Web APIs in Android through the Lens of Security

Pascal Gadient, Mohammad Ghafari, Marc-Andrea Tarnutzer, Oscar Nierstrasz

Software Composition Group, University of Bern
Bern, Switzerland
scg.unibe.ch/staff

***Abstract*—Web communication has become an indispensable characteristic of mobile apps. However, it is not clear what data the apps transmit, to whom, and what consequences such transmissions have.**

We analyzed the web communications found in mobile apps from the perspective of security. We first manually studied 160 Android apps to identify the commonly-used communication libraries, and to understand how they are used in these apps. We then developed a tool to statically identify web API URLs used in the apps, and restore the JSON data schemas including the type and value of each parameter.

We extracted 9 714 distinct web API URLs that were used in 3 376 apps. We found that developers often use the `java.net` package for network communication, however, third-party libraries like `OkHttp` are also used in many apps. We discovered that insecure HTTP connections are seven times more prevalent in closed-source than in open-source apps, and that embedded SQL and JavaScript code is used in web communication in more than 500 different apps. This finding is devastating; it leaves billions of users and API service providers vulnerable to attack.

***Index Terms*—Web APIs, network libraries, communication, security**

I. Introduction

Mobile applications (apps) increasingly rely on web communication to provide their services. Apps access the internet through web APIs in order to use an increasing number of public web services, or to communicate with private backends. Researchers have recently studied the use of such APIs in mobile apps, and, for instance, found that a large number of web requests are not directly traceable to source code [1], cloud and mail service credentials are hard-coded in the apps [2], many web requests are harmful [3], many web links targeting well-known advertisement networks impose serious risks on users [4], and lax input validation in many web APIs could compromise the security and privacy of millions of users [5].

We could not, however, find any publicly available tool that researchers can use to study web APIs. Also, There are several third-party libraries to implement network communication, but existing studies are mainly limited to `java.net` APIs. Finally, dissecting the distribution of elements that comprise the web API URLs is never studied, which is necessary for collecting security-related information stored in query keys and values, as well as to fuzz web APIs.

We manually studied the use of common web communication frameworks in 160 randomly selected Android mobile apps, *i.e.*, more than 4.7% of the whole dataset, and developed a static analysis tool to investigate whether network communications in 3 376 closed-source and open-source apps differ. We manually inspected the tool's output for 100 random apps, and used the reported URLs to connect to the servers and to investigate their response. We found eight security code smells, *i.e.*, *symptoms in the code that signal the prospect of a security vulnerability* [6], on both ends, dominated by the use of embedded computer languages. We handcrafted regular expressions to automatically identify the use of those languages, and other languages prevalent on GitHub.

In this work we address the following research questions:

RQ$_1$: *Which API frameworks are used in Android mobile apps, and what is the nature of the data that apps transmit through these frameworks?* We identified six different web API communication libraries, and learned that open-source apps rely on simpler request paths including only one or two path segments, while closed-source apps mostly include two or three path segments. Unexpectedly, the opposite is true for key-value pairs: Open-source apps frequently use one to three pairs, while closed-source apps mainly use one pair. Fragments have only been used very sparsely in both types of apps. We found that open-source and closed-source apps are similar in the choice of web communication libraries, but advertising services are more prevalent in closed-source apps.

RQ$_2$: *What security smells are present in web communication?* We found eight security smells in the apps and the server software. For instance, 500 apps use embedded computer languages (*e.g.*, SQL, and JavaScript commands) in web API communications, thus introducing the threat of code injection attacks. A horrific 67% of the closed-source and 9.5% of the open-source apps communicate with servers over insecure HTTP connections. Many apps neglect to use the HTTP strict transport security policy. Finally, we observed a lack of authentication and authorization mechanisms for services that are supposed to be private.

In summary, this work attempts to shed more light on the use of web APIs in mobile apps, by studying what data the apps transmit, to whom, and for what purpose. The tool and the obtained results in this study are available online.[1]

The remainder of this paper is organized as follows. We describe the methodology of our web API mining approach in section II, and we present the results of our empirical

[1] https://github.com/pgadient/jandrolyzer

study in section III. We report numerous web API security smells in section IV. Finally, we recap the threats to validity in section V, and we summarize related work in section VI. We conclude this paper in section VII.

II. Web API Mining

We manually inspected Android apps to identify what APIs developers use to call web services, and how they are used. Then we took advantage of this information to develop a tool to automatically extract the web API URLs and their corresponding HTTP request headers statically from the apps.

A. Library Inspection

An Android app can call a web API either with the help of the built-in Java classes, or by using external third-party libraries. We consulted the official Java and Android documentations to compile a list of built-in APIs that are relevant to network communication, and to establish how these APIs are used. We mainly focused on the `java.net` package, which includes a number of classes such as `Socket`, `HttpsURLConnection`, and `URLConnection` to implement network-related operations

Next, we manually inspected 160 randomly selected apps from a dataset of 3 376 apps (see section III) that request Android's `INTERNET` permission to investigate what third-party libraries they may use for web communication, and how. These libraries are often built on top of the built-in Java network APIs. Therefore, we first checked whether a call to such Java APIs exists, and, if so, we checked whether the call belongs to the app or an external library. For each library, we studied the documentation, and investigated how developers use the library in each app, *e.g.*, to construct URLs, and to attach headers to web requests. During the inspection of each app, we collected the web API URLs and any data that are transmitted to the servers to determine if what we collect from the source code is actually helpful to issue valid requests.

In this study, besides the native Java network libraries, we found that libraries such as *Apache HttpClient*, *Glide*, *Ion*, *OkHttp*, *Retrofit*, and *Volley* are used in the apps.

While studying the use of web communication libraries, we also noticed that besides the built-in *org.json* package, developers often use two external libraries, namely *Gson* and *Moshi*, for parsing and manipulating JSON (JavaScript Object notation) data, which is commonly used for data exchange in web services.

B. API Miner

We then developed a tool that leverages our finding in the library inspection phase, and statically analyzes apps to extract web API URLs, query keys and the corresponding values where applicable. The tool takes the following steps:

1) Decompilation: Given an APK file, the tool first decompiles the app using the command line version of the *JADX* decompilation tool.[2] A successful decompilation will provide us with a project folder that contains decompiled Java source code of the app and the resource files. Although decompilation errors are common, JADX is quite robust and produces code with a correct syntax. In particular, method declarations and class structures remain intact with comments in place where the decompilation did not succeed completely.

The tool uses the *JavaParser* framework to create an abstract syntax tree (AST) for every `.java` file within the project.[3] When the actual source code of an app is available, we use the information from the build and configuration files to accurately inject specific library versions into the *JavaParser* framework to enable the resolution of library dependencies in the subsequent app analysis. If the desired library version is unavailable in our collection, the next available more recent version is added instead. Closed-source apps (*i.e.*, APKs) do not require those dependency injections as they already contain the required code themselves.

2) Detection and Extraction: In principal, we need to track flows of data in relevant APIs, and several static analysis frameworks exist to track data flows in Android apps. Nevertheless, in our experience as well as according to recent studies, these tools may not perform as described in the relevant papers [7], [8], [9]. We therefore decided to implement our own lightweight analysis tailored to reconstruct web APIs in the code.

The tool traverses the AST to identify APIs, *i.e.*, `MethodCallExpression` nodes, that are used to access web APIs in a network library. For each method call, it recursively resolves the nodes on which the API depends, *e.g.*, the object on which the method is called, and its parameters. In detail, we rely on the *JavaSymbolSolver* framework to associate a variable in the code to its declaration.[4] We track all `Assignment`, and `MethodInvocation` constructs on each variable in each relevant `VariableDeclaration` node. Moreover, depending on the target library, the tool also tracks implicit dependencies, *e.g.*, the annotation-driven dependency injection.

URL and header[5] construction largely depend on string concatenation. We therefore support the extraction of strings that are built using the `StringBuilder.append()` method, the `String.concat()` method, and the "+" operator.

3) Reconstruction: All web API URLs and JSON data structures that contain at least one unresolved value are further processed in the reconstruction stage. We set the value of variables whose types are number or boolean to `0` and `true`, respectively. For those variables (*i.e.*, JSON or query keys) whose types are `String`, and for which we did not find a concrete value during the extraction, we compute the *Jaro-Winkler* similarity distance [10] between the variable names and every variable declaration in the code. In the end, for each successful analysis, the tool reports the web API, as shown in Listing 1, and the corresponding request headers, as shown in Listing 2.

[2]https://github.com/skylot/jadx

[3]https://javaparser.org

[4]https://github.com/javaparser/javasymbolsolver

[5]The HTTP request header is a plain text record providing input details for the web API request.

```
Path:
/Users/marc/...
Library:
com.squareup.retrofit
Scheme:
http://
Authority:
retrofiturl.com
Base URL:
http://retrofiturl.com
Endpoints:
    Path: api/loadUsers
    Queries:
        Query key: position, query value: <String>
        Query key: order, query value: <String>
    Fragments:
    HTTP Methods:
        HTTP Method: GET
```

Listing 1. The tool's output for a successful web API extraction

```
Path:
.../User.java
Library:
com.squareup.moshi
JSON Object:
{"address":{"street":"<STRING>",
   "number": <NUMBER_INT>"},"name":"Bob"}
```

Listing 2. The tool's output for a successful JSON object extraction

4) Evaluation: We performed a lightweight evaluation of the tool on 10 open-source and 10 closed-source apps randomly selected from our dataset. In each app, we manually searched for the terms "http://" and "https://" in the (decompiled) source code. For each finding, we evaluated which entries were related to web APIs, and then tried to understand what are the URLs and the other request parameters.

We manually identified 24 distinct URLs for web APIs in the apps, of which 21 were found in the Java source code. The tool reported 39, of which 18 URLs referred to web services: 17 were amongst the URLs identified manually, and the tool uncovered one new case that was overlooked due to complex string concatenation. The tool achieved a precision of 46% and a recall of 80%.

There are several reasons for the tool missing the remaining seven URLs, such as URLs in open-source apps being hidden in build scripts and XML resource files rather than Java code, and incomplete library injections for closed-source apps.

The tool reported 21 URLs that did not refer to a web service. In particular, 18 URLs referred to static HTML pages, and three suffered from invalid reconstruction.

C. Security Checks

We inspected the result of the tool on a random set of 100 apps in order to identify security smells in the code relevant to web API communications.

We implemented lightweight detection strategies for these smells, mainly using regular expressions. For instance, using search terms such as username, password, *etc.* we could find hard-coded passwords, tokens, and insufficiently protected authorization schemes in the results.

```
HTML:
String uiElement = "<html><body>" +
    ↪ jsonObj.getText() + "</body></html>";

JavaScript:
String customScript = jsonObj.getResponse();

SQL:
String queryParameter = "SELECT * FROM weather";
```

Listing 3. Examples of embedded computer code in app source strings

In many apps we found code from various computer languages embedded in Java strings, such as that shown in Listing 3, thus potentially exposing the app or the server to code injection attacks. We compiled a list of commonly used computer languages based on our own findings, and the scripting languages found in the top ten used programming languages on GitHub.[6] For each language, we pragmatically developed regular expressions inspired by the relevant language specifications, with the aim to match as many occurences as possible. With these regexes, shown in Table I, we counted the key identifiers for each language in each app report, to detect usages of embedded languages in the web communications.

TABLE I
REGULAR EXPRESSIONS USED TO DETECT COMPUTER LANGUAGES

Language	Regular expressions	Language	Regular expressions
Bash	sh[ ]+	SQL	alter[ ]+table
	%.sh		create[ ]+.*index
HTML	%<[ ]*html[ ]*%>		create[ ]+.*table
JavaScript	function[^%(]*%([^%)]*%)		create[ ]+.*trigger
	%<[ ]*script		create[ ]+.*view
	js[ ]*=		delete[ ]+from
PHP	%<%?		drop[ ]+index
Python	import[ ]+%(.*%)		drop[ ]+table
Ruby	require[ ]*%(.*%)		drop[ ]+trigger
			drop[ ]+view
			insert[ ]+.*into
			replace[ ]+into
			select[ ]+.*[ ]+from
			update[ ]+.+[ ]+set

In a subsequent step, we issued requests to each of the URLs extracted from the entire dataset, and observed unexpected responses, *e.g.*, stack traces, error messages, or status information, disclosing sensitive information regarding the API implementation, running software, or server configuration.

III. STUDY RESULT

We investigated the use of network communication in Android mobile apps. In particular, the focus is on the use of libraries, and the request characteristics.

We randomly collected apps that use internet. For closed-source apps we mined the free apps on the *Google Play* store, and for the open-source apps we relied on the *F-Droid* software repository.[7] For each app, we removed the duplicates, *i.e.*, apps with the same package identifier, but different version

[6]https://github.com/oprogramador/github-languages

[7]https://f-droid.org

numbers, and kept only the most recent version of the app. In the end, we collected 17 079 closed-source, and 432 open-source apps.

We applied our tool to these apps, and restricted each app analysis to 30 minutes processing time, with a node resolution limit of 15 iterations on a machine with two AMD Opteron 6 272 16-core processors and 128 GB of ECC memory. The tool could completely analyze 293 open-source apps, and 2 410 closed-source apps. We also included the partial results of the apps whose analyses were incomplete, resulting in a total analysis result of 303 open-source, and 3 073 closed-source apps in our dataset. Only 2 587 apps (15%) were successfully decompiled, due to crashes of the tool caused by various bugs, and incomplete feature support, *e.g.*, reflection, native code, and customized app configurations.

The apps in our dataset come from 48 different Google Play store categories. Most of them belong to `EDUCATION` (317 apps) and `TOOLS` (292 apps), however, a majority (574) have a `GAMES`-related tag. Interestingly, work-related apps are common in our dataset (335 apps). The top five categories whose apps contain the largest number of distinct web API URLs are `EDUCATION` (1 555 URLs), `LIFESTYLE` (1 027 URLs), `BUSINESS` (995 URLs), `ENTERTAINMENT` (704 URLs), and `PRODUCTIVITY` (619 URLs).

We present our findings in the following, and conclude each focal point with a short discussion, which entails similarities or differences in open-source and closed-source apps.

A. Communication Libraries

We investigated the distribution of the seven different communication libraries in 3 376 apps in our dataset.

1) Result: In *open-source apps*, we found that each app uses up to four network libraries. The `URLConnection` (37%), `HttpURLConnection` (24%), `Socket` (9.1%), and `HttpsURLConnection` (6.0%) classes included in *java.net* are the preferred choice of open-source developers, especially `URLConnection` and `HttpURLConnection` are omnipresent in projects. When considering third party network libraries, we found that *OkHttp* and *Retrofit* (each 5.6%) are used the most. It is interesting to see that libraries with specific support for image downloads are similarly used, *i.e.*, *Glide* and *Volley*. The *Ion* library is used only in three apps (1.0%).

In *closed-source apps* each app uses up to seven network libraries. We found that the classes included in *java.net* such as `URLConnection` (42%), `HttpURLConnection` (34%), `Socket` (10%), and `HttpsURLConnection` (4.3%) are the preferred choice. Interestingly, the *OkHttp* library is the most commonly used third-party library even surpassing the well-known *Glide* and *Retrofit* libraries. We found `org.apache.httpcomponents` and `com.loopj.android` are the two least used network libraries contributing only 0.9% and 0.5%, respectively.

2) Discussion: We realized that one to three classes are usually responsible for network communication in an app. In open-source apps we found the use of up to four network libraries in each app, and in closed-source apps it was up to seven. Although each library may provide specific features, *e.g.*, JSON parsing, HTTP connection management, image caching, *etc.*, we expect the reason for the use of multiple libraries in an app is that many developers use the code snippets from other projects or online information sources.

We found fewer *java.net* libraries in open-source apps compared to closed-source apps. During decompilation, the bundled libraries are decompiled together with the app code. Therefore, what the tool reports is not only the network calls in the app code, but also the network APIs on top of which the third-party libraries are developed. However, this is not the case for the open-source apps whose dependencies are defined in Gradle, and are dynamically injected without adding the actual code to the project itself.

The libraries *Ion* and *Volley* have been used only in open-source apps, while *HttpComponents* and *LoopJ* have been used only in closed-source apps. Surprisingly, we did not find any instances of the well-known `AndroidHttpClient` and `SSLSocket` classes. Finally, the use of *Glide*, which supports exhaustive image downloading and caching features, seems much more prevalent on closed-source apps.

B. The Nature of Web API Requests

Based on the analysis results for the apps in our database, we investigated the structure, dissemination and use of 13 276 web API URLs, of which 9 714 were unique.

1) Open-source Apps: The tool extracted 1 533 URLs from the open-source projects. We found that the majority of web APIs consist of one or two queries or path segments. We only found up to one fragment per web API. We further found that 209 web APIs exist with paths consisting of four or five segments to distinguish between resources (the average number of segments in the web APIs is 2.36). Nevertheless, web APIs using more than five elements are rare. Web APIs contain an average of 2.3 key-value pairs in queries. The data do not follow a normal distribution.

Surprisingly, the top base URL was https://github.com, which we observed 29 times (1.8%). Likewise, *Google* services have been widely used, *e.g.*, https://play.google.com or https://plus.google.com, of which the tool could spot 42 instances (2.7%). Rather at the end of the ten most commonly used base URLs the tool found the *OpenWeatherMap* API http://openweathermap.org (7, 0.4%) and the *Twitter* social network API https://twitter.com (6, 0.3%).

Furthermore, we found that the `https` URL scheme (1 012 occurrences, 66%) is much more commonly used than its insecure counterpart `http` (521 occurrences, 33%).

2) Closed-source Apps: The tool extracted 11 743 URLs from closed-source apps. We found that the majority of web APIs consist of one or two queries or path segments. On a second look, we observed that web APIs with two path segments are most prevalent. We further discovered that 2 116 web APIs exist with paths consisting of four to eight path segments to distinguish between resources (the average number of segments in the web APIs is 2.44). Nevertheless, web APIs using more than four elements are rare. Additionally,

we could identify that URL fragments are seldomly used in web APIs; although we found up to seven fragments in a single web API URL, we only discovered 183 web APIs in total using this feature, *i.e.*, 1.5%. Web APIs, on average, contain 2.9 key-value pairs in queries. The data do not follow a normal distribution.

Interestingly, all the most common URLs we could retrieve were pointing towards *Google* services. The top URL, http://schemas.android.com, was observed 1 303 times (11%). Two of the observed URLs were related to advertising distribution services, *i.e.*, http://media.admob.com (283, 2.4%) and https://pagead2.googlesyndication.com (271, 2.3%).[8]

We found that the `http` URL scheme (7 208 occurrences, 61%) is much more prevalent than its secure counterpart `https` (4 531 occurrences, 38%). Besides findings of the two common schemes we found few appearances of the `ws` (WebSocket) protocol (4 occurrences, 0.0%), which provides (unprotected) full-duplex communication on top of HTTP TCP connections.

3) Discussion: The number of used path segments and query keys are an indicator for the complexity of a specific request. Servers usually reject requests with incomplete or flawed parameter configurations, and thus the task of sending a successful request becomes harder the more path segments and query keys are involved.

Open-source apps relied on simpler request paths including only one or two path segments, while closed-source apps mostly included two or three path segments. Unexpectedly, the opposite is true for key-value pairs: Open-source apps frequently use one to three pairs, while closed-source apps majorly use one pair. Fragments have only been used very sparsely in both types of apps.

We did not expect to observe a difference between open-source and closed-source apps. Moreover, we did not expect to find many complex requests, because the idea of providing APIs is that they can be used by other developers who presumably prefer an easy to use interface. We conclude that the majority of the APIs provide a simple interface and are rather straightforward to access.

While the open-source apps contained no advertising services in the ten most used base URLs, the closed-source apps heavily used such services. We expect that the "Freemium" price model, *i.e.*, installation of apps is free but the user must later watch ads or pay a fee, is a major enabler of this setting.

The open-source community prefers the *Twitter* social network over *Facebook*.

We found one major difference in the URL schemes used in the apps. Open-source apps principally rely on secure `https` connections (66%). In contrast, closed-source apps largely use the insecure `http` protocol (60%). We see here much potential for improvement through stricter rejection of apps using insecure connections. The more efficient, but more complex WebSocket protocol seems to be out of interest for the majority of developers.

[8] *Google AdMob* is a popular advertising platform that provides SDKs to developers to integrate *Google* ads into their own apps to increase revenue.

C. Security Risks

We studied the kinds of data communicated through web APIs, and found that both credentials (*i.e.*, user name and password combinations) and embedded code were very common in the web communications. As the former has been reported on extensively in the past, we focus here on the latter.

1) Open-source Apps: The tool extracted 458 JSON schemes in which `STRING` is the most used value type with 1 197 occurrences, followed by `NUMBER` with 234 occurrences.

We found that SQL (91%, 10 affected apps) is by far the most used embedded language. HTML (5.5%, 2 affected apps) and JavaScript (2.7%, 1 affected apps) are very rare. No instances of other embedded languages were detected.

2) Closed-source Apps: The tool extracted 14 606 JSON schemes where `STRING` is the most used value type with 40 017 occurrences, followed by `BOOLEAN` with 5 640 occurrences. `NUMBER` and `NULL` only represent a minority with 2 389 and 1 483 occurrences, respectively.

In contrast to open-source apps, we observed that JavaScript (76%, 170 affected apps) is very prevalent, and SQL (23%, 476 affected apps) is used less, but still frequently. HTML code is almost non-existent (0.7%, 27 affected apps).

3) Discussion: We found that the use of tokens in open-source apps is not as common as in closed-source apps. One explanation could be that the fees associated to web services do not pay off for open-source apps which mostly do not generate any revenue.

Several embedded languages are actively used within mobile apps. While SQL is relatively common in both open-source and closed-source apps, JavaScript is much more commonly used in the latter.

IV. Web API Security Smells

In this section, we present the security smells that we found in web communication during investigation of the tool's results, by manually investigating 100 apps, and by analyzing the responses from requests to each of the 9 714 web API URLs extracted from apps in our dataset. We classify the smells into client side (*i.e.*, within mobile apps), and server side (*i.e.*, on the API servers). For each smell we report the security *issue* at stake, the potential *consequences* for users, the *symptom* in the code (*i.e.*, the code smell), and the recommended *mitigation* strategy of the issue, principally for developers.

We used the results from the manual analysis explicitly to identify security issues, but not to perform any quantitative evaluation. In this section, we do not report any number of occurrences found in the tool's results, because those either have been discussed in the previous section, or the task would require additional research to gather qualitative results.

In our analysis, we could identify eight web API security smells, of which three were in apps and five in server implementations. Two of the three web API app security smells could be mitigated, if only secure HTTPS channels would be

used for communication. We have not yet reported our findings to developers or marketplaces.

A. Client-side

We identified three client-side web API security code smells.

- **Credential leak**. We found hard-coded API keys, login information, and other sensitive data, *e.g.*, email addresses, in the source code. Several of the retrieved data were valid at the time of our investigation: We could access *Google Maps*, *Mapquest*, *OpenWeatherMap*, the *San Francisco transit* API, and a *Telegram* bot.
  Issue: Credentials issued to app vendors are prevalent in apps that use web APIs, and they are statically stored in the Java software to perform the queries. However, the software can be decompiled into source code, which renders the data extraction trivial.
  Consequently, web services can be misused by people who have gained access to unique credentials. Such services allow impersonation, phishing, information leaks, fake messages, or financial infringements for the app developers due to API overuse or lockdowns.
  Symptom: Query keys like `key`, `token`, `user`, `username`, `password`, `pw` are used in web requests and the corresponding values are statically stored in the apps.
  Mitigation: Developers should avoid using access tokens and logins of corporate accounts for apps. Instead, a unique child token based on the corporate token should be assigned to every user. If this option is unavailable, web relay APIs can be provided to the apps which forward the requests to the final destination without disclosing any credentials.
- **Embedded languages**. We found apps that assemble CSS, HTML, or JavaScript code programmatically using external input. In many apps, such constructed code is executed within a `WebView` or Android's UI framework, which is inspired by *Java Swing* and supports HTML elements. Similarly, we found assembled SQL statements that are executed in the local SQLite database engine. In two apps we found assembled shell commands sent over an SSH connection.
  Issue: An attacker could gain control over the app's visual representation, the behavior, the data storage, or the corresponding server by exploiting such code. Shell commands such as `String command = "touch /home/" + username + "/.toolConfig/configuration";` allow an adversary to execute commands on a server by letting the variable `username` be `;echo 'executes on server';`.
  Consequently, for HTML and CSS, an attacker could change the appearance of existing web elements to make space for additional ones, *e.g.*, by reducing the font size of existing text to make it impossible to read and at the same time injecting additional text in regular size. Such changes can trick users into taking unintended actions. With JavaScript, an attacker could gain access to the *Document Object Model (DOM)* of the app's webpage and extract or alter the visible content. Such changes expose sensitive user data, or mislead users through altered information. SQL allows adversaries to perform arbitrary actions on the database, *e.g.*, altering and deleting existing data, or inserting new data. This leads to data loss, corruption, or leaks for the users. Through shell commands an adversary could potentially gain elaborated remote access to the server's operating system. Threats range from DoS attacks to sensitive user information leaks and corporate network infiltrations by disabling security measures and installing malicious software on the server.
  Symptom: At least one statement is manually assembled with the help of external data, *e.g.*, `"<html><body>" + example + "</html></body>"` or `"color:" + color + ";"`. HTML/CSS: common tags or properties occur, *e.g.*, `"<html>"`, `"<body>"`, or `"color:"`. JavaScript: identifiers exist in the app, *e.g.*, `function()`, `<script`, `js=`. SQL: keywords are used in the app, *e.g.*, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REPLACE`, `TRUNCATE`. Shell: commands are not trivial to detect, because developers use a variety of different commands, *e.g.*, `sudo`, `rm`, `cp`, `mv`, `ls`, `exec`, `attrib`, `chmod`, `touch`, *etc.*
  Mitigation: Developers should not use external input when assembling embedded languages, but try to embed the content into the app installation or update package. Static code should be used whenever possible. If dynamic code is required, the built-in sanitizing classes must be used, *e.g.*, `PreparedStatement` for SQL code. User input should *never* be trusted. In general, *any* untrustworthy input must not be used before it is properly escaped and sanitized.
- **Insecure transport channel**. Web API communication relies on HTTP or HTTPS; both variants exist in apps.
  Issue: HTTP does not provide any security; neither the address, nor the header information or the payload are encrypted.
  Consequently, any attacker with access to the transmitted data can read or alter all plain text messages. User data leaks, corruptions, losses, or impersonation are probable.
  Symptom: HTTP URLs are used to establish connections to web APIs.
  Mitigation: HTTPS instead of HTTP URLs must be used for any web communication.

B. Server-side

For every collected API in our dataset, we accessed the corresponding web server and stored the response. We were particularly interested in information such as operating system identifiers, used software modules, and version numbers, which we could initially identify during the manual analysis of a sample of the server responses. We then crafted a number

of search queries to detect occurrences of such features and applied them to our dataset.

We have identified five server-side web API security code smells.

- **Disclosure of API implementation code**. Error messages provide valuable information regarding the implementation of a running system. We found web APIs that leak internal error states and use *status codes* in a different way than what is specified by the RFC7231.[9,10]
  Issue: Error messages that include the relevant stack trace are transmitted as plain text in the server's message response body. Such a message reveals information like the used method names, line numbers, and file paths disclosing the internal file system structure and configuration of the server.
  Consequently, adversaries can obtain detailed information about the service implementation, which may lead to an exploit.
  Symptom: When an invalid request is received, a server responds with a detailed error message containing information that is not required by any user of the API.
  Mitigation: If the used framework provides an option to turn off diagnostic or debug messages: this feature should be used. Otherwise, an API gateway in between the client and the server should filter such responses and deliver regular HTTP `500` messages to the client instead.
- **Disclosure of version information**. Besides useful connection parameters, HTTP headers provide information regarding the software architecture and configuration of a running system. We spotted in the reported HTTP headers version information of web server daemons and API implementation frameworks.
  Issue: We encountered outdated software that suffers from severe security vulnerabilities. For instance, we observed a server that returned `X-Powered-By: PHP/5.5.23` in the response header. This PHP version is at the time of writing more than 6 years old, and a quick search in the Common Vulnerabilities and Exposures (CVE) database showed that this framework suffers from 69 known security vulnerabilities, six of which received the most severe impact score of 10.[11]
  Consequently, the vulnerabilities range from simple DoS attacks, access control bypassing, and cross-site scripting to arbitrary code execution on the server.
  Symptom: One of the following header keys exists in the response header: `engine`, `server`, `x-aspnet-version`, or `x-powered-by`.
  Mitigation: If the used software provides an option to turn off the publishing of version information: this feature should be used. Otherwise, an API gateway in between the client and the server should remove the affected keys and deliver messages with sanitized HTTP headers to the client instead.
- **Lack of access control**. Authentication by a user name and a password provides tailored experiences to end users, *e.g.*, individual chat logs or friend lists, and at the same time enables access control to separate and protect sensitive user data.
  Issue: The access to sensitive data or actions is not restricted by a sane authentication mechanism such as a user name and password pair, instead, easy-to-forge identifiers or no identification data at all are used to secure the access. We found several APIs that did not use any authentication or authorization mechanisms, although they host sensitive data, *e.g.*, for car rental services and accounting. In one app we found code to access an exposed SQL database interface.
  Consequently, every internet user can access sensitive data or perform unauthorized actions including the reading, modification, and deletion of arbitrary user data. We could access information from such APIs, *e.g.*, real-time location data of rental cars and transaction histories on different bank accounts. In one case, we were also able to create new users in the system. Exposing database or other interpreter interfaces with broken authentication allows adversaries to execute arbitrary statements on the server.
  Symptom: A web API server hosts sensitive data or provides actions which would require elevated access rights. The server responds without asking for any login information, that is, no HTTP headers or keys related to personal information are used in the API, *e.g.*, `username`, `password`, or `pw`. The server requires query keys with names of programming languages, *e.g.*, `sql`, and responds when such variables hold a statement in that language, *e.g.*, `SELECT table_name FROM all_tables;`. The decision finding of data sensitivity or elevated actions is non-trivial and involves manual reasoning [11]. Therefore, we cannot infer general purpose terms.
  Mitigation: Application architects have to implement authentication, favorably multi-factor authentication, whenever sensitive data or elevated operations are involved in the process. All user data, and location data in general, have to be considered as sensitive. Developers should never expose interpreter interfaces to a web service without prior authentication and input validation. REST interfaces for specific tasks should be created, preferably each using static statements that do not rely on any user input.
- **Missing HTTPS redirects**. In contrast to HTTPS, HTTP does not provide any security: neither the URL, nor the header information and embedded content are encrypted. We found servers that do not redirect the clients to encrypted connections although they would have been supported.

[9] https://tools.ietf.org/html/rfc7231

[10] Although HTTP servers should reply with the status code `200` to indicate a successful request, we noticed that some servers use this status code when an error has occurred.

[11] https://www.cvedetails.com/vulnerability-list/vendor_id-74/product_id-128/version_id-183021/PHP-PHP-5.5.23.html

Issue: Web API servers do not redirect incoming HTTP connections to HTTPS when legacy apps try to connect, or users manually configure a URL without adding a proper `https://` prefix.
Consequently, the transmitted data remains visible and changeable to anyone within the communication path.
Symptom: For an HTTP web API request, a server does not deliver an HTTP `3xx` redirect message which points to the corresponding HTTPS implementation of the web API.
Mitigation: A server should not offer legacy HTTP services. If they are still required due to legacy clients with hardcoded HTTP URLs, redirects should be provided to guide all clients to the secure version.

- **Missing HSTS**. HTTP header information is used to properly set up the connection by specifying various communication parameters, *e.g.*, the acceptable languages, the used compression, or the enforcement of HTTPS for future connection attempts, a feature which is called *HTTP Strict Transport Security (HSTS)*. HSTS provides protection against HTTPS to HTTP downgrading attacks, *i.e.*, when a user once accessed a web resource in a secure environment (at home or work), the client knows that the resource needs to be accessed *only* through HTTPS. If this is not possible, *e.g.*, at an airport at which an attacker tries to perform MITM attacks, the client will display a connection error. Hence, HSTS should be used in combination with HTTP to HTTPS redirects, because the HSTS header is only considered to be valid when sent over HTTPS connections. We found servers that do not enforce clients to remain on the secure channel for future requests.
  Issue: Servers do not leverage the HSTS feature.
  Consequently, in unprotected public networks or networks under external supervision, if an attacker sets up a fake gateway which runs SSLsniff,[12] the provided services remain vulnerable, because transmitted data is visible and changeable.
  Symptom: A server does not deliver the HTTP HSTS header `Strict-Transport-Security: max-age=31536000; includeSubDomains` for an HTTPS request.
  Mitigation: In combination with HTTP to HTTPS redirects, the HSTS header should be used in all HTTPS connections.

V. Threats to Validity

The main threat to validity is the completeness of this study, *i.e.*, it is not guaranteed that we found all major libraries used for web communication in Android apps.

There may be bias in the apps that we selected for this study. We included all open-source apps that were available on *F-Droid*, but they may not be representative of the whole open-source app community. We collected random closed-source apps that were freely available on the *Google Play* store, but paid apps or the apps on third-party stores may have different characteristics.

We only mined web APIs that were available in the source code; our tool suffers from the inherent limitations which come with static source code analysis. We developed a lightweight analysis, which is not path sensitive. We opted for this design because, during manual inspection of network APIs in the apps, we noticed that these APIs are usually free of conditional statements and loops. Furthermore, we had to decompile closed-source apps for analysis, which introduces further threats to the validity of our results. For instance, the app code and its library code are not easy to discern automatically, and therefore the libraries in such apps may have influenced our findings.

We did not evaluate how complete the tool results are for every app, but just a small number. There is a threat to construct validity through potential bias in our expectancy. However, we examined the tool results for 50 apps, and confirmed that 90% led to successful communication with the web APIs.

VI. Related Work

In previous work, we defined the notion of security code smells and investigated their appearance in 46 000 closed-source Android apps from the official market [6]. We identified 28 different security smells in five different categories, and found that *XSS-like Code Injection*, *Dynamic Code Loading*, and *Custom Scheme Channel* are the most prevalent smells. In a follow-up work, we studied the prevalence of *Inter-Component Communication (ICC)*-related security smells in more than 700 open-source apps, and manually inspected around 15% of the apps to assess the extent to which identifying such smells uncovers ICC security vulnerabilities [12]. We found that almost all apps suffer from the *Common Task Affinity* smell, and that *Unauthorized Intent* and *Custom Scheme Channel* are prevalent among mobile apps. Furthermore, we discovered that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. The manual investigation of 100 apps showed that our tool successfully found many different ICC security code smells, and about 43% of them in fact represent vulnerabilities.

Zhou *et al.* harvested free email and Amazon AWS cloud service credentials with their tool *CredMiner* from more than 36 500 apps from various Android markets [2]. In their case studies, they mention unprotected credentials within the app's source code, obfuscated credentials using a *Base64* encoding, and encrypted credentials, however, in those cases the decryption key has also been found in the app's source code. They alarmingly found that more than every second app using such a service leaked the developers' credentials in the apps' source code. Making matters worse, more than 77% of those collected credentials were valid at the time of the experiment. Such credentials will present a massive threat in the mid-term future, as many of those credentials cannot be easily replaced without

[12] https://github.com/moxie0/sslsniff

temporarily abating the experience of millions of users, but in the meantime they can be easily exploited by attackers.

Rapoport *et al.* studied web requests in Android apps [1]. They demonstrated that a large number of web requests are not immediately traceable to source code and need dynamic analysis. For instance, URLs may originate in app resources, *e.g.*, XML files or Gradle build scripts, they may stem from the content received from previous web requests, or they might be assembled by JavaScript code at run time. In contrast, a significant proportion of URLs are only detected by static analysis: the dynamic analysis may simply fail to produce desired results due to a lack of code coverage during instrumentation.

Zuo *et al.* analyzed 5 000 top-ranked apps in Google Play and identified 297 780 URLs [3]. They fed the URLs to a harmful URL detection service at VirusTotal, and found 8 634 harmful URLs. The harmful URLs have been classified into three different non-distinct threat categories: phishing (23%), malicious sites (37%), and malware (43%). For the malware category, one interesting example they mention is an APK file download triggered by an app, which itself tries to obtain superuser access to the device by exploiting Linux kernel vulnerabilities.

Mendoza *et al.* studied the inconsistencies in input validation logic between apps and their respective web API services [5]. They developed a tool to extract requests to web API services from an app, and to infer sample input values that violate the implemented constraints found in the app, such as email address or JSON content validation executed on the client side. They then analyzed app-violating request logic on the server side via black box testing. From a set of 10 000 popular Android apps, they found 4 000 apps that do not properly implement input validation for web API services. Investigation of web API hijacking vulnerabilities in 1 000 apps showed that the security and privacy of millions of users are at risk.

In summary, we could not find any publicly available tool that researchers can use to study web APIs. Also, existing work usually focused on the use of `java.net` APIs, and did not study several third-party libraries to implement network communication in Android apps. Finally, to the best of our knowledge, dissecting the distribution of elements that comprise the web APIs, and the use of embedded languages, is never studied.

VII. Conclusion

We manually reviewed 160 Android apps to compile a list of commonly used network and data conversion libraries and to learn how they are used in these apps. Based on our findings, we developed a lightweight static analysis tool that identifies network-related APIs, and extracts communication information such as the web APIs, and the associated JSON headers. With the help of our tool we successfully analyzed the network-related information within 450 closed-source and open-source apps. We found that in both open-source and closed-source apps network communication is mainly developed using *java.net* classes. Amongst the third-party libraries we found that *OkHttp* and *Retrofit* are used the most. By far the most used value type in JSON data is `STRING`.

We realized that closed-source apps substantially rely on advertisement services, and that they tend to have more complex URL paths consisting of more path segments. Surprisingly, the secure HTTPS protocol is used in the majority of extracted web APIs from open-source applications, but the opposite is true for closed-source apps. Obviously, when embedded languages are used along with manual string concatenations, the attack surface for code-injection attacks increases. Nevertheless, we could identify numerous such cases during the manual examination of the web APIs, *i.e.*, embedded SQL and JavaScript content was rather common within web communications. Even worse, we found many more issues on the server-side: unnecessary disclosure of server configurations, outdated web servers and language interpreters with known security vulnerabilities, leaks of internal error messages, and other sensitive data. Finally, we also found private APIs without any kind of authentication or authorization mechanisms.

We conclude that a lightweight static code analysis is very helpful in mining web APIs, and that the impact of embedded code in web API requests and the hardening of servers has been deeply underestimated.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Assistance” (SNSF project No. 200020-181973, Feb. 1, 2019 - April 30, 2022). We also thank CHOOSE, the Swiss Group for Original and Outside-the-box Software Engineering of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

[1] M. Rapoport, P. Suter, E. Wittern, O. Lhóták, and J. Dolby, “Who you gonna call?: Analyzing web requests in Android applications,” in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR ’17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 80–90. [Online]. Available: https://doi.org/10.1109/MSR.2017.11

[2] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, “Harvesting developer credentials in Android apps,” in *WISEC*, 2015.

[3] 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*, ser. WWW ’17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 867–876. [Online]. Available: https://doi.org/10.1145/3038912.3052609

[4] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, “Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces,” in *NDSS*, 2016.

[5] A. Mendoza and G. Gu, “Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities,” in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 756–769.

[6] M. Ghafari, P. Gadient, and O. Nierstrasz, “Security smells in Android,” in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2017, pp. 121–130.

[7] L. Qiu, Y. Wang, and J. Rubin, “Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. ACM, 2018, pp. 176–186. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213873

[8] F. Pauck, E. Bodden, and H. Wehrheim, "Do Android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, 2018, pp. 331–341. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236029

[9] C. Corrodi, T. Spring, M. Ghafari, and O. Nierstrasz, "Idea: Benchmarking Android data leak detection tools," in *Engineering Secure Software and Systems*, M. Payer, A. Rashid, and J. M. Such, Eds. Cham: Springer International Publishing, 2018, pp. 116–123. [Online]. Available: http://scg.unibe.ch/archive/papers/Corr18a.pdf

[10] W. E. Winkler and Y. Thibaudeau, *An application of the Fellegi-Sunter model of record linkage to the 1990 US decennial census*. Citeseer, 1991.

[11] G. O. M. Yee, "Model for reducing risks to private or sensitive data," in *Proceedings of the 9th International Workshop on Modelling in Software Engineering*, ser. MISE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 19–25. [Online]. Available: https://doi.org/10.1109/MiSE.2017..6

[12] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, "Security code smells in Android ICC," *Empirical Software Engineering Special Issue*, 2018.