POSTER: When Laziness Snaps Back The Impact of Code Generators on App (In)Security

Yasemin Acar[1], Michael Backes[1,2],
Sascha Fahl[1], Christian Stransky[1]
CISPA, Saarland University[1]
MPI-SWS[2]
{acar,backes,fahl,stransky}@cs.uni-saarland.de

ABSTRACT
The Android ecosystem suffers from many vulnerabilities, which, due to its popularity, affect millions of users. Research has identified many of these problems and, among other approaches, suggested to better educate developers and nudge them to make safe decisions. Developers often employ third party libraries, or even application generators or HTML-to-Android-app frameworks to lighten their coding workload. We identified the prevalence of broken TLS as well as dangerous system permission requests. We also analyzed the security level of access control employed when using activities, services, receivers and providers. Our analysis highlights differences in the (in)security of custom code vs. generated code, and we suggest to specifically target the security education of code providers to make Android more secure.

1. INTRODUCTION
Due to its high market share and open source character, the Android ecosystem is a highly researched area: Over the last six years, security problems of the Android OS and third party applications developed for the Android ecosystem have been intensely researched by the security community [3, 2, 1, 4, 5]. Previous research unveiled that many third party app developers do not follow least privilege by over-requesting permissions [4], that developers have problems to securely employ Android's TLS [3, 5] and crypto API [2] or that developers have problems to securely make use of Android's inter process communication features [1]. A huge number of apps and therefore users are affected by this misuse of the Android API. As of yet, only insufficient countermeasures have been deployed to tackle these problems. Research often states that developers need better education and support, that programming interfaces need to be simplified and that the Android operating system needs to me modified. Implicitly, it is assumed that the number of developers that require support roughly scales with the number of apps affected by security issues. However, as the appification paradigm of the last years has generated the need for (un)skilled (web) developers to transfer their services to Android, the need to quickly produce Android code is often supported by the use of third party libraries, professional application generators or easy-to-use HTML-to-native-code application frameworks. While this trend towards mass production of Android apps is a security threat in itself – as one broken framework, library or generator can break the security of a huge number of apps and affect millions of users at once - it also offers a chance: A move to more secure paradigms by only a few service providers can go a long way to make the whole Android ecosystem more secure, as shown by TLS certificate validation (cf. Section 4.1). In this work, we statically analyze a corpus of 1,005,894 Android apps: We investigate which apps were written by distinct developers versus which were produced by frameworks/generators and correlate these with the occurrence of selected security threats in the apps' code. We find that generated apps are more likely to violate security best practices as well as to break TLS certificate validation. However, we did find app generators for which TLS certificate validation was correct for every app. We understand this as an oppurtunity to fix huge numbers of insecure apps by contacting and educating only a few app generator providers.

2. BACKGROUND
2.1 Android Security Best Practices
The official Android developer API documentation [1] provides a list of security best practices, including suggestions like using HTTPS over HTTP whenever possible, not requesting more permission than strictly necessary, not storing sensitive data on external storage, sanitizing user input, making use of secure cryptographic features only, appropriately protecting inter component communication, not dynamically loading code from outside an APK file and using the Android SDK whenever possible instead of the NDK and using native code.

2.2 Application Generators
Instead of writing a mobile application from scratch, developers can choose to use a service provider that allows to automatically generate (large parts) of a web or native app. Those app generators service providers oftentimes are websites, standalone tools or programming frameworks that allow developers to produce new apps with minimal effort.

[1] `http://developer.android.com`

It is common to write HTML/JavaScript code and then let the generator produce apps for multiple platforms such as Android, iOS and Windows Mobile. A popular framework that works like this is Apache Cordova[2]. The most popular application generators we could identify in our set of Android applications are: andromo [3], Tobit[4] and conduit[5].

2.3 Insecure API Usages

Previous work demonstrated that many developers fail to properly implement security related Android APIs. Fahl et al. [3] statically analyzed Android apps' code and found that many of the apps in their dataset used broken TLS certificate verification. Egele et al. [2] confirmed similar findings for Android's cryptographic APIs. Porter-Felt et al. showed that developers often over-request permissions [4]. Poeplau et al. [5] showed that loading dynamic code is often implemented in an insecure way. Chin et al.[1] found that Android app developers struggle to properly protect inter component communication.

3. METHODOLOGY

We downloaded 1,005,894 free apps from Google Play in November 2014 and used androguard [6] to decompile the whole app corpus; we generated callgraphs for all apps and collected further information we used for the later analyses:

- *Callgraphs* allow us to investigate which Android APIs and third party libraries are called within an app.
- *Manifest.xml* files include information such as the number of requested permissions and information about content providers, intents and services.
- *Packagenames* uniquely identify Android apps in Google Play (e. g. comkatanafacebook identifies the Facebook app) and prevent name collisions of apps on an Android device.

3.1 App Classification

We used the above criteria to decide whether an app was built using one of the app generators (cf. Section 2.2), made use of one or more of the popular third party libraries or if the apps were originally written by an app developer.

Packagenames: By analyzing packagename patterns in our corpus, we could identify 28 app generator frameworks, e. g. all apps that were generated using the conduit [7] app generator have the packagename pattern `com.conduit.*`. 38,467 apps could be assigned to an app generator based on their packagenames. Table 1 illustrates the most popular packagename patterns.

Callgraph Data: Using specific namespaces in apps' call graphs, we could identify 513,793 apps that used third party code: Using callgraph data, we found 11 app generators corresponding to 50241 apps and 51 libraries/frameworks/SDKs corresponding to 463,552 apps. Table 2 shows the most popular namespaces corresponding to third party libraries. Identification using the developer ID is also possible. Table ~~3 gives an overview of the mo~~st prevalent developer IDs.

[2]Cf. `https://cordova.apache.org/`

[3]Cf. `http://www.andromo.com/`

[4]Cf. `http://en.tobit.com/`

[5]Cf. `http://www.conduit.com/`

[6]Cf. `https://github.com/androguard/androguard`

[7]`http://www.conduit.com/`

Packagename	App Count
com.Tobit.*	13,501
{com|net}.andromo.dev*	11,450
com.conduit.*	6,875
com.appsbar.*	1,122
com.phonegap.*	1,050
com.crowdcompass.*	965
com.mobincube.android.*	902
{ru|com}.loyaltyplant.partner.*	468
com.quickmobile.*	452
com.appypie.*	392

Table 1: The 10 most popular packagename patterns.

Library Prefix	App Count
com.google	470,648
com.facebook	118,864
com.actionbarsherlock	77,377
org.slf4j	74,809
com.millennialmedia	61,465
com.inmobi	58,897
com.squareup	51,708
twitter4j	48,860
com.flurry	45,602
com.viewpagerindicator	40,594

Table 2: The 10 most popular third party code namespaces we found in our app corpus.

Developer ID	App Count
Tobit Software	13,501
Brainpub for Theme	1,377
Subsplash Consulting	1,278
Lisbon Labs	1,245
GPSmyCity.com, Inc.	876
Camp Mobile for dodol theme	825
Shopgate GmbH	813
iConnect	786
Skoolbag	750
Thanakorn	741

Table 3: The 10 most popular developer IDs.

3.2 Against Android Security Best Practices

We measured the compliance to several Android security best practices as recommended by Google (cf. Section 2.1):

3.2.1 TLS Certificate Validation

For apps that used TLS in an effort to establish secure network connection, we used the MalloDroid tool[8] to analyze whether correct X.509 certificate verification was implemented. We classify apps as breaking TLS if they

- Implement a broken `TrustManager` for TLS connections.
- Overwrite the `OnReceivedSSLError` method for a `WebView`.
- Use a `HostnameVerifier` that does not properly check a certificate's common name.

[8]`https://github.com/sfahl/mallodroid`

3.2.2 *Manifest File*

We analyzed the Manifest files of all apps to see if they follow best practices. We identified apps that

- Request dangerous system permissions, e.g. `android.permission.SEND_SMS`.
- Define dangerous permissions, e.g. own permission with `protectionLevel=`"dangerous".
- Request SMS or external storage permissions.
- Export unprotected activities, receivers, services or providers.

4. RESULTS

Overall, we analyzed 1,005,894 free Android apps from Google Play and checked the security properties described in Section 3.2.

4.1 TLS Certificate Verification

We found 167,724 apps that implemented insecure X.509 certificate verification via a custom `TrustManager`. Of the 167,724 apps that implemented a broken TrustManager, 37,322 apps were built by app generators. Hence, 61.07% of all app generator apps include broken certificate verification, while 15.17% of all custom apps implement broken certificate verification. Table 4 shows the prevalence of broken TLS certificate validation for popular app generators.

App Generator	Broken Certificate Verification
Tobit Chayns	99.99%
Como	99.87%
SeattleClouds	98.74%
Appy Pie	98.35%
Good Barber	97.88%
Quickmobile	69.15%
CrowdCompass	51.02%
Andromo	13.83%
Apps Bar	0.00%
Mobincube	0.00%

Table 4: Implementation of broken certificate verification in apps for the most popular app generators.

These results illustrate that broken TLS certificate verification is more prevalent in apps built by app generators. However, while some app generators include broken certificate verification into almost all of their apps, other app generators – although offering HTTPS connections – do not include broken certificate verification code at all. Hence, while apps produced by app generators are more likely to be vulnerable to Man-In-The-Middle attacks, app generators have the chance to protect all of their clients' HTTPS connections properly in one go.

4.2 Manifest File

We analyzed the content of all Manifest files (cf. Section 3.2.2) and present the results in Table 5. Almost all generated apps request dangerous permissions (99.34%), which could indicate over-requesting of permissions. Across all apps, 90.27% apps request at least one dangerous permission. 84.90% of the generated apps request access to external storage, as compared to 52.28% across all apps. Depending on what is stored to the external storage, this could lead to leak of private information. Generated apps have a high amount of services (83.14%) that are implicitly exported via intent-filters, as compared to 0.19% which are exported with the `exported=true` attribute. In general, compared to the corpus of all apps, generated apps are more prone to leaving activities, services, receivers and providers unprotected by not requiring permissions.

Vulnerability	Apps	App-Gens Apps
Req. Dangerous Permission	90.27%	99.34%
Def. Dangerous Permission	0.06%	0.00%
Req. External Storage	52,28%	84.90%
Req. SMS permission	4.11%	<0.01%
Expl. Exported Activities	6.10%	35.46%
Expl. Exported Services	6.90%	0.19%
Expl. Exported Receivers	3.00%	21.97%
Impl. Exported Activities	23.42%	30.00%
Impl. Exported Services	34.34%	83.14%
Impl. Exported Receivers	6.20%	1.26%
Exported Providers	7.30%	63.12%

Table 5: Manifest file analysis results. The activities, services, receivers and providers in this table are unprotected (insecure).

5. LIMITATIONS AND FUTURE WORK

In this preliminary work, we might under-report the use of frameworks and app generators, as we only classified the apps according to packagenames, name spaces and developer IDs. In the future, we plan to make use of further characteristic criteria for the classification, such as the signing certificates and the name spaces in the `AndroidManifest.xml`. For the comparison of security relevant behaviours, we only considered a small fraction of known vulnerabilities and violations of the Android security best practices. We plan to take more known vulnerabilities, not only limited to the misuse of TLS certificate validation, as well as further security best practices into account, such as using mis-using cryptographic libraries, or over-requesting permissions that are not used according to the call graphs. Eventually, we plan to conduct interviews with app generator providers and app developers to better understand the reasons that lead to insecure code (or the motivation for writing secure code).

6. REFERENCES

[1] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. MobiSys '11, ACM.

[2] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An empirical study of cryptographic misuse in android applications. CCS '13, ACM.

[3] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. CCS'12.

[4] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions Demystified. CCS'11, ACM.

[5] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. NDSS'14.