The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators

Marten Oltrogge*, Erik Derr*, Christian Stranksy*, Yasemin Acar‡, Sascha Fahl‡
Christian Rossow*, Giancarlo Pellegrino*†, Sven Bugiel* and Michael Backes*
*CISPA, Saarland University, †Stanford University, ‡Leibniz University Hannover

***Abstract*—Mobile apps are increasingly created using *online application generators* (OAGs) that automate app development, distribution, and maintenance. These tools significantly lower the level of technical skill that is required for app development, which makes them particularly appealing to *citizen developers*, i.e., developers with little or no software engineering background. However, as the pervasiveness of these tools increases, so does their overall influence on the mobile ecosystem's security, as security lapses by such generators affect thousands of generated apps. The security of such generated apps, as well as their impact on the security of the overall app ecosystem, has not yet been investigated.**

We present the first comprehensive classification of commonly used OAGs for Android and show how to fingerprint uniquely generated apps to link them back to their generator. We thereby quantify the market penetration of these OAGs based on a corpus of 2,291,898 free Android apps from Google Play and discover that at least 11.1% of these apps were created using OAGs. Using a combination of dynamic, static, and manual analysis, we find that the services' app generation model is based on boilerplate code that is prone to reconfiguration attacks in 7/13 analyzed OAGs. Moreover, we show that this boilerplate code includes well-known security issues such as code injection vulnerabilities and insecure WebViews. Given the tight coupling of generated apps with their services' backends, we further identify security issues in their infrastructure. Due to the blackbox development approach, citizen developers are unaware of these hidden problems that ultimately put the end-users sensitive data and privacy at risk and violate the user's trust assumption. A particular worrisome result of our study is that OAGs indeed have a significant amplification factor for those vulnerabilities, notably harming the health of the overall mobile app ecosystem.

I. Introduction

The proliferation of *online application generators* (OAGs) that automate development, distribution and maintenance of mobile apps significantly lowers the level of technical skill that is required for application development. As a consequence, creating platform-specific apps becomes amenable to a wide range of inexperienced developers. This trend that developers with *"little or no coding or software engineering background"* [19] create software with low-code or no-code platforms has become known as *citizen developers* [19], [28] and has recently received tremendous momentum across the industry. Moreover, many OAGs additionally promise to decrease the app's overall development and maintenance costs since they offer functionality for taking care of various tasks across all phases of an app's life cycle.

However, this convenience comes at the cost of an opaque generation process in which the user/developer has to fully trust the generated code in terms of security and privacy. A large body of literature has revealed various security problems in mobile apps, such as permission management [37], insecure SSL/TLS deployment [17], [36], misuse of cryptographic APIs [15], and inter-process communication [12]. These flaws could be attributed to poorly trained app developers that implemented application features in an insecure manner. With the increasing use of OAGs the duty of generating secure code shifts away from the app developer to the generator service. This leaves the question whether OAGs can provide safe and privacy-preserving default implementations of common tasks to generate more secure apps at an unprecedented scale. However, if they fail, their amplification effect will have a drastic negative impact on the already concerning state of security in mobile apps. As of now, the security implications of OAGs have not been systematically investigated yet, and, in particular, their impact on the security of the overall app ecosystem remains an open question: *"Do online app generators have a positive or negative impact on the overall app ecosystem?"*

Our contribution—In this paper, we present the first classification of commonly used OAGs for Android apps on various characteristics including their supported workflows, automation of the app development life cycle and multi-platform support. We proceed by showing how to uniquely fingerprint generated apps in order to link them back to their generator. We thereby quantify the market penetration of these OAGs based on a corpus of 2,291,898 free Android apps from Google Play and discover that at least 11.1% of these apps were created using online services. This noticeable market penetration already shows that potential security mistakes and misconduct by OAGs would impact thousands of apps and impose a danger for the overall health of the Android ecosystem.

Analyzing the security of OAGs is non-trivial due to the absence of a documentation of the development process. Instead, these services offer a fully-automated, opaque app generation process without the possibility to write custom code. App developers have to fully trust that the generated code follows security best practices and does not violate the end-users' privacy. In order to shed light onto the black-box generation process, we perform a comprehensive security audit on apps created by these services using a combination of dynamic, static, and manual analysis. This allows us to document their internal workflow and to discover a new app generation model based on boilerplate code. We then demonstrate that 7 out of 13 analyzed online application generators fail to defend against reconfiguration attacks, thus opening new attack surfaces of their generated apps. We further analyze whether the generated boilerplate code adheres to Android security best practices and whether it suffers from known security vulnerabilities identified by prior research. Our results, both on self-generated apps as well as apps randomly picked from Google Play, suggest that OAGs are responsible for producing vulnerable code including SSL/TLS verification errors, insecure `WebViews`, code injection vulnerabilities and misuse of cryptographic APIs. Within our data set, all analyzed application generators suffer from at least one of these vulnerabilities, combined affecting more than 250K apps on Google Play. Finally, we have a dedicated look onto the services' infrastructure security. Online service-generated apps are typically bound to their providers' backend servers, e.g. for license checks when the service charges a monthly fee. In addition, some services even provide complete user-management modules that require connections to backend servers. Any of this functionality requires a secure client-server communication since either sensitive (user) data is exchanged or configuration files for the boilerplate code is transmitted. However, our analysis reveals that many services build on an insecure and vulnerable infrastructure, e.g. by using insecure SSL/TLS server configurations, mixed usage of HTTP/HTTPS and usage of outdated SSL libraries.

We conclude with a thorough discussion of our findings, including potential alleys worth pursuing in future research on generating secure code for such module-based app builders. In summary, we make the following tangible contributions:

- We present the first classification of commonly used OAGs for Android, accounting for various characteristics such as the supported workflows, automation of the app development life cycle, multi-platform support, and their boilerplate-based app generation model.
- We show how to fingerprint generated apps and how to quantify the market penetration of OAGs by classifying 2,291,898 free apps from Google Play: at least 255,216 apps (11.1%) are generated using OAGs.
- We derive OAG-specific attacks, such as reconfiguration and infrastructure attacks and show how these services fail in protecting against these attacks.
- We conduct a comprehensive security audit to show that boilerplate code generated by any of the analyzed services violates security best practices and contains severe security vulnerabilities. To estimate the real-world impact, we validate our findings on real, generated apps on Play.

Outline—This paper is organized as follows. We give a general overview of mobile app generators in Section II and a classification of commonly used OAGs in Section III. We describe the methodology of our security audit in Section IV, present new, OAG-specific attack classes in Section V and analyze known security issues in Section VI. Finally, we thoroughly discuss our findings in Section VII, before concluding in Section VIII.

II. Overview of Mobile AppGens

Application generators are tools for partially or even completely automating app development, distribution, and maintenance. The advantages of using application generators are manifold. First, they enable developers to abstract away from implementation aspects and to instead focus only on the conceptual behavior of the application in terms of high-level functionality. Second, they provide functionality beyond core app generation, including support for app compilation, app dissemination, and distribution of patched versions. Third, they offer support for making an app equally applicable to multiple competing architectures, such as Android and iOS. Finally, they may even provide support for recurring, extended app functionality such as user management, user login, and data submission to back-end servers. In this section, we give an overview of commonly used AppGen types within the Android ecosystem based on their supported workflows. Our investigation resulted in three distinct categories of application generators: *standalone frameworks*, *online services*, and software development services that we dub *Developer-as-a-Service* as explained in the following.

A. Standalone frameworks

Standalone frameworks constitute tools that offer a core set of abstract application functionality which are then refined by additional code from the app developer. These frameworks typically expect a program written in a platform-independent language as input, e.g., JavaScript and HTML, or C#, and then package user-provided code together with an execution engine into a native app. Many of those frameworks offer plugins that provide commonly used functionality (e.g., in-app browsers or advertisement) or even skeletons for entire apps, which are provided by plugin developers. While these frameworks assist in the creation of an app, they offer little to no support for further phases of an app's life cycle such as app dissemination and distribution of patched versions. To date, these unsupported tasks are typically performed by the app developer. Prominent examples of standalone frameworks are *Xamarin*, *Apache Cordova*, and *PhoneGap*.

B. Online Services

Online services or online application generators (OAGs) enable app development using wizard-like, point-and-click web interfaces in which developers only need to add and suitably interconnect UI elements that represent application components (e.g., email or login forms, in-app browser, QR scanner, social networking widgets, etc.). There is no need and typically also no option to write custom code. For some of these components, they may even provide the necessary infrastructure such as user management, user login, and data submission to back-end servers maintained by the service provider. These online service tools are thus accessible even for laymen developers that lack any prior experience in app development. Moreover, online services offer support for automatically distributing apps over popular channels such as the Google Play Store. In addition to extensively supporting core tasks of the software development life cycle, online services provide business intelligence and analysis features such as audience reports and dashboard for analytics. Prominent examples of online services are *Andromo* and *Biznessapps*.

C. Developer-as-a-Service

Developer-as-a-Service does not even expect developers to contribute to the technical development of an app. Instead, the developer rather acts as a customer that orders the whole app creation from a contracted app development service, which then delegates the app creation to a team of app developers that are expected to develop the application based on a set of explicitly spelled out customer requirements. Such requirements are typically collected over the phone or via emails. Widely known examples are *CrowdCompass* and *QuickMobile*, services specialized to create customized event and conference apps.

III. Online App Generators

This section presents a classification of apps of commonly used OAGs and an analysis of market penetration and characterization of OAG-generated apps.

A. Classification

We used Google search queries to identify a rich set of common application generators. More concretely, we simulated a user who is searching for an application generator using search terms including but not limited to: "app maker android", "android generate app", and "{business, free, diy, mobile} (android) app {generator, creator, maker}". For each result, we selected the first five entries after removing duplicates. We also issued queries to online resources that offer technical and popularity reviews of application generators such as Appindex[1], Werockyourweb[2], Quora[3], and Businessnewsdaily[4]. We excluded non-online-services and application generators from our analysis, when we were not able to meaningfully assess their market penetration, i.e., we could not determine whether any available app was generated using these particular application generators, see Section III-B.

We have classified all application generators along four dimensions: freeware, multi-platform support, components, and publishing, see the columns on "Classification" in Table 1.

Freeware—Some application generators can be freely used (●), while others require a monetary investment (○).

Multi-platform support—While traditional app development requires developers to write distinct apps for each mobile platform like Android, iOS, and Windows Mobile, many application generators allow developers to develop for one platform and then automatically generate "native" apps for additional platforms (●). We write (○) if this multi-platform functionality is not provided.

Components—Many application generators offer supplementary components for common tasks such as ads, app analytics, crash reporting, and user management. We write (●) if features can be conveniently added via simple web forms, e.g. by means of checkboxes; (◐) if users have to rely on visual programming interfaces to add and remove features; and (○) if supplementary components are not offered.

Publishing support—Conventional app development requires developers to write code, compile, and sign an APK, and then distribute it to their users. While writing code, compiling and signing an APK is arguably a smooth process using dedicated IDEs (e.g., Android Studio), app distribution usually requires further manual effort: register a Google Play account (or an account for an alternative market), upload the app, add description text and publish. Some application generators offer to automate this complete chain from producing and signing an app to publishing it on one or multiple markets (●) , while others only automate parts of this support chain(◐) or do not offer support at all (○).

B. Fingerprinting Application Generators

Once we established and classified our set of online services, we aimed at quantifying the *market penetration* of the individual application generators. To do this in a meaningful manner without relying on bold marketing claims, we identify the number of Android apps generated by the individual application generators. To this end, we first identified unique features of application generators as fingerprints, and then used these features to classify a large corpus of 2,291,898 unique free Android apps. We collected these apps from the Google Play Store between August 2015 and May 2017 using a crawler. The crawler starts with a set of URL seeds and subsequently follows the recommendation links to explore the store. Our crawler revisits previously found apps once per day and downloads them only if new versions are available. Our analysis considered only the latest version of each app.

1) Features: We extracted unique features of application generators using differential analysis between a *baseline* app[5] and sample apps from each application generator. We then manually reverse-engineered all sample apps and created a diff between our baseline app and the sample apps based on the components of a typical Android app. To create this diff, we first analyzed the composition of an Android app, and then identified all parts that can be used to tell apart generated apps and manually developed apps. Our analysis resulted in the following four features (the latter two can be sub-classified, totaling six distinct *Fingerprinting Features*, see Table 1):

App Package Names—The first distinctive feature of apps is the app package name. Package names are unique text

[1]http://appindex.com/blog/app-builders-app-makers-list/

[2]http://www.werockyourweb.com/mobile-app-builder/

[3]https://www.quora.com/What-are-the-best-mobile-app-creators-for-non-coders-both-free-and-paid

[4]http://www.businessnewsdaily.com/4901-best-app-makers-creators.html

[5]We used the default Android IDE Android Studio to create a single-activity "Hello World" Android app that did not include any external third-party library

Table 1: Classification and fingerprinting of online services sorted by category and app count. For each AppGen, we found multiple distinguishing fingerprint features, but, in all cases, a single feature is already sufficient to uniquely classify an application generator.

	Classification				Fingerprinting Features							
Online Service	Freeware	Multi-platform	Components	Publishing	Package Name	Code Namespace	Files in Package	File Content	Sign. Cert.	Sign. Cert. Subject	Market (# of apps)	URL
Seattle Cloud	○	●	●	●	○	●	●	○	●	●	60,314	http://seattleclouds.com/
Andromo	●	○	●	○	●	●	○	○	●	●	45,850	http://www.andromo.com/
Apps Geyser	●	●	●	◐	○	○	○	●	●	●	29,190	http://www.appsgeyser.com/
Biznessapps	○	●	●	●	○	●	○	○	●	●	27,130	http://www.biznessapps.com/
Appinventor	●	○	◐	○	○	●	○	○	○	○	25,338	http://appinventor.mit.edu/explore/
AppYet	●	○	●	○	○	●	○	○	●	●	15,281	http://www.appyet.com/
Como	○	●	●	◐	●	●	○	○	●	●	10,894	http://www.como.com/
Tobit Chayns	●	●	●	●	●	●	○	○	●	●	8,242	http://en.tobit.com/chayns
Mobincube	●	●	●	○	●	●	●	○	●	●	8,074	http://www.mobincube.com/
Appy Pie	●	●	●	◐	●	●	●	○	●	●	4,445	http://www.appypie.com/
Appmachine	○	●	●	◐	●	●	●	●	●	●	4,409	http://www.appmachine.com/
Good Barber	○	○	●	◐	●	●	●	○	○	●	3,622	http://www.goodbarber.com/
Shoutem	○	●	●	●	●	●	○	○	●	●	2,638	http://www.shoutem.com/
App Yourself	○	●	○	●	●	●	●	●	●	●	2,273	http://appyourself.net/
Mippin App Factory	○	○	●	◐	●	●	●	○	○	●	1,785	http://www.mippin.com/appfactory/
Apps Builder	○	●	●	◐	●	●	●	○	●	●	1,191	http://www.apps-builder.com/
Appmakr	●	●	●	◐	●	●	○	●	○	○	1,058	http://appmakr.com/
appery.io	○	●	○	○	●	●	●	●	○	○	846	https://appery.io/
Apps Bar	●	●	●	◐	●	●	●	○	○	●	700	http://www.appsbar.com/
Mobile Roadie	○	○	●	◐	●	●	●	●	○	○	581	http://mobileroadie.com/
App Gyver	○	●	○	○	●	●	●	●	○	○	386	http://www.appgyver.io
Appconfector	○	●	●	●	●	●	○	●	○	○	337	http://www.appconfector.de
Rho Mobile Suite	●	●	○	○	●	●	●	○	○	○	216	http://rhomobile.com/
Appsme	●	○	●	●	○	●	○	○	○	○	158	http://www.appsme.com/
App Titan	○	○	●	◐	●	●	○	○	○	○	152	http://www.apptitan.de/
Applicationcraft	○	●	●	○	●	●	○	●	○	○	100	http://www.applicationcraft.com/
Paradise Apps	●	●	●	◐	○	●	●	○	○	○	3	http://www.paradiseapps.net/
Eachscape	○	○	●	●	●	●	○	○	○	○	3	http://eachscape.com/

● = yes/applies; ; ◐ = applies partly; ○ = no/does not apply

strings that are used by Google Play to unambiguously identify apps. Application generators often use patterns for generated apps that in turn can be used as a distinctive feature, e.g., `com.Tobit.*` or `{com|net}.andromo.dev*`.

Code Namespaces—Java code is organized in namespaces and similar to package names, application generators may use particular namespaces that we can leverage for our classification. Andromo apps, for example, include code namespaces that contain the substring `.andromo.dev` or the prefix `com.andromo`. A similar example are Tobit Chayns apps, which include code namespaces with the prefix `com.Tobit.android.slitte.Slitte`. In contrast, Apps Geyser apps include code namespaces with the prefix `com.w*`, which is not suitable for classification purposes due to its ambiguity (see Section III-B2 for further details).

Signing Keys—Before uploading an app to Google Play, APKs must be digitally signed. This is a security mechanism to ensure that app updates are distributed by the same entity (e.g., developer). A single key is often used to sign multiple apps, e.g., Seattle Cloud uses a unique key to sign all its apps. We can use this single-key pattern to fingerprint the application generator. Whenever application generators use distinct keys, we can still use further information about the certificate to fingerprint the app, e.g., if all keys have the same subject. AppYet apps, for instance, all share the same certificate subject `/C=CA /ST=ON /L=Oakville /O=AppYet /CN=www.appyet.com`.

Files—In addition to an app's code, apps include a list of files such as images, CSS, or configuration files. These files can be used for the classification as well. For example, AppyPie apps include the file `appypie.xml` in the `assets/www` app folder. We moreover use file content for the classification, e.g., we identify AppsGeyser apps, by verifying whether the elements `<webWidget>` and `<registeredUrl>` of `res/raw/configuration.xml` contain the URL `appgeyser.com`.

2) *Methodology:* We start our classification by extracting the aforementioned features from our set of sample apps. We discovered that for each AppGen there are multiple distinguishing features that allow to link the app back to its generator. We further found that in all cases a single feature

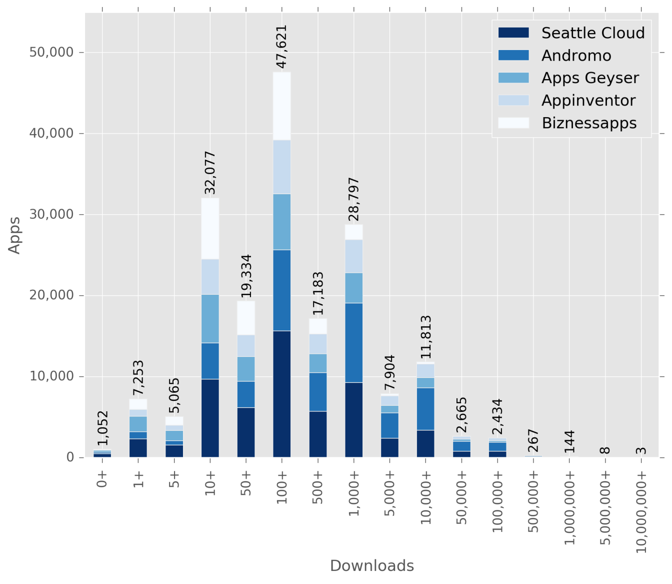

Figure 1: Download counts for all apps from the top 5 OAGs using buckets provided by Google Play.

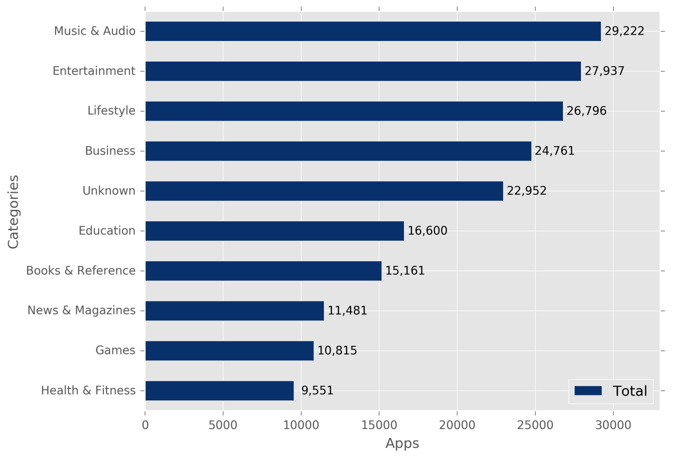

Figure 2: Distribution OAG generated apps by Google Play store categories. The *Unknown* category is for apps that are not classified by the Google Play store

would already be sufficient to unambiguously determine the originating service. The overall classification of application generators with respect to their features is depicted in Table 1 in the "Fingerprinting Features" columns.

As an orthogonal investigation, we also fingerprinted apps that have *not* been created by means of application generators. In this analysis, we considered the two major platforms for Android app development: Eclipse ADT[6]—support ended in August 2015—and Android Studio[7]. We manually investigated apps developed with both IDEs similarly as described for application generators. Our analysis revealed that Android Studio apps can reliably be identified based on the files' structure. Apps developed and compiled with Android Studio include a folder `res/mipmap` that stores launcher images. This folder structure was introduced in Android Studio 1.1[8]. Hence, only apps developed with at least version 1.1 of Android Studio can be identified using this feature. Neither Eclipse ADT nor application generators apps we analyzed exhibited this behaviour. In contrast, we could not find any reliable identification feature for Eclipse ADT apps. To avoid false positives, we limited our analyses to application generators apps and Android Studio apps we could reliably identify.

C. Market Penetration

Our app corpus currently consists of 2,291,898 free apps from Google Play. We managed to successfully classify 255,216 (11.14%) of these apps using our feature detection as generated by OAGs. This is a lower bound based on the OAGs we could reliably identify, since OAGs not considered in our classification might be responsible for additional apps. Based on meta data from Google Play, these generated apps account for more than 1.14 bn downloads. Detailed app counts per online service can be found in the *market* column of Table 1. The numbers suggest that the majority of OAGs is responsible only for a small fraction of the set of generated apps. In contrast, the five most popular OAGs account for 73% of all generated apps, i.e., 8.12% of our corpus. While the many OAG-generated apps have a small user base, there is also a larger number (>17k apps) with a significant number of at least 10,000 downloads (see Figure 1). Reasons for this distribution of download counts originate in the limited set of functionality offered by these services as opposed to traditional app development. In the following, we characterize OAG-generated apps based on the offered functionality.

D. OAG App Characterization

To better understand the potential complexity of the application logic that can be implemented with OAGs, we inspected online services' development IDEs to count the number and types of app components that can be connected to implement the indented logic. We call *component* an app element with a UI and that implements a specific task or functionality. For example, components can be simple UI elements (e.g., buttons, forms, views) or complex modules/plugins (e.g., QR scanners, calendar views, and login forms). The number of components varies across OAGs and it ranges from 12 of AppYet to 128 components of Seattle Cloud. This variety indicates the level of customization that each OAG offers. We also observed a great variety of components. For example, Biznessapps offers 48 components all suitable for business apps, e.g., membership management, mortgage calculator, and loyalty program management components. Another interesting example is Seattle Cloud which provides general-purpose components including simple UI elements such as menu items

[6]http://developer.android.com/tools/help/adt.html
[7]http://developer.android.com/tools/studio/index.html
[8]https://androidbycode.wordpress.com/2015/02/14/goodbye-launcher-drawables-hello-mipmaps/

and image areas, as well as complex ones such as complete PDF readers and barcode scanners.

This variety is also reflected in the different app categories on Google Play. The app category is a string value in the app metadata retrieved by our crawlers that identify the type of app[9]. The ten most popular categories of OAG-generated apps are shown in Figure 2. These categories cover apps with non-trivial logic such as *Business* (e.g., document editor/reader, email management, or job search apps), *Entertainment* (e.g., streaming video apps), and *Games*. These categories contain all of the top nine OAG apps with 5M+ downloads. An interesting aspect is that different OAG dominate different app categories, thus suggesting product specialization. For example, the most popular category of Seattle Cloud is *Entertainment* with 11,073 apps, whereas the most popular category for Biznessapps is *Business* with 9,030 apps.

IV. Analysis Methodology

We now focus on the security of apps generated with OAGs. The naive approach to analyze the security of our dataset is to test all apps systematically. However, we observe that OAG apps are produced in a streamlined process in which app developers do not contribute with source code. We thus hypothesize that the apps share an OAG-specific *boilerplate code* and, as a consequence, either all generated apps share a vulnerability or none. Accordingly, we first identify the app generation model for each OAG, before we analyze the security of the boilerplate code on both self-generated apps and apps from Google Play.

A. Boilerplate App Model

To identify boilerplate apps, we first build the ground truth with self-generated apps for different OAG. To this end, we select the thirteen most popular OAGs based on their market penetration in Table 1 and register as a customer. Among these, six online services (*Appmachine*, *Apps Builder*, *Biznessapps*, *Como*, *Seattle Cloud*, and *GoodBarber*) required us to pay a subscription fee to be able to generate apps and to rent backend resources, such as user management. For each online service we create the same three custom apps to test whether targeted code is generated depending on the selected modules or whether boilerplate code with an app-specific configuration file is output. We create the three custom apps as follows:

- **App1:** The first app constitutes the minimal app, i.e. the smallest app that can be generated in terms of functionality. In most cases, this app just displays a "Hello World" message to the user.
- **App2:** Builds on the minimal app but additionally performs web requests to a web server we control. We perform both HTTP and HTTPS requests to analyze the transferred plaintext data and to test whether we can inject malicious code and emulate web-to-app attacks [22], [13], [32].
- **App3:** The third app implements either a user login or a form to submit user data to our server (if the application generator provides modules for such functionality). We chose such common functionality as it is supported by the majority of AppGens and because handling user data usually requires special care in terms of security. In our test set, *Appinventor*, *Biznessapps*, and *Como* did not offer such additional modules.

[9]A complete list of categories and descriptions can be found https://support.google.com/googleplay/android-developer/answer/113475?hl=en

Table 2: Online services grouped by generation model: Monolithic boilerplate apps with static/dynamic configuration files and module-dependent boilerplate apps.

A. Monolithic Boilerplate Code		
	Static config	Dynamic config
A.1 Native application		
Apps Builder	●	●
Appmachine	○	—
Biznessapps	—	○
GoodBarber	●	●
Mobincube	●	◐
A.2 HTML + Native app		
Apps Geyser	●	—
Appy Pie	●	●
AppYet	◐	—
Como	●	●
Seattle Cloud	●	●
Tobit Chayns	—	○
B. Module-dependent Boilerplate Code		
Andromo		
Appinventor		

● = static config in plain / dyn. config loaded via HTTP
◐ = static config decryptable / HTTPS downgrade
○ = static config encrypted / dyn. config loaded via HTTPS

To test whether an AppGen generates boilerplate code or module-dependent code we analyze the bytecode file(s) of App1–App3 as well as 10 randomly selected apps from Google Play for each service. In the majority of cases, it is sufficient to compare the hash value of the *classes.dex* bytecode file to show that different apps have the exact same code. If the file hash differs we compute a Merkle hash tree over the class hierarchy including package, class and method instruction information to quickly estimate the code overlap (thereby following a similar approach to the one to detect third-party libraries in [6]). The results confirm our observation in which apps generated with OAG are based on a common boilerplate code. In particular, we can derive two distinct generation techniques: monolithic boilerplate apps with configuration files and module-dependent boilerplate code (see Table 2).

1) Monolithic Boilerplate Code: All but two online services generate apps with the exact same application bytecode, i.e. apps include code for *all* supported modules with additional logic for layout transitions independent of what the app developer has selected. Apps only differ in a configuration file that is either statically included in the apk file or dynamically loaded at runtime. Some OAGs support both options, e.g. to deliver app updates as config file updates without the need to change the apk file. Further this allows to bind the app to the service providers' backend servers. In this scenario, apks only need to be updated when the online service changes its boilerplate code. Boilerplate apps can be sub-categorized into pure native applications, i.e., these apps include dex bytecode and optionally libraries written in C/C++ (see category A.1), and hybrid applications combining bytecode with HTML/JS (A.2). For HTML/JS apps, bridge code is generated to interact with the Android middleware, while HTML and Javascript is used to render the apps' user interface in a `WebView`.

2) Module-dependent Boilerplate Code: Two online services, Andromo and Appinventor, generate module-dependent boilerplate code for apps, i.e., only the boilerplate code for modules enabled/added by the app developer is stored within the apk file. Hence, apps share the same code for individual modules, but the set of enabled modules might differ. Module-dependent code requires a more complex app generation process for the online service provider, however, the generated apk is tailored to the configured modules and the chosen layout. The app semantics are not controlled by a pre-defined configuration file. While *Andromo's* app assembly is close to those of the other AppGens, *Appinventor* builds on GNU Kawa and offers, besides modules, a kind of visual programming to give the app developer the choice to implement if-then-else conditions and loops on a high-level. This gives the app developer a bit more freedom in customizing the application but has a slightly steeper learning curve for citizen developers.

B. Security Audit

Following the initial analysis of the app generation model, we then conduct a security audit on the generated boilerplate code. To this end, we follow a dynamic-static approach. We leverage dynamic testing to monitor the test apps' runtime behavior, e.g., obtain traces of the contacted domains during execution and check the possibility of eavesdropping or modifying those connections. We complement our tests with static analysis (e.g., control-flow graphs, program slicing, backtracking) to overcome the limitations of dynamic analysis, e.g., code coverage. Similar to the analysis of the boilerplate app model, we start our analysis with the self-generated apps of Section IV-A. To remove any bias from our set of self-generated apps, we cross-validate our findings with 10 randomly selected apps of the same OAG (=130 apps in total), drawn from our Google Play app corpus. Finally, as the configuration of apps can be provided dynamically by OAG's service, we extend our analysis to the OAG backend servers and the client-server communication.

Our analysis identifies two new attack vectors which are specific to the OAG generated apps, specifically reconfiguration and infrastructure attacks. We present these attacks in Section V. We then test the boilerplate apps for well-known security issues such as code injection vulnerabilities and insecure `WebViews`. The results of this analysis are presented in Section VI.

V. OAG-specific Attack Vectors

Given the inherently different generation model as compared to traditional app creation, we now describe new OAG-specific attack vectors—Application infrastructure attacks that apply to all OAGs and app reconfiguration attacks that apply to OAGs with monolithic boilerplate code only—and illustrate weaknesses that we found during our security audit.

A. Application Reconfiguration Attacks

In our set of tested OAGs (see Table 2), 5/11 services use either static or dynamically loaded config files exclusively, while the remaining six OAGs use an hybrid approach. Dynamically loaded configs have the advantage that apps can be updated on-the-fly without having to download an updated apk file from an app store. Changes are instantly pushed onto the end-users' devices on startup. However, two AppGens—Tobit Chayns and Biznessapps—do not persist their config locally and thus require a permanent Internet connection to work. As an example, Biznessapps—a paid service—uses this as a license enforcement mechanism, i.e., app functionality is disabled via the config file as soon as the app developer no longer holds a valid license. In general, we found that these configuration files carry *any* app-specific data, potentially including the entire business logic of the app and secret credentials. Hence, there is a strong incentive to carefully protect this file in terms of integrity and confidentiality.

1) Static config files: In our test set, 7/9 static config files are stored in plain and can be read and modified without effort. Only AppYet encrypts its config file, however, at the same time, the passphrase is hard-coded in the bytecode and is identical for every AppYet app. This allows us to write a simple encryption/decryption tool to read and modify the AppYet config files. The Appmachine config was the only one where we are not able to extract information. Appmachine is built on top of Mono for Android, thus, most of the app code is compiled to native code, including the classes that process the non-human-readable config file[10]. As for the integrity protection, we found that in the majority of cases standard Android APIs are used to read these files directly from assets or the raw directory. By inspecting the disassembled app code we can, however, not find any additional integrity checks or obfuscation logic for these config files. This allows, in all but one case, to trivially extract the config file from an apk. Cloning or repackaging apps becomes an easy task, since no code has to be reverse engineered, only the app features and properties within the config file–typically a `.xml` or `.json` file—have to be understood once for each application generator.

2) Dynamically loaded config files: Further, we found that eight out of eleven OAGs (A.1+A.2) load config data dynamically at runtime. All but Biznessapps and Tobit Chayns use a hybrid model of static and dynamic config loading. In five cases, the config is requested via HTTP and transmitted

[10]While we abstain from reversing this config file, we assume that, with enough effort, it should be similarly possible to extract information.

in plain without any protection. Only three OAGs load the config via SSL by default. For Mobincube, however, it is possible to downgrade the request to HTTP. Moreover, none of these AppGens uses public key pinning to prevent man-in-the-middle attacks. Similar to static configs, the complete absence of integrity checks and content encryption allows tampering with the app's business logic and data. This is fatal, since in our tests we could on-the-fly re-configure app modules (enabling, disabling, modification), disable advertisements (for free AppGens), compromise app data (localized strings, about information) or modify the application's sync URL (AppYet). Only few settings cannot be compromised when app content is no longer retrievable from the server due to license expiration (e.g. we found that 5,137 out of 27,130 Biznessapps in our app set already expired).

Having the full control over the app's data and business logic as a network attacker allows a range of different attacks to be mounted with moderate effort. Targeted phishing attacks may allow stealing user data/credentials. On-the-fly replacement of API keys for advertisement may allow an attacker to steal ad revenue. Or an attacker may simply try to deface the application which in turn affects the reputation of the app developer.

B. Application Infrastructure Attacks

A large fraction of app generators bind their customers to their web services, e.g., for user management or license validation. Particularly, hybrid apps that make use of web technology, like `WebViews`, to deliver the app's logic, bind the generated app to the generator service's web infrastructure. For instance, in our set of AppGens, Seattle Cloud, Biznessapps, and Como bind both their clients and end-users to their service's infrastructure, e.g., by managing the client's user data or by delivering the client app's content to a generated boilerplate app. Thus, the attack surface of the generated app inherently increases beyond the app and its network connection to the web service backend. Consequently, when considering that a single service's infrastructure can serve many hundreds or thousands of generated apps, it is paramount that the app generator service not only follows best practices, such as correctly verifying certificates, but also that the service's infrastructure maintains highest security standards for their web services. Of particular interest is here, whether such services are resilient to remote attackers, i.e., against state-of-the-art attacks against SSL/TLS [14], [33], [5], [3], [10], [1] that affect content delivery either to generated apps or app data to the service.

We extract the domains of the different services' backend servers from the generated apps and use available online analysis sites (e.g., Qualys SSL Labs[11]) to check the SSL/TLS security of respective backend servers. This particularly includes checks for trusted and valid certificates, support for outdated and weak ciphers and protocols, resilience against recent SSL/TLS vulnerabilities, usage of weak keys, and checks whether any domain contacted by default by generated apps is known to distribute malware (e.g., using Google's SafeBrowsing[12] service).

The results of analyzing the communication with the server backend are alarming. First of all, only Tobit Chayns and Biznessapps use encryption consistently for any communication with the backend, while Apps Geyser completely abstains from secure communication (i.e., HTTP only) and the other services secure their communication only partially. For instance, both Seattle Cloud and Como send sensitive data from user input forms like a login form completely in plain text. Moreover, only three services use a valid and trusted server certificate, while, for instance, Appy Pie uses a self-signed server certificate and Mobincube uses a certificate that expired seven years ago. From a cryptographic point of view, all of the services are running an outdated version of SSL libraries that are prone to one or more recent attacks such as POODLE [33], BEAST [14], LOGJAM [3], or FREAK [10]. Mobincube's server was even vulnerable against all of the tested SSL vulnerabilities.

Data leakage and Privacy Violations: OAGs typically offer modules to connect to third-party services, like Google Maps or social media platforms like Facebook and Twitter. These modules include code to connect to these services via service-specific APIs. Using these APIs typically requires an API key (and secret). Since OAGs do not create third-party accounts on behalf of the application developer, those AppGens provide their own API key (and secret) to any app created by its service. Some keys require a fee for business/volume usage, like Google Maps keys, hence it is of interest if the OAG protects these keys from (easy) eavesdropping. The combination of leaked Twitter key and secret, e.g., hard-coded in boilerplate code of Biznessapps, allows to send arbitrary authorized requests and in particular to tamper with the account that is shared across all apps generated with this AppGens. Although those keys are application-only authentication keys with limited access rights, the Twitter developer documentation recommends that these *"should be considered as sensitive as passwords, and must not be shared or distributed to untrusted parties."*[13] We found keys and secrets for various different third-party providers unobfuscated in config files, hardcoded in boilerplate code, in the AndroidManifest file, and even in the strings.xml. All identified keys were exactly the same across all analyzed apps, underlining the security impact of boilerplate apps. We could not find a single attempt to obfuscate or protect these keys/secrets.

Besides paid-only services, a large number of AppGens provide their service for free. Similar to normal app development they use different approaches to monetize their apps, such as advertisement and/or tracking. Since the literature has shown that such third-party libraries often leak sensitive user data [27], [43], we especially checked outgoing app traffic and compared our findings with the privacy policies

[11]https://www.ssllabs.com/ssltest/

[12]https://developers.google.com/safe-browsing/

[13]https://dev.twitter.com/oauth/application-only

provided by the online service. The results suggest that none of the web domains that the tested apps contacted during our analyses was known for distributing malware. However, four application generators (Como, Mobincube, Biznessapps, and Appy Pie) clearly exhibited questionable tracking behavior. Apps generated with Mobincube sent more than 250 tracking requests within the first minute of execution. In addition, Mobincube includes the third-party library *BeaconsInSpace* to perform user tracking via Bluetooth beacons without the user noticing it. Although *BeaconsInSpace* strongly recommends updating the privacy policy of apps using their library, we could not find any information in Mobincube's terms and conditions. Appy Pie apps contacted Google Analytics, Appy Pie's backend, and Facebook for tracking. Apps generated by Como automatically registered with different tracking websites, including Google Analytics, Como-owned servers, and others. Biznessapps sends device identifier and location to their backend servers on app launch. While such extensive tracking behavior is already questionable for the free services of Appy Pie and Mobincube, one would certainly not expect this for paid services like Como and Biznessapps.

VI. Evaluating Known Security Issues

In addition to the specific attack vectors of online services discussed in Section IV, we further analyzed the generated apps' boilerplate code for violations of security best practices on Android [23] and vulnerabilities identified by prior research on Android application security and privacy [2], e.g., testing the apps' device-local attack surfaces, such as unprotected components. Again, we used our set of self-generated apps as well as the set of generated apps from Google Play for cross-validation. Whenever feasible we run static tests against the entire set of generated apps from Google Play. Table 3 provides an overview of the security analysis results, which we discuss in more detail in the remainder of this section. We distinguish apps in vulnerable, non vulnerable, and risky. We say that an app is vulnerable (●) when we successfully exploit the flaw. We say that an app is risky (◐) when an exploitation scenario exists, but we did not (or could not) reproduce it. Otherwise, we say that the app is not vulnerable (○).

A. Best-practice Permission Usage (P1–P3)

Apps may request more permissions than actually needed [37], [4], which unnecessarily increases the privileges of third-party code, such as ad libs, that have been shown to actively exploit such inherited privileges and to exhibit questionable privacy-violating behavior [26], [11], [40], [43]. The Android security best practices also explicitly recommend developers to request as few permissions as possible to conform to the principle of least privilege.

Moreover, Android apps are by design allowed to engage in inter-component communication (ICC [16]). However, apps that (unintentionally) export their components for access by other applications, but with no or only insufficient protection, may leak privacy-sensitive data or security-sensitive methods to unprivileged attacker apps [45], [31], [25]—a scenario also warned about in the security best practices. In addition, for certain components, such as `Activities` or `BroadcastReceivers`, the app developer has only very limited means to identify or authorize the sender app [9]. This opens the opportunity for `Intent` spoofing attacks [12], [35] and confused deputy attacks [37], where a vulnerable component acts on behalf of an ICC message from an attacker app.

Security analysis: To detect whether an application is overprivileged, we identify the permission-protected API calls in the application (using PScout's [4] and Axplorer's [8] permission maps) and derive from those the set of required permissions. We complement this list with `ContentProvider` and `Intent` permissions necessary for the app to run properly. We then compare the resulting set with the set of actually requested permissions in the application's manifest. If the latter one is a strict superset of the former one, we call the application overprivileged.

We further check applications for explicitly exported `Activity`, `Service`, and `ContentProvider` components or potentially accidentally exported components (e.g., by setting an intent-filter without manually setting flag `android:exported` to `false`). If any of those exported components is not protected with a permission with at least signature protection level, we consider this app as exposing an unprotected component. To also detect receivers potentially prone to `Intent` spoofing attacks, we conduct the same analysis as above for `BroadcastReceivers` , but additionally considering receivers registered dynamically at runtime via the app's context.

Results: All AppGens that generate monolithic boilerplate code create over-privileged apps by design (P1 in Table 3). As long as an app developer chooses a subset of modules (from the set of 12–128 modules across AppGens), the resulting app has, with a high percentage, more permissions than actually necessary. For instance, the simple *Hello World* app (App1) has between 7–21 permissions for monolithic boilerplate apps, including camera access, write/read contacts, audio recording, Bluetooth admin, and location access. At the same time App1 of Andromo and Appinventor—that generate module-dependent code—request only a single permission and three permissions, respectively.

Application generators do not satisfactorily protect generated apps' components from illicit access (P2). Except for Andromo, AppInventor, and Biznessapps, all tested generators failed to protect one or more components that we identified through manual analysis (e.g., using their package and class name) to be intended as internally-accessible only. This can potentially lead to severe implications for the end-users' or app generator clients' privacy. For example, apps generated with the Seattle Cloud or Mobincube service expose unprotected components for an `InternalFileContentProvider` and `AppContentProvider`, respectively, through which an attacker can read all files to which the app's UID has access, including internal files like databases, private shared preferences, or in case of Seattle Cloud the

Table 3: Categorization of considered attack vectors against generated apps in the Android ecosystem.

	Free service							Paid service					
Attack vector	Andromo	Apps Geyser	Appinventor	AppYet	Mobincube	Appy Pie	Tobit Chayns	Appmachine	Biznessapps	Seattle Cloud	Como	Apps Builder	GoodBarber
P1. Overprivileged Apps	○	●	○	●	●	●	●	●	●	●	●	●	●
P2. Unprotected Components	○	●	○	●	●	●	●	○	○	●	●	●	●
P3. Intent Spoofing and Confused Deputies	○	◐	◐	◐	◐	◐	◐	◐	○	○	◐	○	○
P4. Cryptographic API Misuse	○	●	○	●	○	●	●	○	●	●	○	○	●
P5. SSL/TLS Verification Errors	○	●	●	●	○	●	○	○	●	○	○	●	○
P6. Fracking Attacks	○	●	○	●	●	●	●	○	○	○	○	●	○
P7. Origin Crossing	●	●	○	●	●	●	○	○	○	●	●	●	●
P8. Code Injection (native / WebView)	○/●	○/●	○/●	○/●	●/●	○/●	○/●	●/○	○/●	○/○	○/●	○/○	○/○

● = vulnerable; ○ = not vulnerable ◐ = risky

asset file `app.xml` [39] in which Seattle Cloud apps store, among other things, the user accounts and passwords for logging into the app. We statically searched for this vulnerable `InternalFileContentProvider` in 60,314 apps from Seattle Cloud and found it in 100% of those apps. We also discovered the vulnerable `AppContentProvider` in 7,953 (98.5%) of analyzed 8,074 Mobincube apps in our test set.[14] This underlines the high security impact of application generators that are using vulnerable boilerplate code. We also discovered that unprotected components are not always within the package domain of the application generator service, but can sometimes be traced back to included third party packages, e.g., by Radius Networks or ZXing, and it remains to be determined whether the app generator failed to correctly protect such third party components or whether the design of those components prohibits a more secure integration.

A particularly worrisome aspect are unprotected `BroadcastReceivers` and `Activity` components that might accept spoofed `Intents` from untrusted senders and act upon such received data (P3). Eight of the tested application generators produce code that is prone to such `Intent` spoofing attacks. For example, Appinventor apps react to a fake SMS notification, Mobincube apps can be triggered to interact with the WiFi service, and Appmachine-generated apps have a remote command receiver exposed that forwards received `Intents` unfiltered to a native command for execution.

B. Insecure Cryptographic API Usage (P4)

App developers might use cryptographic APIs to secure their data on the end-user device. However, the security that cryptographic APIs can actually deliver, strongly depends on the correct usage of the cryptographic building blocks (e.g., adequate block cipher modes, correct salting, etc.). App developers frequently make mistakes when using those primitives [15], such as using ECB mode for encryption, using a non-random IV for CBC encryption, or using constant salts/seeds (see also [24]). The Android security best practices documentation picked up some of these recommendations and, for instance, advises using `SecureRandom` instead of `Random`, initializing cryptographic keys with `KeyGenerator`, or using the `Cipher` class for encryption with AES and RSA.

Security analysis: To detect misuse of cryptographic APIs, we re-apply the analysis methodology presented in [15] by leveraging R-Droid [7] to search for usage of cryptographic API methods and then track their parameters. To this end, we focused on APIs in Android's `javax.crypto` package. For symmetric encryption, we analyzed the usage of the `Cipher.getInstance` parameter, where developers are expected to specify a symmetric encryption algorithm, mode and padding—e.g.`"AES/CBC/PKCS5Padding"`. Similar to related work and security best practices, we rated the use of the ECB mode of operation as insecure and rated, additionally, the following outdated algorithms as insecure: (3)DES, IDEA, RC4, and Blowfish. Moreover, the use of non-random IVs for CBC mode or in general the use of a static encryption key is rated as insecure.

To use hash functions, app developers are recommended to use the `MessageDigest` class, where the hash function's algorithm can be chosen via a string parameter—-e.g. `"SHA-256"`. App developers can include message authentication codes (MACs) into their apps by using the `Mac.getInstance` API call. Again, MAC algorithms are expected to be passed as a string parameter—e.g. `"HmacSHA256"`. For hash functions and MACs, we consider the use of MD2, MD4, MD5, SHA0, SHA1 and Ripemd128 as insecure.

[14]We believe that the 121 apps without this provider are older, discontinued apps, built or last updated prior to the introduction of the `AppContentProvider`.

Regarding sources of randomness, we consider the usage of `Random` instead of `SecureRandom` as insecure; however, usage of `SecureRandom` is also rated as insecure when a static seed is used.

Results: Seven out of thirteen tested app generators failed to use Android's cryptographic APIs securely. A rather pathological weakness seems to be using an insecure random number generator. We discovered in five of seven vulnerable AppGens that `Random` values are generated with a static initialization vector, and most frequently used when generating symmetric encryption keys or initialization vectors for CBC-mode symmetric encryption, rendering the generated apps prone to cryptanalytic attacks. For instance, Biznessapps creates predictable session identifiers by concatenating the output of `Random` with the current system time. Additionally, three of those generators relied on the insecure ECB mode for encryption.

C. Insecure WebViews (P5–P8)

App developers frequently fail in validating SSL/TLS certificates correctly [17], [18], [41], [21], making their apps vulnerable against man-in-the-middle attacks. The Android security best practices have a dedicated, extensive section on security with HTTPS and SSL, explaining the pitfalls and their solutions in implementing a secure SSL connection and even providing tools for testing the SSL configurations of apps.

Moreover, web utility classes, such as `WebView`, allow app developers to combine the features of web apps (e.g., platform independent languages) with those of native Android apps (e.g., rich access to the device's resources). However, the access controls that govern web code (e.g., JavaScript from different web domains) and local code (i.e., Android native code) are not properly composed: the bridge code between web code and local code can interact with the Android system with the same access rights as its native host application, but does not enforce the same origin policy on calls from the web code to the bridge functions, thus opening this dangerous bridge interface to all loaded web code. The security best practices suggest to enable JavaScript only if really necessary, to prevent cross-site scripting. In addition, it warns that bridges between web code and local code should be used only for websites from which all input is trusted, as it allows fracking attacks [22], [13], [32].

This lack of origin-based protection of the JavaScript bridge also opens the door for various origin-crossing attacks. A particularly concerning cross-origin attack is based on the *scheme* mechanism. Schemes allow apps on the device to be invoked through URLs whose scheme part equals the scheme registered by the app. However, any app can register for arbitrary schemes. In combination with `WebViews` this allows for unauthorized cross-origin attacks [44], when the user clicks on a malicious link in the `WebView`, which refers to a local application that might act on the parameters given by the URL.

Lastly, Android's programming model allows app developers to dynamically load code from different sources, such as public application packages, dex files, or the web via `WebViews`. However, if the application does not correctly verify the integrity and authenticity of loaded code, the app becomes vulnerable to be compromised by an attacker that can modify the loaded (or injected) code. This attack has to be differentiated between platform native code (i.e., dex or C/C++) [36] and web code (HTML, JavaScript) [29], [34]. In the former attack, the attacker is able to modify the loaded code, e.g., dex bytecode on the local file system or inject malicious code into the download stream of such loaded code. In the latter attack, the attacker achieves execution of custom JavaScript code within a trusted website in a `WebView` or manages to navigate a `WebView` away from a benign, safe website to an attacker-controlled website. As a result, the attacker can control the web resources within the compromised `WebView` (e.g., to exfiltrate credentials) and further leverage the `WebView's` host app's privileges to the extent they are exposed through bridge code between host app and `WebView` instance. Android's security best practices strongly discourages app developers from dynamically loading code from outside of their application for the aforementioned reasons and, again, recommends only loading web code from trusted websites.

Security analysis: We tracked the parameters of the `HTTPUrlConnection` and `HTTPSUrlConnection` classes respectively. We rated plain HTTP URLs as insecure and HTTPS URLs as secure. Moreover, we investigate the use of non-default `TrustManager`, `SSLSocketFactory`, or `HostnameVerifier` implementations with permissive verification strategies, which we deem as insecure. Additionally, for `WebViews` we search for custom `SSLErrorHandler` implementations in the `WebViewClient` with a permissive or insecure error handling, which we deem as insecure.

We classify `WebViews` that enable JavaScript as insecure when the bridge functions expose security- and privacy-sensitive functionality, and as secure if those options are disabled or non-critical functionality is exposed. Since apps with target SDK 19 or higher reject mixed content by default, we consider those secure, unless developers used the `setMixedContentMode` method with the `MIXED_CONTENT_ALWAYS_ALLOW` parameter to deviate from the default; in this case, we consider the app's behaviour insecure according to the previously described metrics.

We further investigate the presence and implementation of the `shouldOverrideUrlLoading()` method of `WebViewClient`. We consider the app prone to origin crossing if the `WebViewClient` is missing, i.e., the opening of the URL is deferred to some installed app registered for the URL's scheme. Additionally, we consider the app prone to this attack if a `WebViewClient` is present, but it's implementation of the `shouldOverrideUrlLoading()` defers the URL loading to apps via sending `Intents` with the URL as parameter.

To determine whether apps load external code, we check for API calls to `DexClassLoader` and subclasses that load code over (insecure) network connections, which we consider insecure behavior. Apps that use the `URLClassLoader` with HTTP URLs are of particular danger. In case of `WebViews`, we consider code injection possible if either the `WebView` uses insecure Internet connections or if the `WebViewClient` is present but does not override the `shouldOverrideUrlLoading()` function (or implements a permissive URL overriding that opens attacker provided links in the `WebView`). In those cases, an attacker can potentially lure the `WebView` to an attacker-controlled website.

Results: Six of the tested application generators rely on web technology, i.e., `WebViews`, to display their client's content. Thus, for those app generators it is paramount to prevent untrusted content from being loaded into the `WebView` or securely sandboxing the `WebView's` interaction with the Android system and other installed apps. Out of the thirteen tested app generators, six failed to correctly handle SSL certificate verification errors and accept any self-signed certificate (P5), which also eases the task of an attacker to inject code into a `WebView` by manipulating the download stream (P8). Apps Geyser catches verification errors, defers the decision about the trustworthiness of the certificate, however, to the end-user, who has repeatedly been shown in the literature to be unable to make such trust decisions correctly [38]. At the same time, we found that none of the investigated online services implemented measures to enhance the SSL security, e.g., by pinning the certificate. Similarly, we discovered that only about half of the tested apps correctly limited the scope of navigation inside the `WebViews` or enforce a same origin policy on the loaded web content, thus opening the possibility to navigate the `WebView` to untrusted web resources that deliver malicious code with full access to the JavaScript bridge to native platform code. This is particularly worrisome when considering that almost all of the tested generators with `WebViews` expose quite substantial JavaScript interfaces and hence enable fracking attacks (P6). For instance, Apps Geyser exposes over 90 JavaScript bridge functions, providing an attacker with all tools needed, such as camera and microphone access, storage access, or `Intent` sending. Mobincube and Appy Pie even exceed this number by exposing more than 100 functions, including methods such as `createCalendarEvent`, `getCurrentPosition`, `getGalleryImage`, `makeCall`, `sendSMS`, `takeCameraPicture`, `uploadMultipleFiles`, or `processHTML`. Further noticeable is that several of the tested generators convert the loading of a custom URL in the `WebView` (e.g., through a crafted link provided by the attacker) to an `Intent` that will be sent by the generated app to other installed apps. This opens the possibility for cross-origin attacks (P7).

VII. Discussion

We now interpret the key findings of our online application generator study and propose some short and long-term actionable items to improve the current status quo.

A. Citizen App Developers on the Rise

The first key finding of our study is that citizen developers are indeed a growing phenomenon in the mobile application development ecosystem. As AppGens promise to decrease the app's development costs, more and more organizations are interested in this new development paradigm. Financial reports, already in 2011, expected citizen developers to build at least 25% of new business applications by 2014 [20], with an estimated a total revenues of \$1.7 Billion in 2015 and an expected growth of +50% per year [19]. Our analysis is the first to confirm the growth forecast in terms of market penetration for the mobile ecosystem, showing that at least 11% of free apps in Google Play (250K apps) are already generated by *Online Services*.

B. Pitfalls of the "One Size Fits All" AppGens' Strategy

Online Services provide simple means of creating apps without requiring any knowledge about programming or mobile operating systems. This is achieved by abstracting the implementation task to some kind of drag-and-drop assembly of predefined modules and by limiting the degree of freedom of app customization. Such a "One Size Fits All" strategy led to a new paradigm in app generation, distributing an apk file with monolithic or module-dependent boilerplate code that is statically or dynamically configured with an app-specific config file. While this provides a convenient way to generate and distribute applications for a large number of clients, from a security perspective, this creates new points of failures, that, if not considered carefully, might compromise end-user security or even online service security.

The results in Section IV show that the majority of OAGs that base their business model on monolithic boilerplate apps fail to properly protect config files from tampering and eavesdropping. Only 2 out 8 OAGs in Table 2 correctly use HTTPS to retrieve config files. Moreover, we found that none of the services applied certificate pinning to prevent man-in-the-middle attacks. Similarly, only a single service properly protected its statically included config. However, none of the services checked the integrity of the config file during app launch. This opens the door for many attacks such as reconfiguration attacks, ad revenue theft (through replacing API keys), and, in general, changing arbitrary app-specific data.

Boilerplate apps that use HTML/JS for layouting (see category A2 in Table 2) are additionally prone to code injection and fracking attacks (P6–P8 in Table 3). This is caused by the web-to-app bridge these services use to access the Android API. Due to the boilerplate app pattern, these bridges expose more functionality than typically necessary and/or are not properly protected from being misused.

Following the principle of least privilege, Andromo and Appinventor (see category B) generate targeted boilerplate code based on the modules selected by the app developer. Since their code generation model follows the traditional app development, they are not prone to OAG-specific security problems such reconfiguration attacks. However, the trade-off is an additional code generation effort, when any combination of pre-defined modules must be flawlessly composable. This is why Andromo, with only 19 available modules, is at the lower end in terms of available modules, while Appinventor with its community support offers notable 59 modules.

C. Amplification of Security Issues

The increasing use of online services has shifted the duty of generating secure code from app developers to the generator service. Users rarely have options to customize or change their application beyond the ones provided by the service. As a consequence, users have to fully trust the service to generate non-vulnerable code and to not include hidden or non-obvious user tracking or data leakage. Particularly worrisome examples include the paid service Como, that performs heavy user tracking although its privacy policy explicitly emphasizes the importance of the security of the users' personal information and Mobincube that silently tracks users via BLE beacons without explicitly stating this in its terms and conditions.

The results of our security evaluation in Section VI suggest that OAG-generated apps do hardly adhere to security best practices and exhibit common app vulnerabilities that have been identified by prior research. Although these findings are in line with coding practices of traditional developers, the worrisome aspect is the amplification effect of online services, putting millions of users and their private data at risk. Another key insight is, as opposed to traditional apps, vulnerabilities in unused boilerplate code can still be exploited when a network attacker is able to compromise the application config and re-configure or activate app modules with known security issues or when `ContentProviders` can be queried to retrieve internal data (see Section VI-A).

We conclude that in the current online service ecosystem the level of security does not depend on whether it is a free or paid service, but rather on the underlying app generation model. For the two module-dependent code generators Andromo and Appinventor we found the least security issues. Particularly for Appinventor this is unsurprising, since it is open-source and does not follow commercial interests. The boilerplate model is not generally insecure, however, from a security perspective, server communication and config protection require a more careful design. This could include certificate pinning for dynamically retrieved configs, obfuscation or encryption of static configs, and integrity checks to prevent unauthorized tampering.

D. Missed Opportunity for a Large-Scale Security Impact

A patch to the current situation is to inform online services about the discovered security issues to allow them to fix their code generation. We are currently in the process of a responsible disclosure to allow the respective service providers to fix the security flaws. However, while this is a short-term mitigation, it does not address the root cause of these issues, thus not producing more desirable long-lasting effects. In our opinion, OAG services need a thorough investigation from the research community in the way AppGens are built. This investigation requires a solid understanding of the underlying technique in use. Other areas of research from which lessons can be learned or transferred are tailored software stacks. Prior works have shown that the attack surface can be considerably reduced by compile-time [30] and run-time configurations [42].

VIII. Conclusion

In this paper we present the first classification of commonly used online services for Android based on various characteristics and quantify the market penetration of these AppGens based on a corpus of 2,291,898 free Android apps from Google Play to discover that at least 11.1% of these apps were created using online services. Based on a systematic analysis of the new boilerplate app generation model, we show that online services fall short in protecting against reconfiguration attacks and running a secure infrastructure. A subsequent security audit of the generated boilerplate code reveals that OAGs make the same security mistakes as traditional app developers. But in contrast, they carry the sole responsibility of generating secure and privacy-preserving code. Due to their amplification effect—a single error by an OAG potentially affects thousands of generated apps (250K apps in our data set)—we conclude that *Online Services* currently have a negative impact on the security of the overall app ecosystem. But, at the same time, these services are in the unique position to turn these negative aspects into positive ones through spending more effort into securing their application model and infrastructure from which ultimately millions of users benefit.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345, 16KIS0656), the projects SmartPriv (FKZ: 16KIS0377K) and CAMRICS (FKZ: 16KIS0656) and the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

[1] "CVE-2016-2107: Padding-oracle attack against AES CBC," Jan. 2016.

[2] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *Proc. 37th IEEE Symposium on Security and Privacy (SP '16)*. IEEE, 2016.

[3] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[5] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, , and Y. Shavitt, "DROWN: Breaking TLS using SSLv2," Mar. 2016, CVE-2016-0800.

[6] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and its Security Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 2016.

[7] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, "R-Droid: Leveraging Android App Analysis with Static Slice Optimization," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, 2016.

[8] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis," in *Proc. 25th USENIX Security Symposium (SEC'16)*. USENIX Association, 2016.

[9] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric ipc provenance on android," in *Proc. 30th Annual Computer Security Applications Conference (ACSAC'14)*. ACM, 2014.

[10] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *Proc. 36th IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 2015.

[11] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," in *MoST'13*. IEEE, 2013.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[13] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *Proc. Information Security Applications*. Springer-Verlag, 2014.

[14] T. Duong and J. Rizzo, "Here come the ⊕ ninjas," May 2011.

[15] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[16] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," in *Proceedings of the IEEE International Conference on Security & Privacy*, 2009, pp. 50–57.

[17] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: an analysis of android ssl (in)security," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61.

[18] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[19] A. Fisher, "How companies are developing more apps with fewer developers," http://fortune.com/2016/08/30/quickbase-coding-apps-developers/, Aug. 2016, last visited: 11/29/2017.

[20] Gartner, "Gartner says citizen developers will build at least 25 percent of new business applications by 2014," http://www.gartner.com/newsroom/id/1744514, Jul. 2011, last visited: 11/29/2017.

[21] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[22] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks ," in *2014 Network and Distributed System Security (NDSS '14)*, San Diego, February 2014.

[23] Google, "Android Developers: Security Tips," https://developer.android.com/training/articles/security-tips.html, last visited: 11/29/2017.

[24] Google, "Android developers blog: Using cryptography to store credentials safely," http://android-developers.blogspot.de/2013/02/using-cryptography-to-store-credentials.html, 2013, last visited: 11/29/2017.

[25] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[26] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in *Proc. 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. ACM, 2012.

[27] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*. ACM, 2012.

[28] D. Hinchcliffe, "The advent of the citizen developer," http://www.zdnet.com/article/the-advent-of-the-citizen-developer/, Apr. 2016, last visited: 11/29/2017.

[29] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[30] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time OS kernel tailoring," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[31] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[32] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.

[33] B. Möller, T. Duong, and K. Kotowicz, "This POODLE Bites: Exploiting the SSL 3.0 Fallback (Security Advisory)," Sep. 2014, CVE-2014-3566.

[34] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Proc. 2015 Mobile Security Technologies Workshop (MoST'15)*. IEEE, 2015.

[35] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[36] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[37] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conference on Computer and Communication Security (CCS'11)*. ACM, 2011.

[38] A. Porter Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proc. 8th Symposium on Usable Privacy and Security (SOUPS '12)*. ACM, 2012.

[39] Seattle Cloud, "Login page type tutorial," http://seattleclouds.com/login-page-type-tutorial, 2016, last visited: 11/29/2017.

[40] S. Son, G. Daehyeok, K. Kaist, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2015.

[41] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[42] K. Stengel, F. Schmaus, and R. Kapitza, "Esseos: Haskell-based tailored services for the cloud," in *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware*, ser. ARM '13. ACM, 2013.

[43] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[44] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[45] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in Android applications," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.