A Testbed and Process for Analyzing Attack Vectors and Vulnerabilities in Hybrid Mobile Apps Connected to RESTful Web Services

Matthew L. Hale
School of Interdisciplinary Informatics
University of Nebraska at Omaha
Omaha, NE USA
mlhale@unomaha.edu

Seth Hanson
School of Interdisciplinary Informatics
University of Nebraska at Omaha
Omaha, NE USA
sethhanson@unomaha.edu

***Abstract*—Web traffic is increasingly trending towards mobile devices driving developers to tailor web content to small screens and customize web apps using mobile-only capabilities such as geo-location, accelerometers, offline storage, and camera features. Hybrid apps provide a cross-platform, device independent, means for developers to utilize these features. They work by wrapping web-based code, i.e., HTML5, CSS, and JavaScript, in thin native containers that expose device features. This design pattern encourages re-use of existing code, reduces development time, and leverages existing web development talent that doesn't depend on platform specific languages. Despite these advantages, the newness of hybrid apps raises new security challenges associated with integrating code designed for a web browser with features native to a mobile device. This paper explores these security concerns and defines three forms of attack that can specifically target and exploit hybrid apps connected to web services. Contributions of the paper include a high level process for discovering hybrid app attacks and vulnerabilities, definitions of emerging hybrid attack vectors, and a testbed platform for analyzing vulnerabilities. As an evaluation, hybrid attacks are analyzed in the testbed showing that it provides insight into vulnerabilities and helps assess risk.**

Keywords-hybrid mobile application; vulnerabilities; web services; web browser; thin native containers; attack vectors;

I. INTRODUCTION

Large scale web analytics data [1], show that mobile devices now account for roughly 25% of all web page views in the US, up from just 10% two years ago. This shift, from desktop to mobile, has led to a mobile first revolution that encourages developers to tailor web content to mobile devices and contextualize web applications (web apps) using mobile-only capabilities such as geo-location, accelerometers, offline storage, and/or camera features.

Hybrid mobile applications (hybrid apps) provide developers a platform and device independent means of accessing mobile-only features by allowing them to build a single application in familiar web languages, such HTML5, CSS, and JavaScript, and then wrap it in thin native containers (TNCs) that provide access to local device features. This approach allows developers to build the rich, high performance, user experiences that mobile users expect while alleviating the need to develop and maintain multiple standalone *native apps* in platform specific languages, such as Objective-C for iOS or Java for Android. These advantages are increasingly leading developers, currently around 32% of all mobile developers [2], to choose hybrid apps for serving multiple mobile platforms. Most hybrid apps are made with an underlying TNC framework called Apache Cordova [3] or its derivative PhoneGap [4].

While hybrid apps, like native apps, can operate offline, the majority are *service consumers* meaning that they consume, and typically display, data from RESTful web services [5]. REST, or REpresentational State Transfer, is a client-server architectural style paradigm that enables client-side applications, such as hybrid apps, to perform user actions immediately while asynchronously making stateless CRUD (Create, Read, Update, and Delete) requests to a web service. In terms of reads, this allows client-side apps to dynamically load different data ranging from JSON objects (objects in the JavaScript Object Notation), to static text or images, to entire HTML DOM (Document Object Model) elements such as *divs* or *scripts*. This means that client-side apps, such as the iTunes store app (a hybrid app), can display web content not originally included in the app itself, such as storefront data from an Apple web service. Other CRUD operations, namely create, update, and delete allow client-side apps, including hybrid apps, to make Ajax (asynchronous JavaScript and XML) requests to external services to save user data, upload files such as images or videos, or make transactions, e.g. iTunes purchases.

From a security perspective, client-side apps are not new. Bodies of work [6-10] have explored and are continuing to explore how best to secure apps. Hybrid apps though, present many unique challenges due to the much larger reachability provided by their native containers [3, 11, 12]. Unlike browser-based web apps, JavaScript executing inside of hybrid apps has direct access to native device features [3, 11]. This means that previously well understood web-based attacks, such as cross site scripting (XSS), that have been historically limited to targeting browser assets can now expand to a profoundly larger attack surface [12], namely the mobile operating system (OS) itself and all of the features granted to the hybrid app, such as geolocation, camera, and other data sensitive features. Despite the emergence of hybrid apps and the fertile new target landscape, security research continues to focus on native mobile apps [9, 10, 13] and web apps [6], leaving the security challenges with hybrid apps largely unstudied.

This paper seeks to bolster efforts [14] to begin closing this gap. We create a high level process and low level

978-1-4673-7275-6/15 $31.00 © 2015 IEEE
DOI 10.1109/SERVICES.2015.35

testbed that allows security researchers to re-imagine web-based browser attacks within the expanded reachability context introduced by hybrid apps and probe applications for vulnerabilities. The testbed allows researchers to deploy attack vector vulnerability test cases to a hybrid app of their choice or a prototype template app that comes with the testbed. It also includes a backend web service for mocking application data inputs and collecting and analyzing the results of attacks. Using the process and testbed, the paper defines three emerging forms of *amalgamated attack vectors* that arise as a result of combining web-based and native attacks. To evaluate the testbed, a proof of concept of each newly identified attack vector is demonstrated and risk is assessed.

The rest of the paper is organized as follows. Section II overviews relevant background that underscores the growth of the mobile application market share, highlights the differences between the three different types of mobile apps, discusses the features available to each type, and covers the existing state of security with each. Section III overviews the uniqueness of the hybrid app security landscape, introduces the high level discovery process, defines emerging amalgamated attack vectors, and develops the testbed. Section IV evaluates the approach through proof of concepts and analysis of each attack vector in different smartphones. Section V concludes the paper.

II. BACKGROUND

Highlighting the trend towards mobile, Figure 1 shows aggregate data [1] collected from roughly 96 billion views across over three million unique websites over a two year period. Mobile devices now account for about 34% of all web usage, (24.45% smart phones, 9.77% tablets) [1]. These market shares are over double the roughly 15% they were two years ago underscoring just how fast the mobile market is growing. Drilling down into these trends, the data [1] shows that 94.83% of the smart phone views counted came from either an iOS (51.4%) or Android device (43.43%).

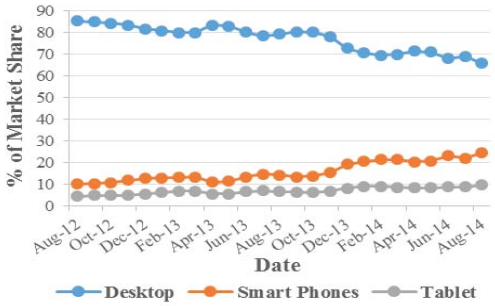

Figure 1: Web Usage by Platform in the US [1] showing a shift from desktop to mobile

Since security threats are closely tied to market share, these trends suggest that mobile devices will increasingly be targeted by attackers. The rest of this section defines the three types of mobile apps and overviews security research.

A. Mobile Web Applications

Web apps are not truly mobile applications. Instead, they are websites that reside on remote web servers, rely on persistent internet access, and simulate the user interfaces (UI) of mobile devices. Typically sites are prefaced with the letter m, e.g. m.facebook.com, offer a subset of the full web app features, and utilize simplified UIs optimized for touch interfaces. All web apps share three common attributes [15]:

1) they must execute in a web browser, e.g. Android Browser, Safari, Chrome, or Firefox for Android,
2) they render content in the browser using web languages, i.e. HTML5, CSS, and JavaScript, and
3) they have limited or no capability to operate offline and utilize device specific functionality [16]

B. Native Mobile Applications

In contrast to web apps, *native apps* offer all of the device and platform specific features available on mobile devices (such as accelerometers, contact list, GPS, and camera), but require multiple disparate codebases to be maintained [16, 17] for each platform an app developer wants to support. Native apps generally provide the richest and most fluid user experience [16, 18], since they utilize device and platform specific UI elements and functionality. They also are able to operate offline and tend to have better performance [16] since they interact directly with the OS. Despite these advantages, native apps are entirely platform dependent, take longer to code, and do not re-use existing web code. This means that to support native apps, developers must learn Objective-C (for iOS devices) or Java for Android (for Android devices) and create entirely new codebases. These factors translate to longer development time, code redundancy, and longer update periods as part of the software maintenance lifecycle [16-18].

C. Hybrid Mobile Applications

The third kind of mobile app resides somewhere between the other two and inherits many of the best attributes of both. Hybrid apps allow developers to create or modify existing web apps and then deploy them to specific platforms using thin native containers (TNCs). TNCs [3, 4] are a type of wrapping technology that utilize platform specific browser engines and native APIs to provide encapsulated web apps with endpoints to native device features unavailable to web apps. Many web developers prefer Hybrid Apps, since existing codebases can be re-used without learning and porting apps to platform specific languages such as Objective-C or Java for Android. These advantages are tempered only by a slight hit in performance that comes with executing encapsulated JavaScript instead of purely native application languages.

Table 1 provides feature comparisons across the three types of mobile applications that highlight some of the reasons why the hybrid app market is growing [1]. These features are compiled from [2, 15-18] and color coded based on desirability (low to high from red to yellow to green). The different features under the device features heading in Table 1 are the major differentiators between web ecosystems and hybrid or native mobile apps. Later sections return to these features and highlight specific hybrid app attack vectors that target them.

Table 1: Feature Comparison across Mobile App. Types

Generic Attributes	Web	Native	Hybrid
Languages	HTML5, CSS, JavaScript	Platform Specific	HTML5,CSS, Javascript
Distribution	Web	Appstore	Appstore
Graphics	HTML, Canvas, SVG	Native APIs	HTML, Canvas, SVG
Connectivity	Online	On & offline	On & offline
Performance	Slow	Fast	Moderate
Home button	Bookmark only	Default	Default
Platform/Device Independent	Yes	No	Yes
Dev. Time	Lowest	High	Low
Device Features	**Web**	**Native**	**Hybrid**
Accelerometer	Some Devices	Yes	Yes
Account Access (Stored Accts)	No	Yes	Yes
Battery State	Some Devices	Yes	Yes
Camera	Some Devices	Yes	Yes
Calendar	No	Yes	Yes
Contacts	No	Yes	Yes
Flashlight	No	Yes	Yes
Geolocation	Yes	Yes	Yes
Microphone	Some Devices	Yes	Yes
Network Sockets	No	Yes	Yes
Offline Storage	No	Yes	Yes
Push Notific.	No	Yes	Yes
Phone Identity	IP only	ID / IP / Phone #	ID / IP / Phone #
Phone Status	No	Yes	Yes
SMS	No	Yes	Yes
Vibration	Some Devices	Yes	Yes
Video	Some Devices	Yes	Yes

D. Mobile Application Security

Existing mobile security research generally falls into one of three major categories: privacy and policy research [19, 20] which focuses on the cultural and human factors that affect how users trust and use mobile apps; operating system research [21] that looks into OS level capabilities and vulnerabilities, and technical mitigations; and application level research [6, 9, 10, 13] that targets native or web-based mobile app development. Some research efforts also specifically target hybrid application security [12, 14, 22, 23], but the field is still in its infancy.

Privacy and policy research is nicely summed up by Hurlburt et al. in [19] and Balebako and Cranor in [20]. Hurlburt highlights the range of issues that confront mobile device users and their general lack of awareness of what apps they are trusting are actually capable of [19]. The thesis is that users are increasingly trading conveniences afforded by mobile apps with data and personal privacy protections. In contrast to the user side, Balebako and Cranor [20] examine the ways in which application developers can be encouraged to develop apps that support basic privacy protections. They discuss how the economic pressures and time constraints that encourage app developers to push products to the market can be offset by development privacy guidelines, education, and platform modifications. Of the three they highlight how platforms supporting mobile apps can encourage good privacy practices by making it so that developers have to "dig for the more specific privacy-invasive information" [20] rather than getting it without needed it as part of API calls.

Operating system security research typically focuses on a number of factors relevant to different OS capabilities. Of particular interest to the work in this paper, Hunt [21] examines and demonstrates OS-level vulnerabilities that can potentially allow for data leakage or file stealing by malicious entities. Specifically, Hunt examines types of App store malware that pose as legitimate software in the google play store, but when downloaded and executed run malicious scripts behind a seemingly legitimate veneer to steal data and upload it to malicious servers. One example provided was a weather application that iterated over local files and uploaded images stored on the victim's phone to a malicious web server by circumnavigating data access parameters. Such malware could potentially leak private photos or documents or log contextual information (such as GPS data) without a user's knowledge.

Application level security research focuses mainly on native [9, 10, 13] or web-based apps [6, 10]. Aside from a handful of researcher efforts such as [22] and [23] that focus on security additions for hybrid app TNCs, hybrid app security research is lacking [22]. Jain and Shanbhag [9] highlight several security issues affecting mobile applications (native and web) that include client-side injection, weak server-side controls, improper session handling, security decisions via untrusted input, side-channel data leakage, and insecure data storage, among others. In addition to general secure coding guidelines such as least privilege and data input validation, they suggest a few mobile secure coding best practices that include avoiding the storage of sensitive information in areas of memory accessible to other apps, using device-based cryptography, and avoiding cut-copy-paste and autocompletion features offered by the mobile OS.

With regard to hybrid apps, Jaramillo et al. [22] provide an initial look at security extensions for TNCs that wrap hybrid apps. Their work provides a model of what Apache Cordova [3] and Phone Gap [4] extensions would look like if security measures were added, but stops short of an implementation. They point to the need for better vulnerability analysis within TNCs. Nadkarni [23] proposes an application wrapping architecture that embeds an arbitrary website within a hybrid app and distributes it as an application package. Their approach is useful for limiting permissions to web apps, but doesn't address security issues with hybrid apps or TNCs themselves. Shehab and AlJarrah [12] look at reducing the attack surface on Cordova-based hybrid apps. They propose a policy based approach to apply least privilege to limit hybrid app permissions on a per page (in the encapsulated app) basis.

In highly related work, Xing et al. [14] look at new malicious code injection channels in hybrid apps and propose a slicing method to identify malicious code when deployed to Cordova features. Their approach offers the largest step forward in hybrid app vulnerability mitigation, but has (stated) limitations that restrict its usage on malicious code that injects itself into certain contexts. The attacks in this paper are examples of some that are not detectable by their framework.

III. EXPLORING HYBRID APP SECURITY

Trends towards hybrid mobile apps spotlight a critical need for parallel security efforts. The work discussed in this section targets this need and makes forward progress towards better hybrid app security. First, the unique issues with hybrid apps are overviewed. Second, a five step vulnerability discovery process is discussed for finding and categorizing hybrid app security issues. Next, three *amalgamated attack vectors*, i.e. combining multiple domains, are derived. Lastly, a testbed is developed for deploying and analyzing attacks in real world smartphones.

A. Hybrid Resource Landscape

In a traditional web environment the *attack surface* of an application, i.e. the various points at which an attack vector can occur, is limited to the web server(s) hosting the application, the browser a user uses to connect to the application, and the network(s) in which the application and user communicate. Attacks targeting end users, therefore, are limited in scope to the resources that are both reachable and manipulatable by the browser. In this sense, the browser forms a *sandbox* [7, 8] that contains any malicious effects and limits the scope of attack. Figure 2a depicts a vulnerable web application from the perspective of the end-user environment. In this situation, operating system features and capabilities, such as file storage, other desktop or mobile applications, or system functions, are outside the reachability of the sandboxed browser container and, therefore, safe from attack.

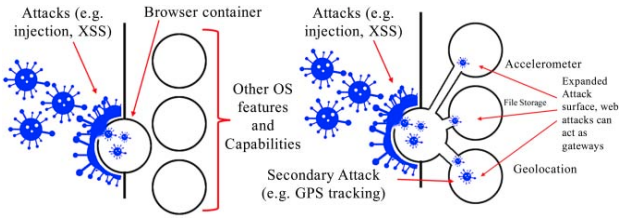

Figure 2: a) Vulnerable web app with sandboxed browser b) vulnerable hybrid app with no sandboxing

Injection attacks [9] are a common form of malicious user input attack that targets a web application's upload capabilities through the use of a web form or REST API in order to inject malicious server-side or client-side scripts into the application code. These malicious scripts, when executed, have the capability to expose sensitive user information or misuse server or end-user resources. In typical web apps such as the one shown in Figure 2a, injection attacks only have the capability to manipulate resources held in the browser (such as session cookies, or stored user information) since the browser stands between the script and local system resources on a user's computer. This assumption is simply not true in hybrid web applications where the TNC API makes local device functionality available to the internal web app [15, 22].

For these reasons, standard web application attack vectors must be reexamined in the native application ecosystem to determine what additional local resources may be attacked and misused. Returning to the injection attack example, it might be possible for malicious attackers to access any of the features overviewed in Table 1. This means an injection attack could potentially read and write to offline local device storage, manipulate core device features, or exhaust resources such as the battery or memory. This example is demonstrated in the generalized vulnerable hybrid app shown in Figure 2b. Like its companion in Figure 2a, the app shown in Figure 2b is vulnerable to some form of web-based attacks. Unlike its companion though, hybrid apps integrate the other OS features and capabilities with the browser container. While this allows the app to use device features such as GPS, it expands the attack surface of the app leaving it vulnerable to secondary attacks.

B. Vulnerability Discovery and Assessment

Malicious scripts executing in hybrid app environments can potentially use any or all device features the app was granted permissions for upon installation. Therefore, it is vitally important to re-imagine standard web attacks as *gateways* for launching additional attacks specific to hybrid apps. This section defines a five step process for discovering vulnerabilities and assessing risks in hybrid apps. The process works by modifying traditional attack vectors to include hybrid specific resource targets and then re-assessing criticality in different deployment environments.

The five step process is shown in Figure 3. The overarching goal of the process is to determine if there are unique hybrid app specific vectors that are derived from traditional web attacks. Step 1 chooses a web attack vector for further examination in hybrid apps. Existing web attacks identified in projects such as OWASP (Open Web Application Security Project) [24] and the web based subset of CAPEC (i.e. Common Attack Pattern Enumeration and Classification) [25] are a good starting point for considering hybrid app specific attack vector variants.

In step 2, a native device resource or feature exposed by the TNC API is selected for examination. Such resources or features could be any one of the entries in Table 1. The goal of the remaining steps is to elicit and understand any unique effects that arise from applying the web attack vector in hybrid apps while native features are *reachable* by app. For example, one can imagine a *cross site scripting* (XSS) attack that might attempt to abuse usually unreachable targets such as camera or GPS features.

Step 3 creates such test cases while step 4 deploys them to different platforms of interest to the security researcher.

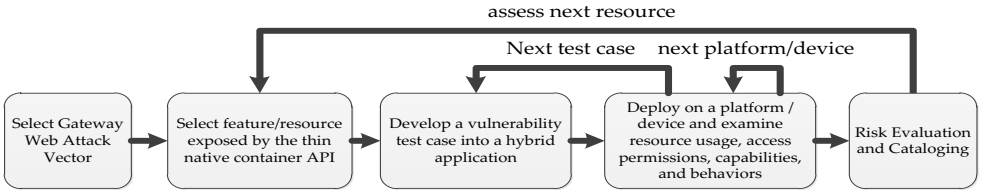

Figure 3: Five Stage Vulnerability Discovery and Risk Assessment Process

For instance if an organization offers a hybrid application to their users in just Android, then it would be prudent to examine all relevant OS versions, e.g. Android 4-5, and devices, e.g. Galaxy, Nexus, etc., that users might have. After all relevant platforms and devices have been examined, the process either proceeds to the next test case (i.e., repeats step 3), or moves on to risk evaluation (step 5).

In the final step, the test cases and deployment profiles are combined to assess the criticality of the identified amalgamated attack vector and document it accordingly. Assessments can drill down into OS type to provide specific information about differences in the way the vector applies to particular OS versions or to different devices. The goal of risk assessment is to quantify the impact and likelihood of attack vector exploitation across the breadth of devices and platforms an organization supports with their hybrid apps.

C. Amalgamated Attack Vectors

Exemplifying the vulnerability assessment process, this section identifies three emerging *amalgamated attack vectors*. Each vector connects the dots between web attacks and hybrid app features, allowing attackers access to features not present, or accessible, in web browser ecosystems. Following the Figure 3 process, a standard web attack called *stored cross-site scripting* [24, 26] (sXSS) was selected in step 1 as the gateway vector for the three amalgamated attacks below. Defined by OWASP [24] as a subset of XSS attacks, sXSS involves persisting injected scripts in user viewable content. When a victim views the content, a script executes and the attack is launched.

The first attack examines the effect of applying sXSS when the GPS function is reachable (step 2). In this scenario an attack might exploit a vulnerable web form to insert a script that specifically targeted the Cordova *geolocation* library [11] wrapped into GPS-enabled hybrid mobile apps. Given the ability to access GPS capabilities, an attacker might passively track a user's movements by continually polling the GPS and logging the results to a server. This amalgamated attack vector, labeled *sXSS-geolocation*, is modeled as a *vulnerability test case* (step 3) in Figure 4.

```
var r = { 'lat':'','lon':'','acc':'',          1
          'h':'','s':'','ts':''};              2
var geolib = navigator.geolocation;            3
geolib.getCurrentPosition(function (pos){      4
    r.lat = pos.coords.latitude;               5
    r.lon = pos.coords.longitude;              6
    r.acc = pos.coords.accuracy;               7
    r.h   = pos.coords.heading;                8
    r.s   = pos.coords.speed;                  9
    r.ts  = pos.timestamp;                    10
    $.ajax({                                  11
       url: 'http://<attackerurl>',           12
       type: "POST",                          13
       data: r,                               14
       contentType:"application/json",        15
       datatype: "json"}) }                   16
```

Figure 4: SXSS-geolocation test case (JavaScript)

The script polls the GPS and send the results to an attacker. Lines 1-2 define a data structure, called *r* (for result), to store captured GPS data as *lat* (latitude), *lon* (longitude), *acc* (accuracy), *h* (heading in degrees from North), *s* (speed of travel), and *ts* (the timestamp when the data was recorded). Lines 3-4 invokes the Cordova geolocation API, in the *navigator* BOM (Browser Object Model) object, while lines 5-10 populate *r* with the returned data. Finally, lines 11-16 form and execute an Ajax POST request to the attacker's web server (located at *<attackerurl>*) using the jQuery (signified by $) Ajax library. For larger applicability, the attacker could remove the jQuery dependency by using an *XMLHTTPRequest* object provided directly by the *window* BOM object.

Following the logic of the first attack, the second amalgamated vector, labeled *sXSS-accelerometer*, again applies sXSS, but targets the accelerometer instead of the GPS functionality. Based on research results [27] indicating that accelerometer data can be used to deduce phone passcodes or other types of touch inputs, an attacker might want to capture and log accelerometer data as a form of *keylogging* [28]. Such a test case is shown in Figure 5.

```
var r = { 'x':'','y':'','z':'','ts':''};    1
var acclib = navigator.accelerometer;       2
acclib.getCurrentAcceleration(function (a){ 3
    r.x =  a.x;                             4
    r.y =  a.y;                             5
    r.z =  a.z;                             6
    r.ts  = a.timestamp;                    7
    //same Ajax request as in Figure 4})    8
```

Figure 5: SXSS-accelerometer test case

While conceptually similar to the geolocation test case this case attacks the accelerometer. Here, line 1 defines a data structure consisting of the *x*, *y*, and *z* components of acceleration as well as a timestamp *ts*. Lines 3 and 4 invoke the *accelerometer* Cordova library and lines 4-7 log the current accelerometer data for the polling sample *a*. Finally, line 8 executes the same Ajax script used in Figure 4 (i.e., lines 11-16) to log the data to the malicious web server.

The last attack concept imagined using sXSS to access and abuse a user's camera functionality, i.e. *sXSS-camera*. Here the goal of an attacker might be to actively spy on an unsuspecting hybrid app user. This form of attack could be the most harmful to user privacy and might have fallout akin to recent web cam attacks [29] or security issues with messenger apps such as snap-chat [30]. A test case might involve capturing camera data on some frequency and sending the data to the attacker's waiting web service. Unlike the previous two attacks, the sXSS-camera attack vector involves uploading a file rather than simple JSON. For this reason the Ajax request would need to involve a *multi-part form* request or make use of base64 encoding in order to transmit the file. Furthermore, the attacker would need to actually snap a picture without user input.

Figure 6 prototypes a test case for conducting sXSS-camera attacks using a base64 encoding scheme. Line 1 constructs *r* as simply an *img* string field to store a base64 encoded image. Lines 2 and 3 invoke the Cordova *camera* library, passing the *DestinationType* parameter *DATA_URL* on line 8 to tell Cordova to return the image in its base64 encoded form. Using the returned data *imageData*, lines 4 and 5 escape URL-sensitive characters. Finally, the encoded image is uploaded using an Ajax call in the same fashion as in Figure 4.

```
var r = { 'img':''};                                    1
var camlib = navigator.camera;                          2
camlib.getPicture(function (imageData){                 3
    r.img = imageData.replace('+','-')                  4
    r.img = r.img.replace('/','_')                      5
    //same ajax as in Figure 4                          6
}, { destinationType:                                   7
     Camera.DestinationType.DATA_URL})                  8
```

Figure 6: SXSS-camera test case

D. *Hybrid App Testbed*

Up to this point, steps 1-3 of the five part vulnerability discovery process have been illustrated. To proceed beyond these steps one needs an actual hybrid app and RESTful API to deploy vulnerability test cases to and analyze the results. This section develops a testbed that consists of a prototypical sXSS vulnerable hybrid app and configurable REST API that can be used to deploy and explore vulnerabilities on different smartphone platforms and devices. These components will be used in Section IV to explore steps 4 and 5 of the vulnerability discovery process.

The first component of the testbed is a RESTful API that provides researchers with a configurable way to allow web-based attacks (including sXSS). The API was built using Django [31], a python-based web app framework. To simulate sXSS, researchers can enable an unsafe form field (shown in Figure 7) to accept unescaped user input and persist it in the database as an object called *dataitem*. A dataitem consists of two fields: a *name* that identifies the data and a *payload* which stores user submissions. The unsafe form field in Figure 7 allows users to upload HTML and JavaScript. While the field is obviously vulnerable, it maps to real world cases of sXSS [24, 26] and provides representational clarity for demonstrating attacks.

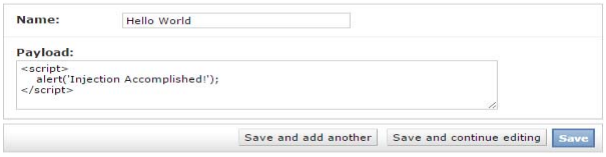

Figure 7: Dataitem with unsafe sXSS vulnerable form field

The second component in the testbed is a hybrid app integrated with the RESTful API. Integrating hybrid apps with APIs is a common practice in real world apps [5, 10], especially for companies with a web-based storefront seeking to increase presence in the mobile application space by porting their existing web apps into hybrid ones. Following this idea, our prototypical app, shown in Figure 8, loads instances of *dataitem* from the RESTful API and displays the *name* in a feed-like structure. If a user clicks the button next to a name, the user is taken to a page that loads the dataitem and its (potentially malicious) content.

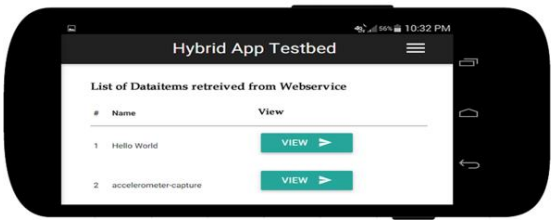

Figure 8: Feed of data in hybrid app loaded from REST API

In the case of unsafe sXSS affected content, such as the simple "Hello World" dataitem in Figure 7, the sXSS attack will launch in the hybrid app whenever the HTML content is loaded into the internal web view page. This is shown in Figure 8. Here, the alert message "Injection Accomplished" is displayed in the app. This design mirrors unsafe practices used load and display external HTML, e.g., advertisements.

Figure 9: Execution of *Hello World* attack in Figures 7 and 8

The last component of the testbed is another element built into the RESTful API for mocking the capabilities of a webserver that an attacker might use to collect and log data gathered from vulnerable hybrid apps. Called *logitem*, it accepts 6 data fields: an *id* to uniquely identify the event, an *attack* field to define which attack occurred in the app, a *result* field for capturing arbitrary string, JSON, or file data uploaded to the attacker's server, a *datetime* field for temporally correlating logs, and three fields that identified the type of device attacked. The last three fields are *device_name*, which captures the network id of the device, *platform*, which defines the OS type (e.g., Android), and *platform_version* (e.g., 4.4.4). Discussed in the next section, *logitems* were used to log the results, *r*, of the identified amalgamated attacks, discussed in Figures 4, 5, and 6.

IV. EVALUATION

To evaluate the overall efficacy of the approach, two questions (RQ1-RQ2 below) were considered.

RQ1. How effective is the testbed for assessing vulnerabilities?

RQ2. Does the test bed yield useful insights for assessing the risk of amalgamated attack vectors? What differences (if any) are encountered on different OS versions / devices?

To answer Questions RQ1 and RQ2 we deployed each of the amalgamated attack vectors test cases enumerated in Section III.C using the testbed from Section III.D and analyzed the results on several smartphones using different versions of Android. Section IV.A discusses the specific details and findings found in each of the three attack vectors. Analyzing Attack Vectors in the testbed

Prior to examination or analysis, three different Android devices, were selected for testing. Due to resource constraints, iOS and Windows phone platforms were not tested. The following devices and OS versions were used:

- Samsung Galaxy S4 (SGS4) / Android 4.4.4,
- Samsung Galaxy S4-active (SGS4a) / Android 4.4.2,
- and Samsung Galaxy Light (SGL) / Android 4.2.2.

A. *Analysis of sXSS-geolocation test case*

To begin the analysis, the sXSS-geolocation attack vector outlined in Figure 4 was selected. When the script executed in the app testbed, an Ajax call would make a GET request to the awaiting attacker web service running on the

backend of the testbed. The GET request included a new instance of logitem (as discussed previously in Section III.D). In this case the result field in logitem was formatted as follows to elucidate the attack outcomes.

r ={'*latitude*",'*longitude*','*accuracy*','*heading*','*speed*','*timestamp*'}

In the case of the SGS4 and SGS4a, the sXSS-geolocation attack successful executed and logged the coordinate position of the phone to the mock attacker server. This proved that the attack vector was feasible and provoked further examination. In the case of the older SGL, the test case did not execute at first. In later tests, the attack succeeded on the SGL as well. It is unclear why the attack did not initially execute on the SGL, but one possibility might be that another app was using the GPS at the time.

Given the success of the sXSS-geolocation attack in the lab, the test case was examined in a real world environment to determine the type of information an attacker might be able to gain. To do this, a driving scenario was constructed in which the test case would run in the testbed app every 10 seconds while traveling around Omaha, NE. A JavaScript *setInterval* loop was added around the code in Figure 4 to automatically run the script every 10 seconds after its initial injection. Figure 10 shows data (41 unique GPS coordinates across 305 samples) collected over the 11.74 km (7.3m) route. Each red pin icon is a particular GPS location captured in a logitem (*r*) in the attacker web service. Points were time correlated by their timestamp and Table 2 shows 5 points across 40 seconds of travel time. Each is labeled by id in Figure 10 to show temporal ordering.

Table 2: Small subset of raw data plotted in Figure 10

Log ID	time (GMT)	latitude	longitude
219	6:28:21 PM	41.25179	-95.9343
218	6:28:11 PM	41.2524	-95.9399
217	6:28:01 PM	41.25268	-95.9423
216	6:27:51 PM	41.2536	-95.9512
215	6:27:41 PM	41.2536	-95.9512

GPS accuracy was also investigated to determine how precise (and therefore how harmful) collected attack data might be. Across the samples, the GPS coordinates were accurate to within (on average) 965m of the actual path traveled (shown in blue on Figure 10) using the lowest accuracy *course* location setting in Android. The red circles in Figure 10 show the accuracy estimate for each logged point. Running the attack on the SGS4a travelling the same path using the *fine grained* high accuracy setting yielded values that were within 10m of the actual path. Certainly with the fine, but even with the course grained data an attacker can gain specific information about a victim's movements, raising serious privacy and safety concerns.

Ultimately addressing RQ1, the testbed provided useful data collection capabilities on both the application and web service ends of the spectrum which enabled the test case to be evaluated and expanded upon. With RQ2, the testbed provided insights on the scope and specificity of data collection, which helped assess how an attacker might use the data. Due to the feasibility and the potentially harmful impacts, sXSS-geolocation should be considered to be a high risk amalgamated attack vector.

B. Analysis of sXSS-accelerometer test case

The next attack, sXSS-accelerometer, examined deploying the test case in Figure 5 to the different devices and observing the results. Initially a single accelerometer data collection was taken. All of the devices executed the injected script and passed a result to the waiting attacker web service. Given the proof of concept test case succeeded, a *setInterval* loop was added to automatically collect data. The question was: How frequently could accelerometer data be logged and sent to an attacker's server? This question stems from concerns raised by research showing how accelerometer data can be used as a biometric [32], for reverse engineering movement-based lock screen passcodes [28], and even as a keylogger for recognizing and logging keyboard-based password or other textual entry [28]. Hence, if a high polling frequency could be reached in the testbed, then sXSS-accelerometer attacks could have a potentially very high impact.

To answer this question, which really addresses RQ2, the setInterval loop was set to repeat every 50 milliseconds (ms), i.e. polling frequency of 20hertz (Hz), to determine if an attack could collect enough data to attempt keylogging analysis. Figure 11 displays a graph of 600 points collected by the attacker web service over a 30 second time interval. It shows the fine granularity of the collected data, as captured by the platform. In the graph the x position is represented by a grey line, the y position is shown in blue, and the z position is shown in orange. These map to a Cartesian space overlaid on the device's screen. Hence, when the device is rotated so that the speaker is vertical relative to the ground, the gravity due to acceleration (9.8m/s2) switches from z to x. The large spike in the middle of the graph around 18.5s corresponds to the CPU spike when the device was shaken.

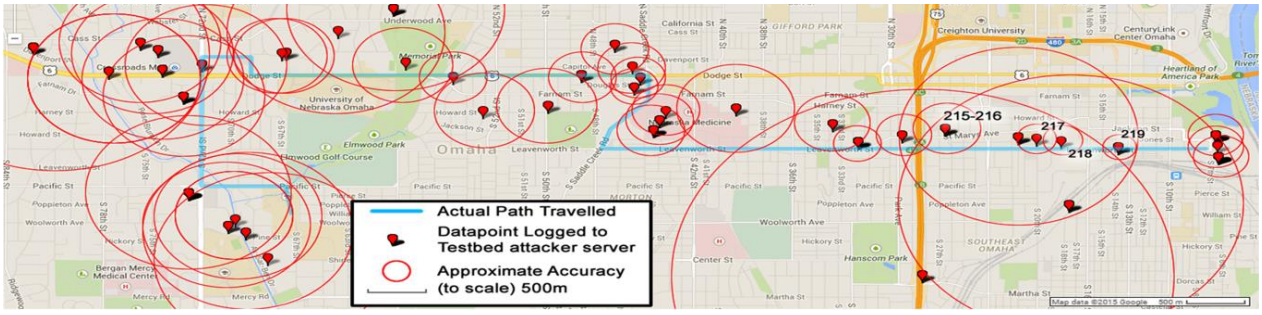

Figure 10: Geolocation data collected by mock attacker web service in testbed as a result of executing the sXSS-geolocation test

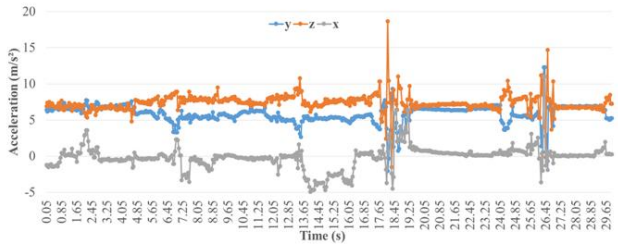

Figure 11: Captured accelerometer data over a 30s interval

Clearly, from this detailed information, an attacker can gain potentially significant insight into user behavior. This combined with the fact that the attack succeeded on all devices indicates that this type of attack vector should be considered highly impactful as it could result in loss of confidentiality for the user's textual input.

C. *Analysis of sXSS-camera test case*

The last attack examined was the sXSS-camera test case from Figure 6. This scenario involved an attacker taking a picture with the user's camera without permission. Again, the test case was deployed to each device. Unlike the previous vectors the attack did not succeed. Instead, we found was that Cordova invoked the default camera management app whenever the *camera.getPicture* method was called. Camera management apps provide the same user interfaces to all camera using applications. In the case of sXSS-camera, this meant that when the attacker's script executed, the user was prompted to snap a photograph. In the event that the user proceeded to take a photo and select the *save* option, the attack did succeed and successfully encoded the image in base64 and transmitted it to the attacker web service. Otherwise it failed, meaning that an attacker does not have control of a device camera and, therefore, *cannot* repeatedly snap photos of the user without any user input - a positive finding for the security of the Cordova hybrid TNC.

V. CONCLUSION

This paper investigated security concerns with hybrid mobile apps, defined a five step process for discovering hybrid specific vulnerabilities, enumerated three forms of amalgamated attacks that can specifically target and exploit hybrid apps, and created a testbed platform for accessing vulnerabilities. Findings from experimentation indicated that two of the three identified attacks, sXSS-geolocation and sXSS-accelerometer, are highly critical and apply to hybrid apps operating in current mobile operating systems. Future work will focus on extending existing efforts [12, 14] to combat these amalgamated attack vectors in hybrid apps. The testbed in this paper supports future investigations and aides security experts in assessment and mitigation efforts.

REFERENCES

[1] "Platform Comparison of Historical Web Views in the US From August 2012 to August 2014," StatCounter Global, 2014.

[2] "The HTML5 vs. Native Debate is Over and the Winner is…" Telerik, 2014.

[3] "About Apache Cordova," Apache Software Foundation, 2013.

[4] J. M. Wargo, *PhoneGap Essentials: Building Cross-Platform Mobile Apps*: Addison-Wesley, 2012.

[5] A. Rodriguez, "Restful web services: The basics," *IBM developerWorks,* 2008.

[6] Y.-w. Huang, F. Yu, C.-h. Tsai, C. Hang, D.-t. Lee, and S.-y. Kuo, "System and method for securing web application code and verifying correctness of software," Google Patents, 2013.

[7] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications," in *Proc. of the 28th Annual Computer Security Applications Conference*, 2012.

[8] P. H. Phung and L. Desmet, "A two-tier sandbox architecture for untrusted JavaScript," in *Workshop on JavaScript Tools*, 2012.

[9] A. K. Jain and D. Shanbhag, "Addressing security and privacy risks in mobile applications," *IT Professional,* vol. 14, pp. 0028-33, 2012.

[10] T. Mikkonen and A. Taivalsaari, "Apps vs. open web: The battle of the decade," in *2nd Wksp on Soft. Engi. for Mobile App. Dev.,* 2011.

[11] "Cordova plugins " 2015. https://github.com/apache/?query=cordova.

[12] M. Shehab and A. AlJarrah, "Reducing Attack Surface on Cordova-based Hybrid Mobile Apps," Int'l Wksp on Mobl Dev. Lifecyc., 2014.

[13] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc.of ACM conf. on Sec. and Priv. in Wireless and Mobile Networks*, 2012.

[14] J. Xing, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation," in *Conf. on Comp. and Comm. Sec.*, 2014.

[15] "Native, HTML5, or Hybrid: Understanding Your Mobile Application Development Options," Salesforce, 2014.

[16] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating cross-platform development approaches for mobile applications," in *Web Infor. Sys. and Technologies*, ed: Springer, 2013, pp. 120-138.

[17] A. Juntunen, E. Jalonen, and S. Luukkainen, "HTML 5 in Mobile Devices -- Drivers and Restraints," in *46th Hawaii Int'l Conf. on System Sciences*, 2013.

[18] E. J. Angulo Cevallos, "Case study on mobile applications UX: effect of the usage of a crosss-platform development framework," ETSI Informatica, 2014.

[19] G. Hurlburt, J. Voas, and K. W. Miller, "Mobile-App Addiction: Threat to Security?," *IT Professional,* vol. 13, pp. 9-11, 2011.

[20] R. Balebako and L. Cranor, "Improving App Privacy: Nudging App Developers to Protect User Privacy," *IEEE Sec. & Privacy,* vol. 12, pp. 55-58, 2014.

[21] R. Hunt, "Security testing in Android networks - A practical case study," in*, 19th IEEE Int'l Conf. on Networks*, 2013.

[22] D. Jaramillo, R. Smart, B. Furht, and A. Agarwal, "A secure extensible container for hybrid mobile applications," in *Proc. of IEEE Southeastcon*, 2013.

[23] A. Nadkarni, V. Tendulkar, and W. Enck, "NativeWrap: ad hoc smartphone application creation for end users," ACM Conf. on Security and privacy in wireless & mobile networks, 2014.

[24] "The Open Web Application Security Project, Web Application Security Risks," OWASP, 2014.

[25] "Common Attack Pattern Enumeration and Classification (CAPEC)," MITRE, 2014.

[26] Y. Wang, Z. Li, and T. Guo, "Program slicing stored XSS bugs in web application," *Int'l Symp. on Theor. Aspects of Soft. Engi.*, 2011.

[27] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith, "Practicality of accelerometer side channels on smartphones," in *Proc. of 28th Comp. Sec. App. Conf.*, 2012.

[28] D. Damopoulos, G. Kambourakis, and S. Gritzalis, "From keyloggers to touchloggers: Take the rough with the smooth," *Computers & security,* vol. 32, pp. 102-114, 2013.

[29] R. A. Rouse, "Is Someone Watching You Through Your Webcam?," 2012.

[30] F. Roesner, B. T. Gill, and T. Kohno, "Sex, lies, or kittens? investigating the use of snapchat's self-destructing messages," in *Financial Crypto. and Data Sec.*, Springer, 2014, pp. 64-76.

[31] J. Forcier, P. Bissex, and W. Chun, *Python web development with Django*: Addison-Wesley Professional, 2008.

[32] D. Gafurov, K. Helkala, and T. Søndrol, "Biometric gait authentication using accelerometer sensor," *J. of comp.,* vol. 1, pp. 51-59, 2006.