Mobile Application Web API Reconnaissance: Web-to-Mobile Inconsistencies & Vulnerabilities

Abner Mendoza, Guofei Gu
Texas A&M University
abmendoza@tamu.edu, guofei@cse.tamu.edu

***Abstract*—Modern mobile apps use cloud-hosted HTTP-based API services and heavily rely on the Internet infrastructure for data communication and storage. To improve performance and leverage the power of the mobile device, input validation and other business logic required for interfacing with web API services are typically implemented on the mobile client. However, when a web service implementation fails to thoroughly replicate input validation, it gives rise to inconsistencies that could lead to attacks that can compromise user security and privacy. Developing automatic methods of auditing web APIs for security remains challenging.**

In this paper, we present a novel approach for automatically analyzing mobile app-to-web API communication to detect inconsistencies in input validation logic between apps and their respective web API services. We present our system, *WAR*Droid, which implements a static analysis-based web API reconnaissance approach to uncover inconsistencies on real world API services that can lead to attacks with severe consequences for potentially millions of users throughout the world. Our system utilizes program analysis techniques to automatically extract HTTP communication templates from Android apps that encode the input validation constraints imposed by the apps on outgoing web requests to web API services. *WAR*Droid is also enhanced with blackbox testing of server validation logic to identify inconsistencies that can lead to attacks.

We evaluated our system on a set of 10,000 popular free apps from the Google Play Store. We detected problematic logic in APIs used in over 4,000 apps, including 1,743 apps that use unencrypted HTTP communication. We further tested 1,000 apps to validate web API hijacking vulnerabilities that can lead to potential compromise of user privacy and security and found that millions of users are potentially affected from our sample set of tested apps.

I. Introduction

The proliferation of mobile devices has resulted in an extensive array of mobile applications (apps) that serve diverse needs of our connected society. Today's modern lifestyle increasingly depends on mobile apps that serve a wide spectrum of functionality including military applications, critical business services, banking, entertainment, and other diverse functionality. Mobile apps are often built as front-ends to services hosted in the cloud infrastructure and accessible through web API services. The web platform, through the use of HTTP and HTTPS [1], serves as the main conduit for communication between mobile applications and their respective web API services. Previous research work in the mobile space has mostly focused on security and privacy of the mobile device and data stored locally on the device. However, remote HTTP-based services form an integral part of the mobile application ecosystem and deserve similar scrutiny with regard to security and privacy concerns. This fact is evidenced by the placement of *Weak Server Controls* as the top vulnerability in the OWASP top 10 mobile vulnerabilities [2].

The ease at which mobile apps can be built using modern tools means that even inexperienced developers can deploy mobile applications that integrate with new or existing cloud services. Additionally, a number of well established cloud infrastructure service providers such as Amazon AWS and Microsoft Azure provide pre-packaged mobile cloud solutions that mobile application developers can integrate into their apps with just a few lines of code. This approach promises to abstract the backend logic and maintenance, freeing the developers to focus on their mobile app functionality and user experience. These services often include ready-made solutions for common tasks such as data storage, user authentication, e-commerce, social-media integration, and push notifications. Cloud services are provided via specialized software development kits (SDK) and application programming interfaces (APIs) for easy integration. However, this fast paced development is often done without full consideration of security implications. Often, there is no robust security design or guidance of the application integration with the pre-packaged components, exposing many mobile applications to exploitation. Similarly, homegrown (proprietary) web API services are often deployed at a fast pace, without much consideration of the security impact of the design decisions and how developers will integrate the API service into their apps.

In every instance, the decoupled mobile web service API architecture mandates that input validation logic is done equally at both the client and server side. This creates a heightened dependency on robust consistency between two disparate platforms: web and mobile. In this work, we are motivated by the insight that the logic implemented in the mobile client can be used to inform audits of server-side APIs. We observe that it is non-trivial to ensure full and robust consistency between app-based and server-based validation routines, resulting in inevitable mismatches between client and server implementations of input validation logic. We introduce the concept of *Web API Hijacking* to generalize these types of threats, and develop an approach to uncover instances of Web API Hijacking. Web API hijacking describes a class of server-side attacks that seek to exploit logic inconsistencies and gain unauthorized access to protected or private server capabilities and resources where robust validation controls are not

consistently implemented. These attacks leverage parameter tampering vulnerabilities on the web platform [3], discoverable through careful analysis of mobile application code logic.

While there have been extensive works in the past to address web server problems such as SQL injection, cross site scripting, and other traditional web security problems [4], [5], today's mobile-first web services are often implemented with scalability as a top priority [6]. As we show in this work, mobile app architectures often defer validation and security to the client-side. Weak server-side input validation is by no means a new problem, but it has received little to no attention, especially from the aspect of integration with mobile applications.

Inspired by previous work in web parameter tampering vulnerabilities [3], [7], and advances in mobile application program analysis techniques, we devise a novel approach, called *WAR*Droid, to analyze mobile application web API interaction, and uncover attack opportunities that can lead to compromise of user security and privacy. *WAR*Droid is a framework that implements semi-automatic **W**eb **A**PI **R**econnaissance to analyze validation routines that make up requests to web API services from an app. *WAR*Droid can then uncover inconsistencies between app-based and server-based validation logic that can lead to Web API Hijacking attacks. *WAR*Droid implements a network-aware static analysis framework that systematically extracts the web API communication profile and logic constraints for a given app. It then infers sample input values that violate the implemented constraints found in the app. *WAR*Droid then analyzes app-violating request logic on the server side via blackbox testing, and is able to uncover instances where web API services do not properly implement input validation. We highlight several interesting case studies that show the potential real world impact of these weaknesses on the mobile ecosystem, affecting even high profile mobile apps used by millions of users.

We enable comprehensive analysis of each individual application with regard to its app-to-web communication template to uncover Web API Hijacking opportunities. Our system primarily focuses on extracting the application layer constraints and interactions that occur over HTTP(S). Our System advances state of the art research toward providing a comprehensive characterization of HTTP-based API communication, especially including the constraints that relate to UI-level input fields that flow to remote web APIs. We formulate our problem in terms of the logic constraints that are imposed by application code, and use it as a model to characterize expected server-side logic.

In short, the contributions of this paper are as follows:

- We develop the first systematic approach for detecting mobile-to-web validation logic inconsistencies that can lead to attacks. We call this class of attacks *Web API Hijacking*.
- We provide a novel mobile application Web API communication analysis framework, called *WAR*Droid, that can extract details of mobile application cloud service interactions. Our approach implements a novel network-aware app-to-web static analysis framework that can assist in uncovering Web API Hijacking vulnerabilities.
- We identify Web API misuse patterns and provide case studies of analysis and discovered vulnerabilities in real world applications. We show concrete exploit opportunities that are uncovered from real world apps that could lead to severe consequences for app developers, users, and app service providers.

II. Problem Statement

While mobile apps may have robust input validation and access control logic implemented in their native code, those are often not equally replicated on the server side for data sent to a web API. As a result, an attacker can bypass client-side controls and exploit a web API service to extricate data or inject malicious data without proper authorization. This is noted in the recent paper by Sudhodanan et. al. [8].

In this paper we aim to systematically study and (semi-)automatically detect the inconsistencies between data validation logic in a mobile app and data validation logic implemented at a remote web API server. While this is inspired by previous work on web parameter tampering [3], [7], we address challenges in uncovering web API data validation logic in mobile apps, where client-to-server communication is not as inherent as on the web platform. We also highlight the real world security impact of inconsistent app-to-web validation on the mobile ecosystem caused by loose coupling between mobile and web validation logic.

Transactions between mobile apps and web API services require careful coordination of data validation logic to ensure that security controls are consistently implemented. For example, if a mobile app restricts the data type of a user input field, we expect that the server should also implement a similar restriction to ensure consistency. Unfortunately, it is difficult or impossible to ensure complete consistency between controls built into the mobile app and controls actually enforced at the server side. In many cases, the server should enforce more constraints than the client (such as enforcing uniqueness of usernames, for example). In this paper, we assume that the server is at least as strict as the client. Remote web API service implementations are often shared among different user agents (mobile and browser), giving rise to further inconsistencies in the implementations of the application logic between different apps that use the same backend web API. For the sake of scalability, web APIs may even skip input validation and defer that job to the apps. It is also not always feasible for remote web API services to authenticate all clients, giving rise to various replay attacks where attackers can impersonate legitimate clients or access functionality intended for legitimate clients without authentication or authorization [8].

The scalability requirements of remote web API services often mandate that the implementations are generic so that multiple client platforms can be supported. However, this can lead to serious security threats when the web API is security-critical, or privacy-sensitive, but defers validation to the client side. We address this problem in the context of the

mobile ecosystem. While we use the Android framework for our research evaluation and testing, it is important to note that Web API Hijacking is not intrinsic to any flaw in the Android framework itself. Rather, this problem applies to any mobile app that follows the model of using web API server endpoints, such as those that use the SaaS app model. This is a vulnerability that exists primarily on the web platform through parameter tampering, but has transitioned into the mobile ecosystem, enabled by the subtle mismatch and inconsistency of data validation logic between the native mobile platform and the web platform.

A. Motivation

Why are we using the mobile platform to uncover potential web server vulnerabilities? Mobile web API services are not tightly coupled with the app front-end, but we posit that mobile apps implement validation logic that serves as a model of expected server-side validation logic implemented by the web API. This is especially true for web API services that are tailored for mobile app consumption and do not have an accompanying traditional web application interface. However, due to the reliance on HTTP(S), any client capable of HTTP(S) communication can therefore communicate with the web API service. If the web API service does not properly validate request data, and instead defers the responsibility to the mobile app, an attacker can hijack the API functionality meant exclusively for the mobile app.

Apps with web API hijacking vulnerabilities are usually not malicious and usually implement fairly robust data validation. However, the inconsistency lies in how the web API server replicates that validation. Attackers in our threat model do not attack the apps themselves but can use the app to understand the web API communication profile and leverage that knowledge to coerce the server to conduct malicious activities, expose sensitive user data, or gain unauthorized access to privileged functionality.

To determine if a given web API endpoint is vulnerable, our analysis finds feasible data flows in the app that generate HTTP(S) requests to the web API server and process some response from the server. By extracting the path constraints on those data flows, we can infer the data validation model of the app for a particular web API endpoint. By generating similar requests outside the app that would violate the app validation logic, we can uncover inconsistencies between the app and server logic. These web API endpoints are referred to as 'hijack-enabled'. By exploiting the inconsistencies in these hijack-enabled endpoints, an attacker can compromise the security and privacy of user data or API functionality.

We consider that a mobile app's input validation logic with respect to its interaction with a web API primarily consists of three steps:

1) Sanitize and Validate input, and generate HTTP(S) Requests to the Web API Server.
2) Reject Invalid Input.
3) Process Web API Server Responses.

B. Formalization

More formally, a mobile app M_a generates a request R_a using input strings S and sends it to the remote web API server for processing. Before sending the request, the application must enforce certain constraints C_a on the strings in S, and abort the request if the constraints are not satisfied. Formally, the constraint checking code can be expressed as a function $C_a(S) \rightarrow \{true|false\}$, where true means that the inputs satisfy the constraints, and false means that the inputs do not satisfy the constraints. We denote the constraint checking function at client app as C_a, and the corresponding function at the server as C_s. Therefore, we assert that if $C_s(S) = true$, then $C_a(S) = true$. That is, if the server constraints on an input evaluate to true, then the client constraints on the preceding web request input should also evaluate to true.

We observe the following rules about constraint checking between the app and the server:

- An input accepted at the server does not violate the constraints at the client. $C_s(S) = true \Rightarrow C_a(S) = true$
- An input that is rejected at the client, should be rejected at the server. $C_a(S) = false \Rightarrow C_s(S) = false$

These rules ensure consistency between validation at the mobile app and at the web API server. We note that an input that is valid in the app may be invalid at the web API server because C_s may be more restrictive than C_a in certain situations. For example, when registering a user account, the server can additionally validate the username for uniqueness. Also, if $C_s(S) = false$ (the server rejects the input), then it does not matter if the client accepts it or not. We are targeting instances where $C_s(S) = true$ AND $C_a(S) = false$.

A violation of these consistency rules could cause the API to be hijack-enabled and exposed to the possibility of being attacked. Specifically, a potential vulnerability exists if the web API server accepts an input that would be rejected by the client side constraints. Such problems can lead to compromise of user data security and privacy, denial of service for all apps that rely on the web API, and other serious consequences to the mobile ecosystem that can lead to monetary losses.

Therefore, our problem is reduced to evaluating the consistency of the constraint checking functions between the app and the web API server. In this work, we treat the app as a whitebox, and the web API server as a blackbox. Since C_s is at least as restrictive as C_a, we can model C_s by precise analysis of the app. Using a derived constraint formula, we can uncover inconsistencies between both platforms by evaluating the responses R_s generated from requests R_a sent to the web API by our test framework. By identifying and further evaluating web API endpoints that show inconsistencies, we are able to uncover web API hijacking opportunities.

C. Threat model

We assume a network attacker as described in [9]. Our attacker has access to the mobile application and can reverse engineer the source code. Additionally, the attacker can observe and manipulate his own network traffic if necessary.

We assume the attacker has a means of sniffing data from legitimate mobile user devices, but he also operates his own mobile device and can observe, modify, and decrypt his own HTTPS traffic. Our attacker is also a legitimate mobile application user. This attacker has full access to the Android client layer through which he can interact with the remote web API server as a legitimate user would.

Attacker Capabilities: An attacker seeks to gain unauthorized access to sensitive resources by leveraging one of the following methods on publicly exposed web API endpoint functionality:

1) GET sensitive data using an API endpoint.
2) POST[1] to data stores using the API endpoint.

Web API hijacking gives the attackers unauthorized access to perform privileged actions on the API server side, and the ability to influence reflected data to various apps and other clients that may access the web API. This is a highly attractive target for an attacker because it is a single point of attack that can affect multiple users. For example, an attacker can leverage capability 2 to write data to a data store that in subsequently read by a website that may display the data to users. If the attacker is able to embed malicious code into the data store, that code would be reflected to the user if the consuming website does not properly sanitize the data.

III. Background

Android apps are packaged as APK files, which contain all the resources necessary to execute the application on the Android Framework. *WAR*Droid starts by extracting the resources from a given APK file and preprocessing those resources for further analysis. The DEX class files are further converted to an intermediate representation called Jimple [10] that lends itself to static analysis using Soot [11]. Additionally, *WAR*Droid inspects the XML resource files that represent the user interface and user input elements for different Activities of the app. In Android, Activities represent the user interface components of an app.

We focus on the Android platform due to its open source nature, and we restrict our analysis to apps that use the HTTP protocol for communication with a web API server. One of the main functions of *WAR*Droid is therefore to model the HTTP(S) communication of the app with respect to different web API services that may be used by the app. An HTTP transaction consists of a Request and a Response pair. A Request is modeled in the output templates as a tuple containing <*Method, Scheme, Domain, Path, Parameters, Headers, Body*>. Similarly, we model a Response as <*Status, Headers, Body*>. Apps may directly open an HTTP stream through the APIs provided by the framework, or they may use an intermediate SDK which abstracts the framework API utilization.

[1]We consider other less common HTTP verbs such as UPDATE and PUT as having similar core functionality

Listing 1. Basic HTTP Request Generation Code

```
1   protected String doInBackground(strings) {
2           URL url;
3       HttpURLConnection urlConnection = null;
4       //create request
5           url = new URL(strings[0]);
6           urlConnection =
                (HttpURLConnection)
                url.openConnection();
7           int responseCode =
                urlConnection.getResponseCode();
8           if(responseCode ==
                HttpURLConnection.HTTP_OK){
9               //response handling code
10          }
11      return null;
12  }
```

The code listing shows a typical HTTP request method in Android apps. This is encapsulated within a class that may extend `AsyncTask` and is called using syntax such as '`new GetMethodDemo().execute(serviceURL);`'. *WAR*Droid identifies the HTTP interface at line 6 as a point of interest (POI) and proceeds with backward program slicing to identify all parameters and UI elements to which the connection has a dependency. Intuitively, this exercise encapsulates the full dependency graph that makes up the web request. The observation is that forward taint propagation from line 6 tracks objects that originate from a web API in a response and backward tainting tracks objects that are used to generate a request to a web API. We refer to such HTTP access functions as *Points of Interest* because they separate the forward and backward program slices. Forward taint propagation reveals the data dependency for objects related to response message processing, and backward tainting identifies objects that make up the URI, request method, and body of a web API request. As a result, the problem is now reduced to searching and identifying POIs from Android and Java APIs, which is much more feasible than performing a full analysis of the entire app call graph and tracking all network-related objects.

Thereafter, the path constraints within the slices are analyzed to extract the web API request templates for which test HTTP requests can be generated and further evaluated. In particular, *WAR*Droid identifies the constraints associated with the web API request path Parameters, Headers, and Body, and can generate test inputs for both valid and invalid API requests.

IV. Approach and Challenges

First, we extract the web API communication templates from mobile apps that encode the input constraints enforced by the app for web API communication. We implemented a network-aware taint analysis approach to extract program slices that represent the web API request generation functionality of the app. We employed existing program analysis tools and techniques to fit our problem and address known inherent challenges. Second, using the extracted constraint templates, we implement a blackbox testing component that

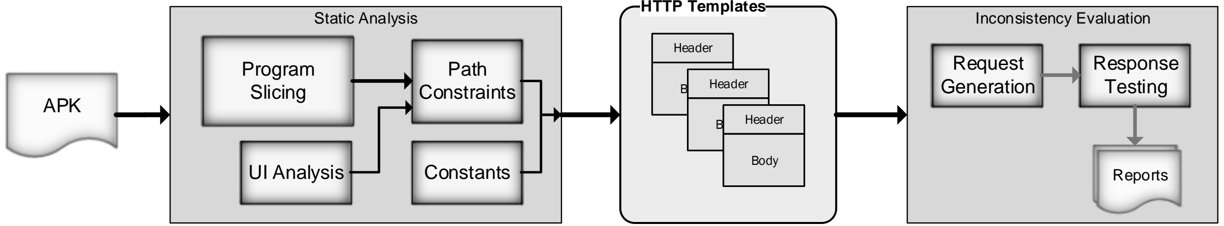

Fig. 1. Overview Architecture of the *WAR*Droid Framework

assesses the consistency between the app validation logic and the web API server validation logic. Using the constraint relationship rules between the app and the web API server, we can generate requests that we expect to be rejected by the server. The intuition is that the app validation logic should be consistent with the web API server validation logic. Any inconsistencies uncovered are opportunities that attackers may be able exploit and can lead to a violation of the application security properties. *WAR*Droid generates both valid an invalid requests that can be replayed to the server to evaluate our hypothesis using a simple cross-validation approach to reduce false positives.

A. General Challenges

The challenges of the whitebox analysis approach lie in the non-trivial nature of static analysis and its inherent limitations. Fortunately, these have been solved by existing work [12], [10], [13], [14], [15]. We utilize these existing work in *WAR*Droid. Still, we address additional challenges in analyzing app-to-web communication.

Modeling Server Logic. Without access to the back-end server code, we must devise a methodology that effectively utilizes the mobile application and the observed HTTP communication logic to the backend API service to model the expected server logic and constraints. This is exactly what an adversary would also have access to, which lends some practicality and feasibility to our analysis approach.

Incomplete Access. While the mobile application binaries are readily available through the open marketplace model of Android, we do not have access to the server side API implementation for a precise comparison. Therefore, we must rely solely on the mobile app and formulate an estimated model of the server logic. Our system must therefore ensure high code coverage and accurately infer the web API request message constraint formula. To overcome this challenge, we employ robust static analysis tools that ensure high coverage and accuracy.

Low Coverage. To increase accuracy and coverage, and further optimize our analysis, we implement symbolic execution to model the input validation logic through path constraints [16]. This allows us to efficiently reason about the constraints of web API requests.

Symbolic execution utilizes the control flow graph, storing an accumulating path condition as the data dependence moves along the execution path. The path condition at the point of interest represents the constraint formula that we later utilize to reason about valid and invalid inputs to compare validation consistency. For our purposes, the point of interests are the HTTP(s) buffers in the mobile application used to communicate with remote web APIs.

However, symbolic execution can be slow, and analyzing an entire app can lead to unnecessary code paths being explored. Since not all the app execution paths are related to web API requests, we must filter only the paths that are of interest to reduce the analysis space, while still maintaining precision and accuracy.

Search Space. To reduce the search space and optimize the analysis, we filter the paths to analyze only those that utilize an HTTP library or system API. We focus on identified points of interest (POIs) that generate or process web API HTTP(S) messages. Fortunately, there is a small set of HTTP(S) libraries and HTTP network buffer APIs that we can use as our starting point for extracting HTTP communication templates.

Validating Inconsistencies. An important goal of *WAR*Droid is to validate inconsistencies in a semi-automated fashion. This requires generation and replay of web API requests and analysis of the corresponding responses. Some human intervention is necessary in formulating proper requests. It is also non-trivial to analyze server responses based on simple heuristics to make a determination of success or failure of the request. A simple approach could be to evaluate HTTP status codes, but that would lead to many false negatives. *WAR*Droid overcomes this challenge by implementing a response analysis approach that compares several response traces of known valid requests with suspected invalid requests. This approach is inspired by a similar method used in [3].

V. System Architecture

The general system architecture is depicted in Figure 1. The primary goal of *WAR*Droid is a novel application of static taint analysis and symbolic execution to uncover web API input validation constraints and reason about web API hijacking opportunities by evaluating inconsistencies. To achieve this goal, we extend Flowdroid [15] to comprehensively analyze

web-related code paths and constraints in apps that lead to network APIs that generate HTTP(S) messages. We therefore model the web API's server-side validation logic using the mobile application validation logic. We can then detect inconsistencies by deriving invalid API requests that fail in our mobile application model but does not fail when testing on the actual server. We characterize the application validation logic as a symbolic path constraint on a static abstraction of the web request functionality which is a subset of the program dependence graph (PDG) of the app. We represent the constraints in the format of Z3 [13] and utilize the Z3-Str library [17] to generate both valid and invalid concrete API requests for testing through message replay.

*WAR*Droid takes the application APK package as input and produces possible web API hijacking opportunities as output. First, we model the mobile app's web API communication into HTTP message templates. To accomplish this, we utilize program analysis techniques that analyze the app to extract the program slices that generate HTTP requests from each POI. The main task is to track all dependencies that eventually flow to network buffers through particular Android framework APIs. This allows us to extract the relevant path constraints and reason about the web API requests generated by the app.

To this end, our system extracts and analyzes the program slices that generate and process HTTP messages using data dependency analysis. We augment the resulting program dependence graph slices with information from the user interface (UI) resources in the app that define additional constraints imposed by UI elements on user input data that eventually make up part of the web API request.

Interesting code paths are those that include a conditional flow that determines the final API request endpoint. These conditions encode constraints that are our main targets for evaluation of inconsistencies. We theorize that this constraint logic is representative of the web API logic intended on the server, but not always implemented with due diligence. First we must understand the normal intended flow, and the semantics of the checks that control the flow to different web API end points. Armed with this information, we can then reason about request messages that would violate the extracted constraints and test if they are accepted by the server. In some cases when the server is not available for testing, or would cause harm, we can still infer success by evaluating the response processing constraint logic of the app that corresponds to the code path under consideration. This correlates to the constraints extracted from the forward static analysis starting at each POI.

A. Static Analysis

*WAR*Droid implements program slicing to reduce the search scope and focus on web API related code paths. The first step is to extract a program slice using backward slicing starting at the web API call points, which are our POIs (Points of Interest). The key idea is to generate a concise representation of the subset of the program that communicates over the network. The slice is an approximation of the code necessary to enable the app-to-web API communication.

*1) **Program Slicing**:* Extracting program slices of interest requires identification and tracking of dependencies to network-bound APIs [18]. We focus on two sets of network message sending APIs as our starting points of interest (POIs). First, we identify the Android framework APIs provided for HTTP communication (e.g., HttpClient.execute). We utilize the semantic models of these APIs devised from [18]. We currently support java.net.HttpURLConnection, org.apache.http, android.net.http, android.volley, javax.net.ssl, and java.net.URL. Second, we also identify low level Socket APIs. When these APIs get called, they will directly perform connections to remote servers, which will then generate the response from the servers. With these method invocations as target points of interest, we can use taint analysis to identify the dependencies and call paths that invokes them.

For tracking web API-related data flows, we modify FlowDroid [15], which is a system built on Soot [11] and provides flow-sensitive, context-sensitive, and inter-procedural data flow analysis for Android apps. We also utilize the output from SuSi [14], which provides a comprehensive list of categorized sensitive APIs. We use the NETWORK and BROWSERONFORMATION entries as the input to FlowDroid. This allows us to identify all the API calls that can communicate using the network sensor or the browser. However, different from the traditional use of Flowdroid to track source to sink tainted paths, we utilize its taint analysis functionality to track taints in reverse from the sinks (POIs) until they converge to a UI element, an event handler, or initial definition. This gives us the ability to extract a web API-related program slice that represents the app's web API communication functionality.

Modifying tainting rules. For high accuracy and coverage, the program slices must contain all operations related to the web API communication from the POI. *WAR*Droid utilizes an open-ended taint propagation approach for this purpose. Flowdroid's default tainting rules implicitly handle forward taint propagation. However, for backward taint propagation we reverse the edge direction rules of the control flow graph to propagate the dependencies in reverse order starting from the point of interest. This is motivated by the approach taken by Extractocol [18], which applies inverted taint propagation rules in Flowdroid to swap the premise and conclusion of the rules. Our previous work in [19] similarly use inverted tainting rules for backward taint propagation.

More specifically, for assignment statements a tainted left-hand side taints the right-hand side, and for function calls the taint information of a callee's arguments is propagated to the caller's arguments. We track the tainted objects until there are no more objects to propagate, either at the object's definition or destruction.

A typical app also contains functionality that generates web requests to entities other than a web API endpoints of interest. For example, most ad libraries or analytics libraries have func-

tionality to communicate with backend servers, often through a web API. These are outside the scope of our investigation, and we therefore exclude popular ad and analytics libraries such as Google AdMob. The goal of the program slicing module is to generate program slices that directly relate to HTTP requests and response processing.

We use static taint analysis to track information flow to web API endpoints. However, unlike traditional static taint analysis whose primary goal is to determine the existence of data flow from taint sources to sinks, in this case we utilize it to track flows through network-bound objects for reconstructing web API message templates. Missing a single statement that has a relationship with the web API message would result in false negatives. Therefore, it is critical that we capture a robust representation of the dependencies that lead to the point of interest invocations. To this end, Flowdroid fits well into our approach since it effectively solves many of the shortcomings of static analysis.

Having extracted the network-aware program slices, we can build the program dependence graph and add additional augmentation, including constraints from UI elements.

*2) **Path Constraints**:* The constraint extraction module takes the filtered program slices as input. We leverage many of the existing functionality of Flowdroid, including call-graph construction, points-to analysis, def-use chains, and taint analysis. The goal of the path constraints module is to reconstruct the app's program dependence graph. Since the dependence graph constructed directly from Flowdroid cannot identify the edges that implicitly call the Android framework APIs, or does not consider UI elements, we must make additional augmentations to generate a complete set of path constraints for any given POI. We augment the built-in PDG output with additional information from the UI as well as implicit call information added by the Edgeminer results [20]. We refer to this as an Augmented Program Dependence Graph (APDG). Our approach ensures that both implicit and explicit call edges are added to our APDG, improving our accuracy and reducing false negatives.

To build the APDG, we analyze the Jimple IR slices from the Program Slicing module and start from each event handler (onCreate, onClick, onTextChanged, etc.), recursively adding the callee edges, including the implicit edges known from EdgeMiner. The results is a set of APDG's, each starting from the event handler functions. Furthermore, we analyze the UI resource files to identify the Activities and UI elements and connect them to their respective handlers. We augment our call graph with UI information so that we can utilize and capture constraints defined in the XML resource files, such as max data input length or data types.

Asynchronous Events: Asynchronous event handling is very common in Android programming. For example, an app may construct a portion of the web API request query string into an object and later, a click event would actually read the saved object to generate the HTTP request. This is not easily handled in static analysis, because the ordering of the events may be lost. For example, FlowDroid assumes an arbitrary ordering of these events, which can lead to a false negative or incomplete results. It results in a failure to identify the full dependencies across all events, resulting in an incomplete dependency graph. Our backward analysis approach in *WAR*Droid naturally solves this problem because it sequentially backtracks from the network API point of interest and naturally reconstructs the order of events as it moves backwards. It also captures implicit events with minimal effort. Dynamic analysis could not solve this problem because it lacks sufficient code coverage capabilities and would result in higher false negative rates.

To further reduce false negatives, we also utilize the results from Edgeminer [20] which previously solved the issue of asynchronous and implicit events and identified 19,647 additional callbacks, as opposed to only 181 identified by Flowdroid. Therefore, to enhance the coverage of *WAR*Droid, we directly use EdgeMiner's results and added the list to Flowdroid's configuration files. This adds support for many popular implicit callbacks commonly observed in web request calls and HTTP libraries, such as AsyncTask and others.

The resulting constraints are expressed in the format of Z3 [13], and we can then use the string solver (Z3-str [17]) to solve the constraints, or negation of the path constraint expressions.

*3) **UI Analysis**:* We also augment the program dependence graph using information extracted from the app's resource files that define the activity layouts. First, we must identify and correlate a given input element from the XML to the event listener in the program slice. We identify and tag the ID from the activity XML files, resource files, and the manifest file. Event handlers can be directly referenced in the XML, or the listeners contain a single callback that the framework uses to initiate the corresponding event handler. We extract the constraints imposed by UI elements, and tag the corresponding event handler node in the program dependence graph.

The UI elements impose additional constraints that may be defined in either of the resource XML files that configure the UI elements. *WAR*Droid handles constraints as defined in Table I.

TABLE I
SAMPLE UI CONSTRAINTS

Control	Constraint
Spinner	$x \in \{spinnerOptions()\}$
Checkbox	$x = \{true \| false\}$
RadioGroup	$x \in \{radioOptions()\}$
TimePicker	$isValidTime(x)$
DatePicker	$isValidDate(x)$
android:maxLength	$len(x) < n$
android:numeric	$x \in [0-9]$

*4) **Constants**:* Constants are defined as static strings used in the application code which represent authentication tokens that are required for each request to the web API. For example, apps that use the Amazon AWS sdk typically send the API authentication key with each request. This key is usually hard-coded in the source code. First, we use simple string searching heuristics to look for strings that resemble 64-bit encoded hash

keys. However, the keys are not always retrievable through such simple heuristics. To efficiently identify the constants, we leverage functionality built into Flowdroid inspired by [21]. Specifically, we use the inter-procedural constant-value propagator, which looks for static strings in static initializers or assignments. A value is considered static if the respective field or local variable is always assigned the same constant value. This fits exactly our use case. We tag these as required fields in the web request templates and augment the constraint formula to include these values.

Other Required Values. Most validation logic includes simple checks for required fields. This is the most simple form of input validation. *WAR*Droid must account for these instances. To address this challenge, we identify required parameters and their types using a simple set of heuristics. For example, when the constraint checks for a non-empty value, we tag the corresponding parameter as required. Another instance is where drop-down UI elements are used.

B. HTTP Templates

*WAR*Droid's program slicing approach effectively identifies the request/response slices in Jimple. The resulting slices only contain a small portion of all the app code, making the static analysis process very efficient. Using our extracted constraints along with additional augmentation information, we can build our HTTP templates for each web API endpoint. Algorithm 1 outlines the basic steps that we use to process a static analysis module output to generate our HTTP web API templates. The input consists of statements from a program slice S, with entry point e, and template T. The output of the algorithm is a set of constraint formulas, C, which concisely represent the web API templates.

Algorithm 1 Extracting Templates

```
1:  procedure TEMPLATE(e,S,T):
2:  begin
3:      Start at entry point e
4:      Get list of statements (stmt) from program slice S
5:      foreach stmt ∈ S do
6:      if stmt = branch then
7:          Get constraints C from predecessors of stmt
8:          merge all constraints C to T
9:      elseif stmt = function call
10:         sb ← get_subSlice(stmt)
11:         p ← get_entryPoint(sb)
12:         C ← Template(p, sb, T)
13: return C.
```

Our flow-sensitive constraint building process outputs a Z3-compliant formula as well as a regular expression that represents the request template that can be replayed by replacing concrete values for regular expression values that is readable by a human analyst for manual replay as well as automated replay.

*WAR*Droid converts the constraints for URI, request template, and response objects into regular expressions form for offline analysis. Variable types are inferred using analysis and heuristics similar to [19]. The regular expression format of a variable object is then derived using its type (e.g., [0-9]+ for integers). We additionally use heuristics from [18] to convert instances of repetitions and disjunctions into the Kleene star (*) and logical OR, respectively.

TABLE II
EXAMPLE HTTP TEMPLATE FORMAT

Method	GET | POST | UPDATE | PUT | DELETE
Scheme	HTTP | HTTPS
Domain	example.com
Path	/api/endpoint
Parameters	?id=x<,parameters>
Header	{HTTP Header}
Body	{content}

We model the HTTP request templates using the HTTP protocol fields that define the Method, Scheme, URI, Body, and Content parameters. Table II illustrates an example template. The constraints are encoded in the parameters, header, and body fields.

VI. WEB API HIJACKING OPPORTUNITIES

Uncovering Web API Hijacking opportunities is facilitated by the output of *WAR*Droid via the resulting HTTP templates. Web API hijacking opportunities for specific API endpoints are uncovered through evaluation of inconsistencies by generating requests from the request templates that violate one or more constraints expressed in the template. Since these are not confirmed attacks at this phase, we call them opportunities for exploit similar to [3]. These would only fall into the realm of actual exploitable vulnerabilities after they have been tested or shown to lead to actual violation of the security of the application or user data privacy.

To evaluate the inconsistencies, we employ a string matching approach to automatically test sample requests to determine inputs that could be successful. We further built heuristics into the test module to identify the server technology from the response headers. For example, some servers will disclose the runtime framework, database, and other details that can be used to fingerprint the server. In our prototype, we use simple heuristics to identify the web server runtime (PHP, asp.net, etc) and the backend server (MySQL, mssql). These are used to suggest further inputs that utilize domain knowledge, such as generating a simple SQL injection type input value.

A. Ethical Approach

We were very careful in our analyses to ensure that we would not cause any harm to the API servers or the mobile apps. The scope of our work did not require an IRB from our University, similar to related works such as [3], [22]. All testing was done in a responsible manner to ensure we did not cross any ethical boundary. We used test and demo accounts where possible, and we ensured that no private data was ever saved from any successful exploit. In one case study, we

worked with the app developer and obtained full permission to test their API.

B. Server Testing

To validate web API hijacking opportunities, we need to generate concrete values from the resulting HTTP templates recovered from the apps. At this point, we do not need the app or the Android framework as we can directly replay these requests using an HTTP library. For this purpose, we built a prototype python-based module. The request generation module takes the constraints expressions from the HTTP templates and utilizes the Z3-Str constraint solver to assist in generating concrete values.

1) Generating Input: Using the extracted path constraints encoded in the request templates, we identify possible invalid input parameter values by solving constraint negations. To this end, we use Z3-Str with the regular expression extension. We additionally take the approach of NoTamper [3] to iteratively solve the constraint disjuncts rather than solving a complete negation of the entire constraint.

2) Generating Requests: The request generation module involves two tasks: (1) constructing new logical constraint formulas whose solutions correspond to potentially invalid inputs and (2) solving those formulas to build requests from templates with concrete values.

Each invalid request sample would ideally test for a unique opportunity on the web server rather than repeating the same effective probe. To avoid redundant invalid requests, we convert the constraint formula to disjunctive normal form, and then we construct an invalid input for each disjunct while solving the rest of the formula to produce a valid input.

First, we generate concrete requests that satisfy the constraints. We generate two valid requests for each template and then replay these valid requests to the server and save the response data. Then, we compare both responses and remove all differences. This effectively removes the noise, such as date stamps, and useless server-generated values that may change across responses. The result is two response data traces that represents the similarity for responses to requests that are accepted by the server. We manually validate these to check that we are indeed comparing two responses to truly valid requests to the API. This will essentially serve as our ground truth to subsequently compare invalid requests.

3) Evaluating Responses: Lastly, we generate potentially invalid requests and collect the response for each one. For each response, we remove the elements that also occur in any of the saved valid responses for that template (sanitization). Then, we employ an edit distance algorithm to measure the distance between the sanitized responses for the invalid input and any of the responses from the valid input. Intuitively, if the two responses are similar to each other, we can infer that the invalid request was accepted by the server.

To determine if invalid inputs were accepted by the server, our approach compares the sanitized server response against a response that is known to have been generated by benign (valid) inputs. Since the server's responses are typically text-based JSON or XML or HTML, we can employ string similarity detection. In our case, since the responses are typically produced by a single web server, it is likely that the responses are similar, and therefore we implement a custom response comparison strategy. We evaluate the edit distance between the sanitized response (sanitized against a valid response) and another known valid response in a simple cross-validation approach. Our experiments and manual verification prove that this approach achieves decent accuracy in classifying server responses. We leave a more robust approach to future work.

VII. Evaluation

We evaluated the efficacy of *WAR*Droid on a set of 10,000 Android apps gathered from the Google Play store using the AndroZoo app crawler [23]. We identify several thousand apps that utilize web API functionality, many of which are flagged as potentially vulnerable to web API hijacking. We provide general details of specific case studies where *WAR*Droid identified and validated web API hijacking opportunities that we further manually validated. We refrain from disclosing app identities because some are either not fixed, in the process of being fixed after our notification, or in one instance we were asked not to make any public disclosure.

A. Test Apps

To test our framework, we evaluated a total of 10,000 apps chosen from the top 10 categories in the Google Play market. In total, *WAR*Droid took an average of 8 minutes to analyze each app and generated a total of 16,451 invalid requests samples for each template and twice the number of valid requests for response testing. This resulted in **4,562 apps flagged** as having a potential Web API Hijacking vulnerability. We tested and validated a smaller set of 1000 apps (using 1000 randomly chosen request samples from distinct apps across our dataset). Of those, 884 invalid requests were accepted by the API server, meaning that 884 of those flagged vulnerable apps were vulnerable, representing about 88.4% of the total tested invalid request templates in the sample set. Since we only tested a single generated invalid request for each app, it does not mean that the rest of the apps were not vulnerable. We further tested the remaining 116 apps using additional request samples and found that an additional 42 apps had an API that accepted an invalid request. In total, we verified that 926/1000 apps had at least one instance where it used a vulnerable web API.

Additionally, we found that 1,743 apps in our dataset generated unencrypted web API communication. While these do not strictly fall in line with our stated goal of uncovering validation inconsistencies, they nevertheless exacerbate the problem of vulnerable web API implementations. One app that has both a validation inconsistency and used an unencrypted channel is a gift card app that stores a monetary value that can be used to purchase goods from different online and offline stores. We worked with this particular developer to perform additional tests with their permission. We provide details of

some of these case studies below, but cannot disclose the full details for ethical reasons. Table III provides a summary of the distribution of apps and web API hijacking opportunities analyzed. Most vulnerable apps fall under the Tools category, but this turns out to be just a broad characterization of apps that perform diverse utilities. A flagged app is one for which *WAR*Droid detected a possible validation inconsistency. A verified app is one where we tested and verified the inconsistency using a generated request template. In all cases, we performed tedious inspection and ensured that no harm was done.

TABLE III
EVALUATION ON 10,000 APPS, AND TESTING ON 1,000 FLAGGED APPS.

Category	Apps	Flagged	Tested	Verified
Education	1000	201	46	42
Lifestyle	1000	398	15	12
Entertainment	1000	232	79	67
Business	1000	405	90	82
Personalization	1000	549	21	18
Tools	1000	734	303	291
Music	1000	434	22	17
Reference	1000	697	130	124
Travel	1000	224	86	85
Game	1000	688	208	188

False Positives: To further reduce false positives, *WAR*Droid applies some heuristics to remove responses flagged as vulnerable. We use a set of negative keyword instances such as 'Error' and 'Unauthorized' to filter responses that otherwise were very similar to successful responses. We also used a threshold response data size to filter responses where the data was too minimal to evaluate a meaningful edit distance. After applying these heuristics, we manually inspected random responses. There is an important distinction to make between false positives in the overall app, and the server validation routine. Here we are evaluating the false positives in individual server validation based on single requests. Overall, the app-level false positive is difficult to measure because even if a tested server request turns out to be a false positive, it does not guarantee that another server request for the same app will not be a true positive. For this reason, we merely flag apps as potentially vulnerable in the first instance.

Note that we do not evaluate false negatives because we do not guarantee complete code coverage, especially since we utilize program slices to reduce the search space and improve the usability of our tool. However, *WAR*Droid also generates reports for apps that include template definitions that can be further utilized by a human analyst to further test web API implementation through a manual process, especially where user authentication is required. This is noted in our limitations section. We argue, however, that our approach provides a lower bound on the total true positive web API hijacking opportunities that could be present for any given app/server combination.

Efficacy. We also evaluated *WAR*Droid against a manually generated list of web requests from an app. To accomplish this we chose a random app to test manually. We ran the app through *WAR*Droid and found that it generated a total of 8 web request templates. We then manually ran the app through a MITM proxy and captured the web request traces while a user performed typical app tasks for 2 minutes. We counted the total number of manual templates as the unique URI/path combinations from the request trace. We found only 6 such unique pairs, confirming that our analysis can perform better than manual testing. We leave a more extensive evaluation of the efficacy in this regard to future work. Our goal was to ensure that our prototype implementation had decent efficacy to gather reliable results.

B. Victim Population

To estimate the potential victim population of vulnerable applications, we checked the download statistics of each app flagged with a web API hijacking opportunity. Using the app package id's we checked the estimated download numbers for the application using a third-party service, AppBrain [24]. Using this information, we are able to get insights into the estimated potential victim population if web API hijacking opportunities can lead to actual exploits.

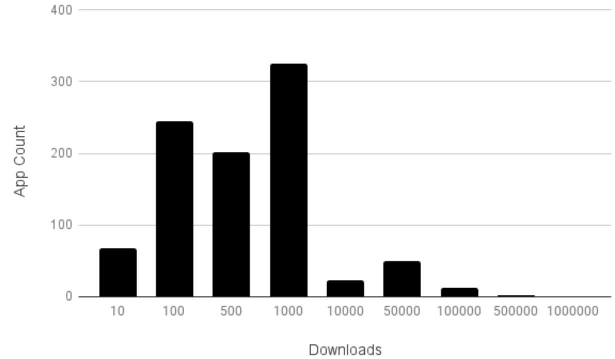

Fig. 2. Victim population distribution among verified apps with web API hijacking problems.

Figure 2 shows the download number distribution with most vulnerable applications having a user population between 100 to 1,000. Note this number is merely the lower bound of the real victim population, especially since these statistics do not consider other third-party marketplaces. This also suggests that the problem may be more prominent with less popular apps, which is an intuitive observation, although it also shows that popular apps are not excluded from this problem.

This represents a total estimated victim population of over 6.47 million users from only 926 apps that displayed web API hijacking opportunities. If we consider this to be a representative sample of the total number of apps, we can assert that the potential impact is widespread, reaching many millions of users throughout the world.

C. Impact Analysis

In this work, we focus on validation inconsistencies that enable a number of attacks to the mobile app server backend. Below are some of the specific attack case studies we

uncovered on apps that we tested. These are merely sample attacks of a wider array of possible attacks that are possible due to validation inconsistencies. We note that we also found apps that communicated over an unencrypted channel, which makes it easy for attackers to capture the required field values for a request template and replay the requests by leveraging validation inconsistencies as a means to an end. We refrain from identifying the apps and SDKs involved because some of these issues are still not fixed and we are in the process of properly notifying the app developers. The variation and potential severity and reach of these attacks illustrate the importance of this problem. We stress here that we were careful in evaluating these case studies in a safe manner without causing harm. In most cases, we used our own dummy accounts.

Unauthorized data access. Many apps we analyzed included basic to non-existent authentication and authorization mechanisms to control access to their backend services. Most apps include an authentication token (key) with each request that identifies the app to the backend and authorizes access to data and services on the backend. While backend services may provide additional layers of security, we found that many apps choose to bypass these additional authentication steps.

As an example of unauthorized access, we discovered an app that simply sent the user's email address as an authentication and authorization token. This app had over 5,000 downloads at the time of our testing. We setup test accounts with the app owner permission and discovered that the server did not perform any authorization checks. *WAR*Droid identified the email address parameter constraints as imposed by the app and suggested an invalid email parameter as a test case. After coordination with the app developer team, we were given permission to test a non-production web API server that was an exact copy of their production server, but with fake test data. It turns out that the app team consisted of a small number of inexperienced developers, which is not uncommon in the mobile space. Informed by the web request template constraints, we were able to launch a SQL injection attack on the test server and retrieved a full list of all test app user data. This would allow us to access any user account on the app

The root cause of this was the inconsistent validation of the email string format at the server side. Since this was a virtual money transfer app used in actual online and offline stores, our discovery had serious potential consequences. Upon further testing, we verified that the web API allowed us to freely transfer funds between two user accounts. Since working with this app team, they have fixed the validation inconsistency issue, but they asked us to remain anonymous for fear of bad publicity. This is an extreme case, but we think it is indicative of many apps on the market, especially those deployed by less experienced team.

JSON-based SQL Injection. On yet another app, we uncovered a different SQL-injection vulnerability facilitated by inconsistent data validation in a login form that allows us to login as any user to an app. This is a less popular app that had only over 1 thousand downloads at the time of testing. This app sends the username and password as a JSON array data type in the form {username: $usr, password: $pwd}. *WAR*Droid further reports that the password field is constrained by the app to only use alphanumeric values. While *WAR*Droid does not suggest a proper invalid input, we utilize domain knowledge to test this potential inconsistency. We found that the server does not implement a similar constraint on the password and happily accepts any input as long as the JSON data is properly formatted. Subsequently, we are able to login by replacing the password parameter with the following value: ”,”$or:[{},{’1’:’1’}]. We note here that we used our own sample dummy accounts and notified the app developers of the potential problems, which has since been fixed.

Shopping for Free. We discovered a problem with a popular ecommerce SDK utilized by thousands of apps and online stores across the world, with millions of users. *WAR*Droid reported a template where the constraint on the quantity field for shopping cart items disallows numbers less than 1. Naturally, a quantity zero would have no effect, but *WAR*Droid also suggested a violating input as a negative quantity. This is disallowed by the sdk's constraints in the app, but we discovered that it was allowed by the server because the same functionality is used to process returns and refunds, where a negative quantity is indeed valid. However, since this inconsistency exists, we can bypass the app and replay a checkout action using a negative quantity on a line item that can be manipulated to cause the checkout total to be zero dollars. We tested this on a demo store account that we created and confirmed the problem with the app developer. We note that this problem has been fixed in a new release of their SDK, although the old version still exists in production apps.

Cross Platform Content Injection. On a news app with over 500,000 downloads, we discovered a problem where the mobile app allows a user to enter comments on a news article that is not properly sanitized at the server for proper formatting. We discovered that the accompanying website for the news station also displays comments entered on the mobile app, and the mobile app disallows HTML characters in the comments. *WAR*Droid suggested that HTML characters could be accepted by the server, which would be inconsistent with the app constraints. Indeed, we were able to replay a comment posting request with HTML characters, and the server stored the values as is. This is not a problem when displaying the comment on the mobile app, as it does not render HTML. However, since the company's website uses the same data store, and the API design requires only client apps to validate content, then the website renders the incoming comments as HTML. This is a serious problem that could cause all kinds of havoc on the website, including cross-site scripting attacks.

Account DoS. On a particular health app used by millions of users around the world, *WAR*Droid reported a constraint on the password change request that restricted the password length to 10 characters in addition to typical password constraints. This is a popular fitness app that had over 10 million downloads at the time of testing. The server did not apply the same validation as the app and allowed us to update a password

to a longer string. This caused the account to get locked out of the app. While this attack may have no effect and may not be useful, since an attacker wouldn't find much use in locking himself out of his own account, it does illustrate the pervasive nature of the types of simple inconsistencies between app input validation logic and server API validation logic.

Transferring Money. *WAR*Droid analyzed an app by a major US bank and reported a potential inconsistency in the money transfer functionality. The app restricts transfers only to connected accounts displayed in a spinner UI element. The author used two of his own disconnected accounts to test this inconsistency opportunity and was able to successfully transfer funds between his two accounts although it was not possible directly through the app or through the bank's website. Again, this may not be of particular interest to an attacker because he may not want to transfer money out of his own account to an unknown account. However, this also shows that the inconsistency problem exists in some of the most important and critical apps used in society. This bank app that had over 10 million downloads at the time of testing. There may be a wider array of inconsistencies that could potentially be exploited, but due to ethical reasons, we are unable to test or validate other potential inconsistencies except where we can use our own account and not cause any harm. As of this writing, this problem no longer exists in the updated bank server's API.

VIII. Discussion

Mobile applications are a necessity in many facets of society these days. In addition to traditional service businesses offering mobile applications, such as banks, and applications already available on the web, the proliferation of Internet of Things means that many more devices have Internet connectivity and can be controlled from a mobile phone. Examples are home and office security systems, cars, classroom audio video equipment, home appliances (thermostats, refrigerators, televisions). It becomes very critical that the Web API endpoints of these devices are properly secured from hijacking vulnerabilities.

A. Defense Guidelines

We attribute some of the observed problems to the shifting app architecture in the modern era where web APIs are generic service that can scale to support multiple client platforms, including web and mobile apps. Additionally, due to the enhanced capabilities of mobile devices, web service providers sometimes opt to defer validation logic to the clients, ignoring or oblivious to the subtle inconsistencies and vulnerabilities that may arise as a result. Following are some guidelines based on our findings in this work.

- Never trust the client. Do not defer validation to the client side. The server must be at least as strict as the client for input validation.
- The server must be prepared to handle and reject input regardless of the client. No assumptions must be made about the client.
- Authentication and Authorization logic must be carefully implemented at the server side.
- Client-side validation must be thoroughly tested for consistency with server-side validation logic. *WAR*Droid can help in identifying potential inconsistencies.
- Clients and Servers must sanitize inbound and outbound data, especially where it can be used on either a mobile or web client interchangeably.

While we have focused on the problems that can arise due to inconsistent input validation logic, we believe that it will take a concerted effort and paradigm shift to address mitigation of this problem.

B. Limitations

Obfuscated code: Obfuscation is commonly observed in popular real-world apps. A recent study has shown that 15% of apps are obfuscated [25]. We find that many real-world apps do not obfuscate their code. Many tools, including Proguard [26], rename identifiers with semantically obscure names to make reverse engineering more difficult. *WAR*Droid does not handle obfuscated application code, but it is included in future work.

*WAR*Droid also does not handle native code and JNI code. We consider these to be out of our scope.

State Changes: Another limitation of *WAR*Droid is that it cannot reason about state changes and values that may originate from a previous request to the API. For example, the app may request a token value from a remote server that could be included in a subsequent request. Previous works such as [21], [18] propose methodologies that can accomplish this task. *WAR*Droid can be retrofitted with this feature to improve its accuracy.

WebViews: *WAR*Droid's analysis is focused on native mobile code, and does not consider web API accesses facilitated through WebView-loaded JavaScript code in hybrid mobile apps. We use a subset of the apps from our recent work which identifies that over 90% of apps included at least one WebView [19]. In that work, we provide an approach for uncovering JavaScript Bridge functionality and semantics in hybrid mobile apps.

Authentication: *WAR*Droid also cannot evaluate requests that require user authentication unless we hard-code test credentials into the request template, such as a valid oAuth tokens. An inherent challenge with most static analysis-based systems, including *WAR*Droid, is the inability to automatically synthesize valid authentication sessions. Some level of human intervention is necessary to overcome this limitation.

C. Convergence of Web and Mobile

In today's Internet-connected mobile society, the web and mobile platforms share some common ground in the effort to provide security and privacy. Indeed, this work is inspired by previous works on the web platform such as NoTamper [3] and Waptec [7] that pursue similar goals in the context of browser-based web applications. In this work, we directly tackle an

important issue that emerges from the amalgamation of the web and mobile platforms.

The combination of mobile and web into new complex systems such as web service APIs, web-based operating system environments, and hybrid applications presents a new frontier in security and privacy research.

IX. Related Work

We build on a number of previous works in the area of program analysis on the Android framework. We especially make use of Flowdroid [15] and Soot [11] program analysis tools. Prior applications of these tools on Android include detection of privacy leakage, malware detection, and other vulnerability detection. In this work, we utilize program analysis techniques to analyze a mobile application's validation logic as a model of it's backend server validation logic.

Web Application Analysis. Our work is inspired by previous research into parameter tampering vulnerabilities on web applications. Attacks that exploit these vulnerabilities leverage the loose coupling of web services between the client and server side. Waptec [7] and NoTamper [3] are two prominent works that automatically identify parameter tampering vulnerabilities in web applications and generate exploits for those vulnerabilities. Similarly, *WAR*Droid uses concepts inspired by these works to analyze the inconsistencies of the loose coupling between mobile apps and their backend web API servers.

SIFON [27] analyzes web APIs to determine the extent of oversharing of user information where the server sends information to the app that is never used. Other related works look at the issues that arise when webview components are used to combine the web and mobile platforms into a seamless experience. Luo et al. found several security issues that arose due to this practice [28]. NoFrak [29] analyzed a similar issue and proposed an approach to augment the security models to allow finer grained access control between mobile and web interaction.

Static Analysis. This work utilizes various static analysis techniques and tools. Static analysis is often scalable since it does not have to execute the app, and can achieve higher code coverage than dynamic analysis. Previous works that use static analysis commonly reconstruct the inter-procedural control flow graph by modeling the Android app's life-cycle. In this work, we leverage FlowDroid [15] to similarly reconstruct and extend the ICGF as an augmented program dependence graph, but our goal is slightly different than detecting data flow from source to sink. Other similar works such as Extractocol [18] and Smartgen [30] follow a similar approach and utilize Flowdroid as the basis for static analysis of apps to uncover the behavior of communications with web servers. *WAR*Droid similarly analyzes the network behavior, but with a different goal of analyzing the validation inconsistency with the server.

Protocol Reverse Engineering. Our work shares some similarities and goals with protocol reverse engineering [31], [32]. However, rather than exhaustive protocol reconstruction, our goal is more aligned with [33] with a focus on uncovering particular server-side vulnerabilities.

Input Generation. Several previous works implement input data generation or fuzzing on Android applications. Intellidroid [34] is a hybrid dynamic-static analysis framework that analyzed event chains and can precisely identify the order or inputs to trigger a specific code path. We used several concepts from Intellidroid, especially as it relates to symbolic execution and solving constraints using Z3 libraries. We opted not to directly use Intellidroid in our approach because it is more suited to malware detection and requires Android framework instrumentation and execution in an emulator.

Symbolic Execution. Symbolic execution has been widely used in many security applications on mobile applications. TriggerScope [35] uses symbolic execution and other program analysis techniques to precisely identify logic bomb triggers in Android apps. IntelliDroid is similar to our work and extracts path constraints that are used to generate app inputs that can trigger specific execution paths. We leverage many of their techniques and motivation in implementing symbolic execution to extract path constraints.

App Network Traffic. Several previous works also analyze app network traffic, but not necessarily through analysis of the apps. Instead, this area of research primarily focuses on the network layer to fingerprint apps through raw packet-level network traffic inspection. FLOWR [36] tries to distinguish mobile app traffic by extracting key-value pairs from HTTP sessions at the network level. NetworkProfiler [37] uses UI-based fuzzing on Android apps to build a comprehensive network trace for a given app.

X. Conclusion

Modern mobile applications rely on web services to enable their functionality through HTTP-based communication. Unfortunately, the disparate nature of the mobile and web platforms causes input validation inconsistencies that can lead to serious security issues. We presented *WAR*Droid, a framework that utilizes static program analysis and symbolic execution to model input validation logic between mobile apps and their remote web API servers. *WAR*Droid extracts and validates web API logic implementation in mobile apps and uncovers inconsistencies between the app and server logic.

The uncovered inconsistencies are shown to expose serious vulnerabilities in web API servers that affect a diverse set of mobile apps. Our analysis of 10,000 apps uncovered a significant portion of apps with web API hijacking opportunities that can violate user privacy and security for millions of mobile app users. The inconsistency problem is not limited to Android apps, but any client that utilizes the deployed web API services, including iOS apps, Windows apps, and web applications. This work sheds light on the existence and pervasiveness of this important ongoing research problem, and our hope is that it will motivate further research in this area.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant no. 1314823

and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," Tech. Rep., 1999.

[2] "OWASP Mobile Threats," https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[3] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "Notamper: automatic blackbox detection of parameter tampering opportunities in web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 607–618.

[4] 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*. ACM, 2010, pp. 921–930.

[5] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 463–478.

[6] A. Mendoza, K. Singh, and G. Gu, "What is wrecking your data plan? a measurement study of mobile web overhead," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 2015, pp. 2740–2748.

[7] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 575–586.

[8] A. Sudhodanan, A. Armando, R. Carbone, L. Compagna *et al.*, "Attack patterns for black-box security testing of multi-party web applications." in *NDSS*, 2016.

[9] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 75–88.

[10] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[11] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[13] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[14] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[16] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[17] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 114–124.

[18] H. Choi, J. Kim, H. Hong, Y. Kim, J. Lee, and D. Han, "Extractocol: Autoatic extraction of application-level protocol behaviors for android applications," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 593–594, 2015.

[19] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and scalably vetting javascript bridge in android hybrid apps," in *Proceedings of The 20th International Symposium on Research on Attacks, Intrusions and Defenses (RAID'17)*, September 2017.

[20] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *NDSS*, 2015.

[21] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," 2016.

[22] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online–security analysis of cashier-as-a-service based web stores," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 465–480.

[23] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 468–471.

[24] "Appbrain android statistics," https://www.appbrain.com//.

[25] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 221–233.

[26] E. Lafortune *et al.*, "Proguard," *h ttp://proguard. sourceforge. net*, 2004.

[27] W. Koch, A. Chaabane, M. Egele, W. Robertson, and E. Kirda, "Semi-automated discovery of server-based information oversharing vulnerabilities in android applications," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 147–157.

[28] 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*. ACM, 2011, pp. 343–352.

[29] S. Pooryousef and M. Amini, "Fine-grained access control for hybrid mobile applications in android using restricted paths," in *Information Security and Cryptology (ISCISC), 2016 13th International Iranian Society of Cryptology Conference on*. IEEE, 2016, pp. 85–90.

[30] C. Zuo and Z. Lin, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 867–876.

[31] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 621–634.

[32] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 110–125.

[33] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications." in *NDSS*, 2014.

[34] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[35] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 377–396.

[36] Q. Xu, T. Andrews, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, and A. Nucci, "Flowr: a self-learning system for classifying mobileapplication traffic," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 569–570, 2014.

[37] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "Networkprofiler: Towards automatic fingerprinting of android apps," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 809–817.