the events defined within the framework. We model this API, such that the event handler (or

callback) is invoked immediately, converting an asynchronous call into a synchronous call to ease static analysis.

Injection Channel	API
External	NFC.addNdefListener
	NFC.addMimeTypeListener
	NFC.addTagDiscoveredListener
	Bluetooth.getUuids
	WifiInfo.get
	barcodeScanner.scan
	FileTransfer.download
Internal	contacts.find
	DirectoryEntry.getDirectory/getFile
	DirectoryReader.readEntries
	Entry.getMetadata
	FileEntry.file
	FileReader.readAsText/readAsDataURL
	Media.getFormatData

Table 2: Injection Channels

Program slice construction. The goal of program slice construction is to construct a fraction of the JavaScript program that reads from an untrusted channel and further processes the input. It means that we do not intend to discover all the possible entry points, which can be a challenging problem by itself, because the asynchronous nature of JavaScript code. Instead, our starting point is the function that calls an API reading from an untrusted channel, and repeatedly include the other functions that can be reachable from this function in the call graph.

In addition, we should also include the functions that are called asynchronously by the functions that are already been added to the slice. Such an asynchronous call can happen in one of the two cases. In the first case, an event handler is registered by the API `addEventListner` to be triggered at certain event. According to our modeling in Figure 5, the registered event handler will be called inside the code snippet as an indirect call. In the second case, application code itself may contain a similar event registration and callback logic.

To discover these functions and add them into the program slice, we take the following approach. For each callsite in the slice, we examine all the parameters and see if any of them refers to a function name defined in the app's code. If so, we consider it to be a potential asynchronous call target and add the function body into the slice. This process is also performed repeatedly until no more functions can be included. Of course, this simple approach may introduce noise, because a variable name may happen to be the same as the function name but in fact refers to something else. This is acceptable, because in the next step we will perform context-sensitive points-to analysis on the slice and the noisy functions will be pruned away.

Algorithm 1 presents this slice construction algorithm. It first identifies all the functions that call the APIs reading from untrusted channels, and put them into a set C. Then starting from each function in C, it builds a call graph and puts the offspring functions into the slice O. It uses a stack S to build the call graph. To include asynchronous call targets, it examines every callsite. If a parameter refers to a function definition, that function will also be included into the slice O and pushed onto the stack S. This procedure terminates when the stack S is empty.

Static taint analysis on the slice. We rely on WALA [18] to build the call graph on the slice constructed from the previous step. In order to link the asynchronous calls with their actual targets, we perform context-sensitive control flow analysis and construct an extended call graph. On this call graph, we look for callsites of unsafe APIs and treat them as the sinks, and compute a backward slice starting from each sink. In the generated slice, if we can identify the input of unsafe APIs coming from the code injection channel, we consider this app vulnerable.

Algorithm 1 JS Slice Builder

Input: One HTML file and its included JS files
Output: JS slice O

$E \leftarrow \{APIs\ that\ read\ from\ untrusted\ channels\}$
$D \leftarrow \{HTML\ file\ and\ its\ included\ JS\ files\}$
$C \leftarrow GetCallers(E, D)$
$O \leftarrow \emptyset$
for all $c \in C$ **do**
  $f_d \leftarrow FindFunctionBodyOfCaller(c, D)$
  $O \leftarrow O \cup \{f_d\}$
  $S \leftarrow Push(f_d)$
  while $S\ is\ not\ empty$ **do**
    $f_d \leftarrow S.Pop()$
    $C_e \leftarrow \emptyset$
    for all $s \in getStatements(f_d)$ **do**
      if $s\ is\ callsite$ **then**
        $c_e \leftarrow getCallee(s)$
        for all $p \in getParameterList(s)$ **do**
          if $p\ is\ string$ **then**
            $C_e \leftarrow C_e \cup \{p\}$
          end if
          if $p\ is\ function\ definition$ **then**
            $S \leftarrow Push(p)$
          end if
        end for
        $C_e \leftarrow C_e \cup \{c_e\}$
      end if
    end for
    for all $c_e \in C_e\ \wedge function\ body\ of\ c_e \notin O$ **do**
      $f_d \leftarrow FindFunctionBodyOfCallee(c_e, D)$
      $O \leftarrow O \cup \{f_d\}$
      $S \leftarrow Push(f_d)$
    end for
  end while
end for

Limitations. While our evaluation results in Section 4.4 demonstrate the effectiveness of our solution, we have to admit several limitations and plan to address them in future work. To name a few, our slice construction may still miss certain asynchronous functions calls that are registered beyond the identified calling contexts; our modeling of the framework is manual and may be incomplete; and our static taint analysis may not be sound. We may leverage several existing efforts in JavaScript analysis to address these limitations. For instance, a recent work by Madsen et al. provided a practical solution to analyze JavaScript code in presence of large frameworks and complex libraries [41], and Guarnieri et al. offered the soundness guarantee for static taint analysis in JavaScript [29].

4.4 Evaluation

We test our detection tool on the 15,510 PhoneGap apps downloaded from the Google Play market. The experiments are conducted on a computer with Core i7-2600 3.4GHz and 16GB of RAM. To optimize the analysis throughput, we limit the processing time of each app to 20 minutes and each HTML file to 30 seconds.

Performance. We measured the execution time of our detection tool. The average running time for each app is 15.38 seconds, but the running time for different apps varies quite significantly. There are two main factors that affect the performance. First, our tool needs to conduct pre-processing and construct all relevant program

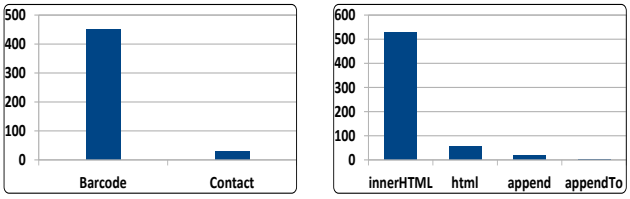
slices; for apps with many HTML files or large JavaScript files, the pre-processing will consume quite a long time. Second, long call-chains in program slice will also cost WALA more time to build the call graph.

Accuracy. From the 15,510 PhoneGap apps, our tool flags 478 apps as potentially vulnerable to the code injection attack. We manually verified all the flagged apps. The process of verification mostly relies on human expertise. In the end, we identified 11 flagged apps as infeasible to exploit; therefore, the false positive rate is 2.30%. We further looked into the 11 false-positive cases, and found that the vulnerable parts in these cases are dead code, which will never be triggered. We have high confidence to speculate that our tool has high accuracy to determine vulnerable apps.

4.5 Case Studies

We would like to conduct a further study on the 478 vulnerable apps identified using our tool. Our goal is to understand what makes them vulnerable and what damage can be achieved if they are under attack.

Vulnerabilities. As we described earlier, there are two conditions that make an HTML5-based app potentially vulnerable: (1) the presence of one of the code injection channels, and (2) the use of unsafe APIs. We would like to see what channels and unsafe APIs that these vulnerable apps use.



(a) Identified Injection Channel (b) Unsafe APIs

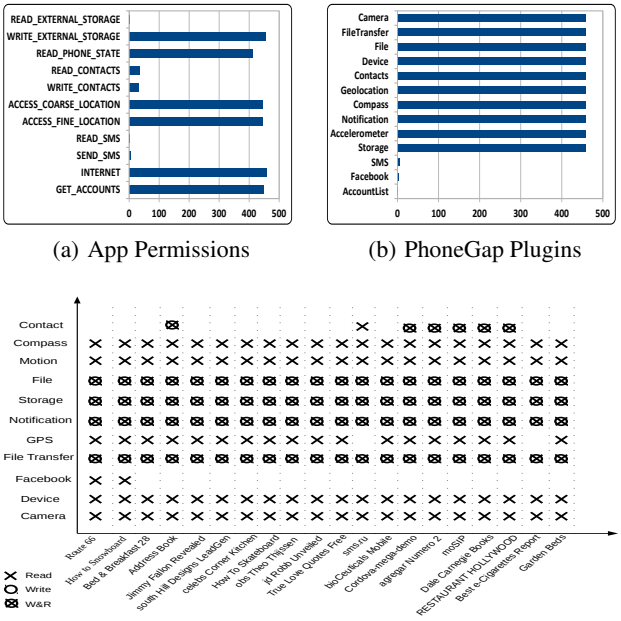
Figure 6: The cause of vulnerabilities

We analyze two channels that are identified by our detection tool. Figure 6(a) shows the distributions of these channels among the vulnerable apps. It is clear that the vulnerabilities of most of these apps attribute to the barcode channel. The popularity of barcode is probably due to the fact that most apps allow users to scan QR coupon or other promotion information.

We examined the unsafe APIs that have led to the vulnerabilities among these apps. Figure 6(b) depicts their distribution. The figure clearly shows that `innerHTML` is the most frequently used API in these apps. This is consistent with the statistics shown in Table 1, which reveals that 90.9% of the 15,510 PhoneGap apps use `innerHTML`.

Damage. We would like to see how much damage can be achieved if an app is compromised via the code injection attack. After malicious JavaScript code is injected into the victim app and gets triggered, there are two conditions that decide what damage can be achieved by the malicious code. We analyze the situations of these conditions among the vulnerable PhoneGap apps.

First, in Android, apps need to have the corresponding permissions in order to conduct certain operations, such as operating cameras or getting locations. We have selected a set of security-sensitive permissions, and count how many vulnerable apps have those permissions. The results are plotted in Figure 7(a). These permissions not only enable the malicious JavaScript code to steal private information, but also allow the attackers to inject a copy of the malicious code to other vulnerable apps. For example, after compromising an



(a) App Permissions (b) PhoneGap Plugins

(c) Case Study of 20 Selected Apps

Figure 7: The damage of the attacks on the vulnerable apps

app that has the `SEND_SMS` and `READ_CONTACTS` permissions, the malicious JavaScript code can send a copy of itself to all of the victim's friends, and can thus potentially affect more people.

Second, JavaScript code runs inside WebView, so it cannot conduct arbitrary actions due to the sandbox implemented in WebView. There are two types of channels that allow JavaScript to conduct potentially damaging operations. One channel is the APIs provided by HTML5, such as getting the geolocation; the other channel is the plugins provided by PhoneGap, such as sending SMS messages or reading from the Contact. The HTML5 APIs channel is always present, as it is part of the implementation of WebView. However, the PhoneGap plugins channel depends on whether the app has included the plugins. We wrote a tool to identify what plugins are included, and the results are plotted in Figure 7(b). We only show the security-sensitive plugins.

The figure shows that many dangerous plugins are present in all vulnerable apps. This is because prior to Version 3.0, these plugins are all included in the PhoneGap library (it has 16 built-in plugins), and are thus included in apps. Other than these plugins, some apps do use third-party plugins, including SMS plugin, Facebook plugin, and AccountList plugin.

Here we like to estimate the capability of an average attacker, who can only make use of HTML5 APIs and installed plugins to cause damage, as long as the app has the corresponding permissions. This is definitely a lower bound, because a sophisticated attacker may be able to exploit known or unknown vulnerabilities in WebView (e.g., a documented WebView vulnerability in Android, CVE-2013-4710 [6]) to execute arbitrary commands in the Android system [22].

Under this assumption, we select 20 most powerful apps among the 478 vulnerable apps, and depict what damage can be achieved if their code-injection vulnerabilities get exploited. Fig 7(c) shows the results. From the figure, we can see that all of these apps have device, notification, storage, motion, compass plugins; the use of these entities do not need any permission. Moreover, all these apps have plugins to access file system and camera gallery, which require the `READ/WRITE_EXTERNAL_STORAGE` permission, but all these apps are granted with either permission. Through these

plugins, attackers can steal personal information or infer the user's behavior information (e.g. using the motion sensor). From the figure, we can also see that all these apps, except two, can access the victim's location; this is because all these 20 apps have access to the HTML5's geolocation APIs, but 2 of them are not granted the required `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permission.

The figure also shows that six vulnerable apps can read from and write to the Contact, because they have both plugins and permissions. Therefore, malicious code injected into these six apps can not only steal the user's information, but also inject a copy of itself into the Contact, so when another vulnerable app reads from the Contact, the malicious code can spread to the next target.

5. ATTACK MITIGATION

In this section, we discuss how to mitigate the code injection attacks on HTML5-based mobile apps. Mitigation can be designed at three different levels: the app level, the framework level, and the operating system level. For the framework level, we only focus on the PhoneGap framework, but the same idea can be applied to other frameworks. For the OS level, we only focus on Android.

5.1 App-Level Mitigation

To defend the code injection attack, app developers can provide protection for their own apps. We provide the following guidelines to app developers.

Sanitization. Before displaying data coming from untrusted resource, it is important to filter out any code inside the data. Developers can use the existing HTML sanitizer, such as google-caja [4], to conduct the filtering.

Use safe APIs. As we have shown in Table 1, there are APIs/attributes that are immune to the code injection attack. For example, `textContent`, `innerText`, and `text()` are safe to use. These APIs simply display the data as they are, without trying to extract the code from the data. These APIs do have limitations, as they also display HTML tags as pure text, leaving the intended HTML tags not interpreted. Another viable solution is to use the auto-escape technique to apply escaping modifiers on the untrusted data, essentially disabling the JavaScript code inside the data. C-template [8] is a good candidate for this purpose.

Use different ways to display data. In addition to safe APIs, there are many other ways to display data without triggering the code inside. We describe some approaches here. (1) An app can predefine a text field using `<input type="text" ...>`. When the app needs to display potentially dangerous data, it can display the data in this field; JavaScript code will not be triggered. (2) An app can use `alert()` to display data to users in a separate window, instead of inside the same HTML page. This call does not trigger JavaScript code. (3) An app can invoke the default browser to display untrusted data, instead of displaying data inside the app. This way, even if the JavaScript code is invoked, the damage is limited to the default system browser, which does not have much privilege.

5.2 Framework-Level Mitigation

As we describe in Section 2, The PhoneGap framework consists of two parts: bridge and plugin. The bridge part connects JavaScript and native code, while the plugin part is used to directly access different types of resources, such as Camera, SMS, Contacts, etc. Plugins can be developed by third parties.

To mitigate code injection, we can implement filters inside the framework, because all the data have to pass through the bridge and plugins. We have two choices. One choice is to place the filter inside plugins, and the other is to place the filter inside the bridge.

5.3 System-Level Mitigation

A well-known system-level solution designed to defeat XSS attacks is called Content Security Policy (CSP) [49, 52]. CSP enforces a fairly strong restriction on JavaScript by disallowing inline JavaScript and `eval()`. CSP can be used to defeat the code injection attack identified in this paper, but CSP is not yet fully supported by WebView. The code that implements CSP can be found in WebKit, which is the basis for WebView, but it is not clear whether, if at all, WebView will enable CSP, because CSP is quite restrictive, and enforcing it may break many existing apps, especially those that have inline JavaScript code. A great amount of efforts are needed to rewrite the existing apps once CSP is enforced.

5.4 Our Implementation

After comparing the above approaches from the practicality aspect, we chose to implement the framework-level mitigation, because this is the most practical solution: it does not involve operating system modification, while still being transparent to apps. We just need to convince the PhoneGap group to adopt this approach, or convince app developers to use our modified PhoneGap library.

We have implemented a prototype called NoInjection as a patch to the PhoneGap framework in Android. Its main design principle is to add a filter inside the bridge, right after the data enter the bridge from plugins (see Figure 1 for the architecture of PhoneGap). This implementation is transparent to plugins, as well as to apps. App developers do need to download our revised PhoneGap library (called `cordova.jar`) when compiling their code. This library can be downloaded from our web site [5].

For the filtering part, we simply use an open-source library called `jsoup`. `jsoup` is an HTML parser implemented in Java; it provides an API called `clean()` to filter out JavaScript code from a string. This library has been tested quite extensively, and it can filter out JavaScript code that is embedded in a variety of ways. Therefore, the effectiveness of our solution depends on the `jsoup` library. Moreover, `jsoup` also provides a whitelist mechanism to allow some valid HTML tags. This mechanism allows us to filter out all the JavaScript code, while keeping the valid HTML tags, e.g., we can keep the `img` tag if it does not have an event attribute.

Overall, our implementation of NoInjection adds only 148 lines of Java code to PhoneGap, plus the `jsoup` library. To measure the performance overhead of NoInjection, we benchmarked the modified PhoneGap against the original PhoneGap. We called each plugin's API 1000 times to get the average time spent on each invocation. The results show that the modified PhoneGap takes 2.675 ms for each call, and the original PhoneGap takes 2.435 ms. The overhead for our implementation is 9.85% for each call. Since these invocations only contribute to a small portion of the overall running time, the overhead caused by our work is quite insignificant.

Theoretically, our solution should be able to apply to the existing PhoneGap apps, by replacing their PhoneGap library with our revised one. However, in practice, this is hard, because most of the PhoneGap apps use older versions of PhoneGap library, while our revision is conducted on the most recent version. To apply our solutions to the existing PhoneGap without recompiling the app, we need to patch the corresponding PhoneGap versions. While our solution can benefit new PhoneGap apps, it does show a limitation when dealing with the existing PhoneGap apps.

6. RELATED WORK

6.1 XSS Attack Detection and Mitigation

Due to the high practical prevalence, XSS attacks have been studied by a lot of researchers. Here we will focus on the client-side XSS attack detection and mitigation, because some work may help detect and mitigate the code injection attacks on HTML5-based mobile applications if they are properly used.

Detection on XSS Attack A series of works use static and dynamic analysis to detect XSS vulnerabilities. Vogt et al. [50] use taint-analysis to track the flow of sensitive information inside web browsers. Saxena et al. [46] propose a system called Flax that uses "taint enhanced blackbox fuzzing" to find command and code injection vulnerabilities in JavaScript. Sebastian et al. [38] directly integrate tainting techniques into the browser's JavaScript engine to track unsafe data flows. DOMinator [23] uses Firefox's SpiderMonkey Javascript engine to understand the code. It keeps a call stack on user controllable strings and raises alert when these strings are passed into the sinks. IceShield [30] performs dynamic analysis of JavaScript code in browsers to detect code injection attacks. Rozzle [37], a JavaScript multiexecution VM, explores multiple execution paths within a single execution to expose environment-specific malware. These works may help detect the code injection attacks on HTML5-based mobile applications. However, defining source entries for HTML5-based mobile applications still remains as a challenge.

Mitigation on XSS Attack One way to mitigate XSS attack is to use sanitization mechanisms. The key challenge is how to identify the code mixed in data. Several approaches have been proposed to address this challenge, including XSS Auditor [20], Bek [32], CSAS [45], ScriptGard [47], etc. We can adopt some of the sanitization methods to remove script from string to prevent the attack; however, the challenge is where to place the sanitization logic. Our prototype chooses to implement the logic at the framework level, because it is the most practical solution.

Another way to mitigate XSS attacks is to limit the damage caused by the code injection attack, rather than preventing it. Content Security Policy [49, 52] is the representative of this category, and it was discussed in Section 5. ConScript [42] and Escudo [34] also defines policy to limit privilege of the script in some specific DOM elements. They have the same problem as CSP, which requires significant code rewriting for the existing apps.

6.2 Static Analysis of Android Vulnerabilities

Recently, some works use static analysis tools to detect various vulnerabilities in Android systems. Stowaway [26] finds over-privileged apps that require additional permissions beyond normal functionalities. Woodpecker [28] analyzes preloaded apps in the phone firmware to expose capability leaks on stock Android phones. ContentScope [55] focuses on passive content leak and content pollution vulnerabilities. SMV-HUNTER [48] and MalloDroid [25] can be used to detect the apps that are vulnerable to SSL/TLS Man-in-the-Middle attacks. Sebastian et al. analyze unsafe and malicious code loading in Android applications [43]. CHEX [39] detects applications that expose components to other applications in an insecure way. AppIntent [53] addresses the unintended sensitive data transmission. AppSealer [54] combines static and dynamic code analysis to mitigate component hijacking attacks. Karim et al. [24] present an efficient approach to identify malicious Android applications through specialized static program analysis. Compared to these studies, our work is the first one to address the code injection problem in HTML5-based mobile apps.

6.3 Other Related Attacks.

Some other attacks are also related to our work. mXSS [31] attack can bypass XSS filters by taking advantage of innerHTML's mutation. XCS [21] finds some interesting channels to ship the code, such as printer, router and digital photo frame etc. Once the code is loaded in the web browser, it will get executed. In our work, most of the channels are quite unique to mobile platforms.

A work-in-progress version of our work has appeared in a workshop paper [35]; however, that paper only focuses on describing how the attack works, while this submission includes the following significant contributions that are not covered by the workshop version: (1) We have developed a detecting tool, using which we have conducted a large-scale analysis of over 15,510 HTML5-based (PhoneGap) apps; we have discovered 478 vulnerable apps. The work in the workshop paper used manual effort, and only found less than 10 vulnerable apps. (2) We have developed a mitigation solution that can be immediately adopted by PhoneGap app developers or the PhoneGap framework. (3) We have extended the attacks to include internal channels, which are different from the external channels identified by the workshop paper.

Several studies have also analyzed the security of WebView and PhoneGap [27, 36, 40, 51]. Georgiev et al. [27] and Jin et al. [36] address the problem caused by the code coming from untrusted origins. The solutions limit the privilege of the untrusted code by not allowing it to access local mobile resources. However, these solutions are not suitable to defend our code injection attack, because in our cases, app developers do not even know that there may be code inside the data.

7. SUMMARY

In this paper, we study the potential risk imposed by HTML5-based mobile applications. We have identified a number of unique channels that can be used to inject code, including `Contacts`, `SMS`, `Barcode`, etc. To assess the extent of such a vulnerability in Android apps, we have implemented a tool to analyze 15,510 PhoneGap apps collected from the Android Market. The tool flagged 478 apps as vulnerable, with only 2.30% false positive. We have also implemented a prototype called NoInjection as a patch to the PhoneGap framework in Android. It can successfully filter out the malicious code from the attack channels identified in this paper. The tools developed from this work, the demonstration of the attacks, and guidelines to users and developers can be found from our web site [5]. In our future work, we plan to extend our detection and mitigation to non-PhoneGap HTML5-based apps.

8. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable and encouraging comments. This work was supported in part by NSF grants 1017771, 1318814, 1018217, 1054605, a Google research award and McAfee Inc. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

9. REFERENCES

[1] 75% of developers using html5:survey. `http://eweek.com/c/a/Application-Development/75-of-Developers-Using-HTML5-Survey-508096`.

[2] Appcelerator. `http://appcelerator.com`.

[3] appmobi. `http://www.appmobi.com/`.

[4] Caja. `http://code.google.com/p/google-caja/`.

[5] Code Injection Attacks on HTML5-based Mobile Apps. http://www.cis.syr.edu/~wedu/android/JSCodeInjection/index.html.

[6] CVE-2013-4710. http://www.exploit-db.com/exploits/31519/.

[7] The future of mobile development: Html5 vs. native apps. http://www.businessinsider.com/html5-vs-native-apps-for-mobile-2013-4?op=1/.

[8] How To Use the Ctemplate (formerly Google Template) System. http://google-ctemplate.googlecode.com/svn/trunk/doc/guide.html#auto_escape.

[9] Html5 vs. apps: Where the debate stands now, and why it matters. http://www.businessinsider.com/html5-vs-apps-where-the-debate-stands\-now-and-why-it-matters-2013-4/.

[10] Html5 vs native: The mobile app debate. http://www.html5rocks.com/en/mobile/nativedebate/.

[11] MooTools-a compact JavaScript framework. http://mootools.net/.

[12] MoSync:Cross-platform SDK and HTML5 tools for mobile app development. http://mosync.com.

[13] Owasp. the ten most critical web application security risks. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf.

[14] Phonegap. http://phonegap.com.

[15] Prototype javascript framework. http://prototypejs.org/.

[16] Rhomobile. http://rhomobile.com.

[17] The shared future of html5 and native apps. http://itbusinessedge.com/blogs/data-and-telecom/the-shared-future-of-html5-and-native-apps.html/.

[18] The T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[19] Mobile future in focus 2013. *comScore*, 2013.

[20] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[21] H. Bojinov, E. Bursztein, and D. Boneh. XCS: cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 420–431. ACM, 2009.

[22] E. Chin and D. Wagner. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Proceedings of the 14th International Workshop on Information Security Applications*, Ocean Suites Jeju Hotel, Jeju Island, Korea, August 19 2013.

[23] S. Di Paola. DominatorPro: Securing Next Generation of Web Applications. [software],. https://dominator.mindedsecurity.com/, 2012.

[24] K. Elish, D. Yao, B. Ryder, and X. Jiang. A static assurance analysis of android applications. *Virginia Polytechnic Institute and State University, Tech. Rep*, 2013.

[25] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 50–61. ACM, 2012.

[26] A. 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*, pages 627–638. ACM, 2011.

[27] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[28] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[29] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2011.

[30] M. Heiderich, T. Frosch, and T. Holz. Iceshield: detection and mitigation of malicious websites with a frozen dom. In *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses*, pages 281–300. Springer, 2011.

[31] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Yang. mxss attacks: attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, CCS '13. ACM, 2013.

[32] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[33] N. Huy and D. vanThanh. Evaluation of mobile app paradigms. In *Proceedings of the 10th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '12, 2012.

[34] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceeding of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 231–240, 2010.

[35] X. Jin, T. Luo, D. G. Tsui, and W. Du. Code Injection Attacks on HTML5-based Mobile Apps. In *Mobile Security Technologies (MoST) 2014*, 2014.

[36] X. Jin, L. Wang, T. Luo, and W. Du. Fine-Grained Access Control for HTML5-Based Mobile Applications in Android. In *Proceedings of the 16th Information Security Conference (ISC)*, 2013.

[37] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of 2012 IEEE Symposium on Security and Privacy*, pages 443–457. IEEE, 2012.

[38] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM conference on Computer and Communications Security*, pages 1193–1204. ACM, 2013.

[39] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 229–240. ACM, 2012.

[40] 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)*, 2011.

[41] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[42] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of 2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[43] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceeding of the Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[44] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, 2010.

[45] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (ACM CCS)*, 2011.

[46] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceeding of the Network and Distributed System Security Symposium (NDSS 2010)*, 2010.

[47] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *18th ACM Conference on Computer and Communications Security (ACM CCS)*, 2011.

[48] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and K. Latifur. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceeding of the Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[49] 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)*, pages 921–930. ACM, 2010.

[50] 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 *Proceeding of the Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[51] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *ACM Conference on Computer and Communications Security (ACM CCS)*, Berlin, Germany, 2013.

[52] J. Weinberger, A. Barth, and D. Song. Towards client-side html security policies. In *Workshop on Hot Topics on Security (HotSec)*, 2011.

[53] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM conference on Computer and Communications Security*, pages 1043–1054. ACM, 2013.

[54] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[55] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.