CHAPTER

7

On Discovering Vulnerabilities in Android Applications

X. Li[*], *L. Yu*[†], *X.P. Luo*[†]

[*]Chinese Academy of Sciences, Beijing, China [†]The Hong Kong Polytechnic University, Kowloon, Hong Kong

1 INTRODUCTION

The prosperity of the app economy boosts the number of mobile apps. More than two million Android apps[1] have been published in Google Play and around 2.5 million iOS apps[2] are listed in the Apple store. It is expected that the app economy could reach $101 billion in 2020.[3] At the same time, smartphones have become an indispensable part of our daily lives, and many apps have been downloaded and installed. A recent report shows that Android users will install on average 95 apps from June 2014 to January 2015 (Sawers, 2015).

Not all apps are well designed and developed. Recent studies illustrate that many apps are prone to various attacks because of their internal vulnerabilities. For example, HP research found that 90% apps are vulnerable after analyzing 2107 apps from companies on the Forbes Global 2000 (Seltzer, 2013). The latest application security report from Arxan shows that 90% of 126 mobile health and finance apps under investigation contain at least two critical security vulnerabilities (Arxan, Inc., 2016). Our study of 557 randomly collected apps with at least one million installations reveals that 375 apps (67.3%) had at least one vulnerability (Qian et al., 2015). There are many potential reasons for this embarrassing situation, such as short development cycles, lack of security awareness, insufficient security development guidelines.

In this chapter, we survey the vulnerabilities found in Android apps by collecting vulnerability reports from many sources, such as common vulnerabilities and exposures (CVE), because Android has occupied more than 80% of global market. Besides introducing major vulnerabilities in Android apps, we model how to discover them as graph traversals so that

[1] http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] http://www.pocketgamer.biz/metrics/app-store/

[3] http://venturebeat.com/2016/02/10/the-app-economy-could-double-to-101b-by-2020-research-firm-says/

http://dx.doi.org/10.1016/B978-0-12-804629-6.00007-9

VulHunter (described in Section 2.3) can use them to check whether an app has such vulnerabilities. Note that VulHunter uses Android property graphs (APGs) to represent apps and store them in the graph database. We also review the approaches for discovering various vulnerabilities in apps, which could leverage static analysis or dynamic analysis or the hybrid approach. Moreover, we discuss the limitations of existing approaches and suggest future directions of research (Heelan, 2011).

The chapter is organized as follows: Section 2 gives an introduction to the taxonomy of vulnerabilities in Android apps and the architecture of VulHunter. We introduce and model various common vulnerabilities in Section 3. Section 4 reviews existing approaches for discovering vulnerable apps and Section 5 discusses their limitations. We conclude the paper in Section 6.

2 BACKGROUND

2.1 Security Mechanisms of Android

We introduce three security mechanisms that are closely related to major vulnerabilities in apps: process sandbox, permission mechanism, and signature mechanism. There are other important security mechanisms in Android, such as interprocess secure communication mechanism, memory management mechanism, system partitions, and loading mechanism. Interested readers can refer to the relevant papers (Enck et al., 2009; Drake et al., 2014) for details.

Process sandbox. Android's process sandbox mechanism achieves a separation between apps. It creates a Dalvik virtual machine (DVM) instance for each app and grants a UID as the identification in the app installing process. In the Linux kernel, UID acts as the identification for different users. By default, different apps are separated. If they need to visit each other directly, they can set their SharedUserID to the same value.

Permission mechanism. Android's permission mechanism defines whether the app has the ability to access protected APIs and resources. The main functionalities of the permission mechanism include: permission confirmation during installation, permission check, permission use, and permission management during execution. A permission statement includes the permission name, the group it belongs to, and the protection level, which includes normal, dangerous, signature, signature or system. Developers can declare permissions required by the app through <uses-permission> tag in AndroidManifest.xml.

Signature mechanism. All apps must be signed with a private key before being released. The signature can be used to confirm the identity of developers, to test whether an app has any changes, and to establish a trusted relationship between two apps. Signature methods are divided into debug mode and release mode. The signature in debug mode is used for program testing during development, and the signature in release mode is used for publishing apps to markets.

2.2 Taxonomy of Android App Vulnerability

We have made a collection of 242 Android vulnerabilities, which have detailed information from many sources such as, vulnerability databases, security communities, and so on. After analyzing these vulnerabilities according to CERT Secure Coding Standards (CERT, 2015) and the OWASP's Mobile Security Project (OWASP, 2015), we classify the 242 vulnerabilities into 20 categories, as shown in Fig. 1, and number them M1-M20 (M20 can be negligible,

Name	Vulnerability Name	Name	Vulnerability Name
M1	Weak server side controls	**M11**	Linux kernel universal vulnerability
M2	Insecure data storage	**M12**	Program logic design flaw
M3	Insufficient transport layer protection	**M13**	Signatures vulnerability
M4	Unintended data leakage	**M14**	Code execution vulnerability
M5	Poor authorization and authentication	**M15**	Malicious application behavior vulnerability
M6	Broken cryptography	**M16**	Mobile terminal web vulnerability
M7	Client side injection	**M17**	Applications communications vulnerability
M8	Security decisions via untrusted inputs	**M18**	Configuration error vulnerability
M9	Webview vulnerability	**M19**	Denial of service vulnerability
M10	Linux kernel driver vulnerability	**M20**	others

FIG. 1 Taxonomy of Android vulnerability.

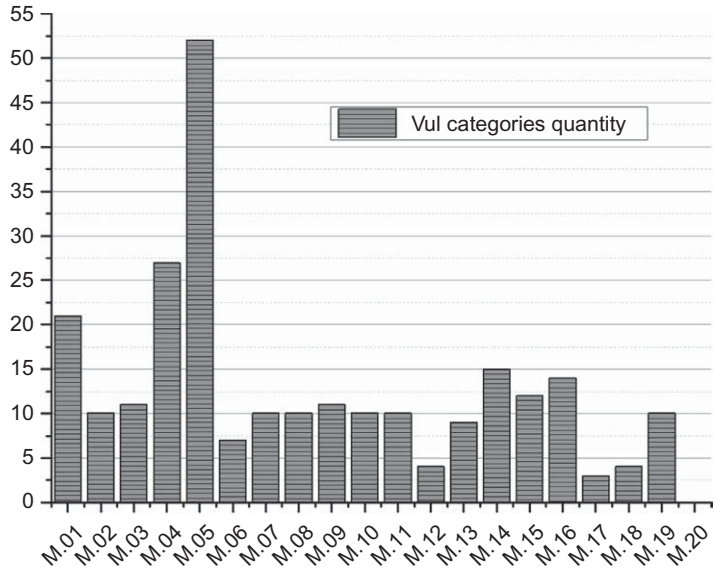

FIG. 2 Distribution of Android vulnerabilities.

because its number is very small). It is worth noting that M1-M8 are based on OWASP Mobile Risk Top10. Fig. 2 depicts the distribution of these vulnerabilities. It shows that most of the reported vulnerabilities result from poor authorization and authentication as well as unintended data leakage.

2.3 VulHunter

We proposed and developed Vulhunter to discover common vulnerabilities in Android apps (Qian et al., 2015). Given an app, Vulhunter will turn its dex file into an APG which integrates the abstract syntax tree (AST), interprocedure control-flow graph (ICFG), method call graph (MCG), and system dependency graph (SDG) of the app, and is stored in a graph

database. Moreover, we model five common vulnerabilities as graph traversals and perform them over APGs to determine whether an app is vulnerable. We have demonstrated in Qian et al. (2015) how to discover syntax level, control flow level, and data flow level vulnerabilities. In this chapter, we will cover more vulnerabilities and model how to identify them as graph traversals following (Qian et al., 2015). For ease of explanation, we will use the following symbols in this chapter:

$$MATCH^{p}_{label}; \tag{1}$$

$$ARG(N)_i; \tag{2}$$

$$N1-[R^{p}_{type}]^{len} \rightarrow N2, \tag{3}$$

where,

- $MATCH^{p}_{label}$ represents matching nodes with label `label` and properties `p`.
- $ARG(N)_i$ indicates traversing from *Invoke-Stmt* node `N` to get its ith argument.
- $N1-[R^{p}_{type}]^{len} \rightarrow N2$ denotes a path from node `N1` to node `N2`. The path is connected by relationship `type` with length of `len`, which can be omitted if it equals 1.

3 MODELING COMMON VULNERABILITIES

3.1 Insecure Data Storage

Sensitive information disclosure on external storage. If an app calls openFileOutput() but the second parameter is not set to Context.MODE_PRIVATE, this vulnerability may happen depending on whether the file is encrypted. To locate the suspicious code, we can conduct the following graph traversals:

$$MATCH^{p3}_{ast} \circ ARG(N)_2 \circ MATCH^{p2}_{stmt}, \tag{4}$$

where p2 refers to `openFileOutput` and p3 indicates the constant parameter node `MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE`. This graph traversal first locates the **statement** which invokes `openFileOutput` and then checks its second parameter. If the parameter in the AST tree is p3, the app may have such vulnerability.

3.2 Insufficient Transport Layer Protection

SSL/TLS trusts all certificates. When using SSL, if the app invokes setHostnameVerifier() with the parameter ALLOW_ALL_HOSTNAME_VERIFIER, this vulnerability exists, which could make the app vulnerable to the MITM (man-in-the-middle) attack. To detect the problematic code, we can use the following graph traversals:

$$MATCH^{p2}_{ast} \circ ARG(N)_1 \circ MATCH^{p1}_{stmt} \tag{5}$$

where p1 represents `setHostnameVerifier` and p2 denotes `ALLOW_ALL_HOSTNAME_VERIFIER`. This graph traversal first matches the statements which invoke method p1, and then checks whether its first parameter in the AST tree matches p2. If so, the vulnerability exists.

3.3 Unintended Data Leakage

Log sensitive information disclosure. If an app uses one of the following methods to save sensitive information: Log.d(), Log.e(), Log.i(), Log.v(), Log.w(), such a vulnerability may exist, because when the terminal is connected to the PC, Log information can be accessed. To detect such a vulnerability, we can use the following graph traversal:

$$MATCH_{stmt}^{p_1} - [R_{SDG_{Data}}]^{+} - > MATCH_{stmt}^{p_2} \tag{6}$$

where p1 denotes sink functions, such as *Log.i()*, *Log.d()*, and p2 refers to source functions that collect sensitive information, such as *getDeviceID()*. If there exists a path from source functions to sink functions, the vulnerability may exist. The edges on this path have a data dependency relationship.

3.4 Poor Authorization and Authentication

Exposed components (e.g., `contentProvider`, `service`). In the <provider> or <activity> tag of AndroidManifest.xml, if there is not the android:exported="false" statement for apps that set either `android:minSdkVersion` or `android:targetSdkVersion` to a value less than 17 or if the value of the statement is true, such a vulnerability may exist.

Intent leakage or tampering caused by permission granting. If there is an implicit intent broadcasting method call and there is a permission statement with a property value of FLAG_GRANT_WRITE_URI_PERMISSION or FLAG_GRANT_READ_URI_PERMISSION, then such a vulnerability may exist. To locate the suspicious code, we can use the following graph traversal:

$$MATCH_{stmt}^{p_1} - [R_{SDG_{Data}}]^{+} - > MATCH_{stmt}^{p_2} \tag{7}$$

$$MATCH_{ast}^{p_3} \circ ARG(N)_1 \circ MATCH_{stmt}^{p_2} \tag{8}$$

where p1 refers to *Context.sendBroadcast()*, p2 denotes *intent.addFlags()*, and p3 indicates `FLAG_GRANT_WRITE_URI_PERMISSION`. After finding the statements that call *Context.sendBroadcast()*, we check the intent initialization function *intent.addFlags()*. If any of them uses `FLAG_GRANT_WRITE_URI_PERMISSION` as a parameter, this vulnerability may exist.

3.5 Broken Cryptography

Weak AES encryption mode. AES encryption is initialized using javax.crypto.Cipher. If an app uses the ECB mode, the same plaintext will lead to the same ciphertext, which is vulnerable to dictionary attacks. We could use the following graph traversal to identify the corresponding problematic code:

$$MATCH_{stmt}^{p_1} \tag{9}$$

$$MATCH_{ast}^{p_2} \circ ARG(N)_1 \circ MATCH_{stmt}^{p_1} \tag{10}$$

where p1 refers to *Ciper.getInstance* and p2 denotes *AES/ECB/PKCS5Padding*. We first identify the statements that call *Ciper.getInstance*, and then check whether their first parameter is *AES/ECB/PKCS5Padding*. If so, the vulnerability may exist.

3.6 Webview Vulnerability

WebView malicious code execution. Such vulnerability exists in the system with API level less than 17. If the method webView.addJavascriptInterface() is called, and the applicable versions of the app in AndroidManifest.xml are not limited to API level 17 or above, the vulnerability may exist. We can use the following graph traversal to identify the vulnerable code:

$$MATCH_{stmt}^{p1} \tag{11}$$

where p1 denotes *webView.addJavascriptInterface*. After finding a statement that invokes this method, we check whether the minSdkVersion in manifest file is less than 17. If so, the vulnerability may exist.

3.7 App Communication Vulnerability

Content disclosure of implicit Intent broadcast. Intent is used for exchanging information between components in the same app or between apps. Note that using implicit Intent broadcast may lead to the disclosure of Intent contents. More precisely, if there is an invocation of the implicit Intent broadcasting method Context.sendBroadcast(), it will cause this problem. Therefore we can easily identify this issue using the following graph traversal:

$$MATCH_{stmt}^{p1} \tag{12}$$

where p1 denotes Context.sendBroadcast().

3.8 Configuration Error Vulnerability

Information disclosure due to the incorrect setting. Before releasing an app, the developer should ensure that the app is not debuggable in AndroidManifest.xml (android:debuggable="false"). Otherwise, such a vulnerability may exist.

4 DISCOVERING VULNERABILITIES

In this section, we review mechanisms for discovering vulnerabilities. They can be classified into three categories: static analysis-based methods, dynamic analysis-based methods, and hybrid methods. The static analysis-based methods usually investigate the Dalvik bytecode in the dex file or the Java class files converted from the dex file without running the app. The dynamic analysis-based methods commonly execute the app and monitor its behaviors, based on which vulnerabilities could be identified. Since both static analysis and dynamic analysis have their pros and cons, researchers propose hybrid methods that leverage the advantages of static and dynamic analysis.

4.1 Static Analysis-Based Approaches

Grace et al. studied the vulnerability of capability leaks, which belongs to M5 in Fig. 1, and developed a static analysis-based detection tool named Woodpecker, which could

discover both explicit and implicit capability leaks (Grace et al., 2012). Woodpecker builds CFGs of apps and then leverages the CFGs to determine whether there is privilege leakage. Specifically, for the explicit capability leakage, it inspects the preinstalled apps in the system by checking whether their components are exposed. If so, it conducts further path analysis to determine whether the leakage exists. For the implicit capability leakage, it checks each app's sharedUserId in Manifest. If it is used, this app's capability in terms of requested permissions is exposed to the apps with the shared UID.

Wei et al. proposed and developed Amandroid (Wei et al., 2014), a static analysis framework, for detecting intercomponent communication vulnerabilities, which include those in M4 and M17 in Fig. 1. More precisely, given an app, it first turns the Dalvik bytecode into intermediate representation (IR) and then constructs the IDFGs (intercomponent data flow graph) and DDGs (data dependence graph). After that, Amandroid looks for potential vulnerabilities from IDFGs and DDGs, including data leakage, data injection, and APIs misuse. Note that when building IDFGs, Amandroid computes the point-to information for all objects and fields in order to find the target of intent precisely.

Lu et al. developed CHEX (Lu et al., 2012) to identify component hijacking vulnerabilities, which cover those in M4, M5, and M7 in Fig. 1, such as permission disclosure, unauthorized data acquisition, intent deception, etc. CHEX employs data flow summaries to model the execution of entry points, and utilizes data flow analysis based on data dependency graphs to locate hijacking vulnerability. They further enhanced CHEX by proposing AAPL (Lu et al., 2015) to detect privacy leakage vulnerabilities. AAPL can reduce false positives. It combines a variety of static analysis methods, including opportunistic constant evaluation, object origin inference, and joint flow tracking, to detect more invisible data flows. Furthermore, AAPL employs a new approach called peer voting to filter out most of the legitimate privacy disclosures from the results, purifying the detection results for automatic and easy interpretation.

Gordon et al. proposed DroidSafe (Gordon et al., 2015) for statically detecting the apps' data stream related vulnerabilities, which include those in M4 in Fig. 1. DroidSafe creates models for 117 classes in the Java standard library as well as the Android library, Android runtime, and apps hidden state maintained by the Android runtime. It makes global resolution of Intent and Uri, and traces IntentFilter. By analyzing the sensitive data flows in apps, DroidSafe establishes the data flow graphs from sources to sinks. DroidSafe has covered all possible forms of communication to ensure a high coverage. Experimental results show that compared to other methods, DroidSafe increases the detection rate of sensitive data flow by about 10%.

Cao et al. designed and realized EDGEMINER (Cao et al., 2015) to address the issue of implicit control flow transitions through the Android framework. It analyzes the Android framework and constructs the call graphs to find potential callbacks. By using backward data flow analysis to identify registration-callback pairs, EDGEMINER outputs framework summary and then employs other static analysis tools for further analysis of apps.

Fahl et al. developed MalloDroid (Fahl et al., 2012) for studying the MITM vulnerability in Android apps. This tool can be integrated into the app market or installed in the user's mobile device. MalloDroid will inspect apps during installation. If it identifies potential SSL MITM vulnerability in an app, users will be alerted.

4.2 Dynamic Analysis-Based Approaches

Xing et al. discovered the severe vulnerability relevant to Android's upgrading mechanism. In particular, malware that lurks in the lower version system could get privilege rights after the system upgrades, then get access to users' data privacy. To spot such a vulnerability, the authors have developed the detection tool named SecUP. It constructs a database for saving exploit entry points and has a scanner for vulnerability identification. The detection module has constraint rules for determining the existence of vulnerabilities.

Wang et al. investigated the mobile-end same-origin policy (SOP) bypass vulnerability (Wang et al., 2013), and proposed a detection mechanism named Marbs. This system marks the source of information for each communication message and strengthen the SOP strength. The core of Marbs includes setOriginPolicy and checkOriginPolicy, which are implanted into DVM thread. SetOriginPolicy is open to all apps, and checkOriginPolicy is for the system kernel.

Wu et al. investigated the security impact of vendor customization on the Android system (Wu et al., 2013) and designed a system named SEFA to detect potential vulnerabilities. SEFA consists of three parts: including provenance analysis, permission usage analysis, and vulnerability analysis. First, it conducts provenance analysis, classifying the system apps into three categories, namely AOSP native applications, vendor-specific apps, and third-party apps. Then it performs the authorization analysis on the apps in order to check whether sensitive permissions are used. Finally, it checks whether there are redelegation vulnerabilities and privacy disclosure vulnerabilities.

Schrittwieser et al. studies the vulnerabilities in mobile messaging and VoIP apps (Schrittwieser et al., 2012). By conducting dynamic testing, it found vulnerabilities in VoIP and message apps' authentication mechanisms. The vulnerable apps make the user's phone number as the only certification basis, triggering a series of safety risks, such as account hijacking, spoof sender-IDs, and enumerating subscribers.

Hay et al. examined the IAC (Inter-Application Communication) vulnerability (Hay et al., 2015) and developed a new system named INTENT-DROID. It triggers sensitive APIs by constructing and sending probe intents. The externally observable indications are used to validate the test. The experimental result shows that INTENT-DROID can find a lot of IAC vulnerabilities in apps.

4.3 Hybrid Approaches

Sounthiraraj et al. investigated the SSL MITM vulnerability (Sounthiraraj et al., 2014). It first conducts static analysis to identify apps that may have such a vulnerability by examining the implementation of X509TrustManager and HostNameVerifier. More precisely, it identifies key entry points and builds the input function set for the next phase of dynamic trigger detection. Then, it applies UI automation to the key entry points of windows for triggering HTTPS communications, whose traffic will go through the MITM proxy. At the same time, the log of dynamic analysis will be recorded and used to determine whether the SSL MITM vulnerability exists.

Bhoraskar et al. developed Brahamstra (Bhoraskar et al., 2014) for detecting third-party component vulnerabilities in apps. It addresses the weakness of existing GUI testing tools

and can efficiently locate the triggering points of third-party libraries. First, the Execution Planner constructs page transition graphs to find execution paths to third-party libraries through static analysis. Then, the Execution Engine triggers apps in the simulator following the executable paths. Finally, the Runtime Analyzer captures and records running apps' operating status to determine whether there are third-party library vulnerabilities. To improve the speed of execution engine, Brahmastr rewrites app binaries to insert code that automatically invokes the callbacks triggered by the user.

Zhou et al. examined two types of vulnerabilities related to content providers, including passive content leakage and content pollution (Zhou and Jiang, 2013). They developed a detection tool named ContentScope. It firstly filters out apps that do not have exported content providers. Then, ContentScope determines the vulnerable apps by traversing the path from public content provider interface to the low-level database-operating routines in control flow graph. It will conduct dynamic analysis to confirm the vulnerability in apps.

5 DISCUSSION

5.1 Limitations in Static Analysis-Based Methods

By analyzing the bytecode instead of executing the app, static analysis may quickly locate problematic codes and achieve high code coverage. However, it suffers from several limitations. First, since it does not run the app, it is difficult to investigate codes using dynamic language features. For example, Java reflection is widely used in many apps, but it is challenging to investigate it in a precise and scalable manner (Smaragdakis et al., 2014). Moreover, apps could use dynamic class loading to dynamically extend its functionality. Static analysis cannot examine the dynamically loaded class if it is on the remote server. Besides using dynamic language features, more and more apps will employ the hardening services (or packing services) to protect themselves (Zhang et al., 2015b). Note that the dex file in the packed app does not contain the app's major functionalities, thus impeding static analysis.

Second, UI components can be dynamically added, and they will usually react to certain events, such as user input. Without executing the app, static analysis may miss such dynamic UI components and/or the reactions to certain events (Rountev and Yan, 2014; Shao et al., 2014). Third, the callback mechanism provided and orchestrated by the Android framework introduces challenging issues to static analysis (Cao et al., 2015), such as how to trace the information flow through the Android framework, etc. Note that the majority of existing studies just focus on apps, and they will be affected by the issues introduced by the Android framework. Fourth, there are still many open problems in static program analysis, such as pointer analysis, implicit flow analysis, and concurrent program analysis, to name a few (Hind, 2001). Moreover, given more than two million apps, how to quickly spot vulnerable apps is non-trivial.

5.2 Limitations in Dynamic Analysis-Based Methods

Since dynamic analysis executes apps, it has a high accuracy rate and will not be affected by hardening services. However, it also has some limitations. First, the code coverage rate of

dynamic analysis is low because it will take a very long time to execute all paths. Second, many existing dynamic analysis methods use emulator (e.g., QEMU) for analyzing apps. However, emulators may not support all features in a real smartphone, such as various sensors, USB, etc. Furthermore, a number of approaches for detecting emulator have been proposed (Jing et al., 2014), which may render many existing approaches useless. Third, the majority of existing approaches do not handle implicit flows. Note that tracking implicit flows could reveal more security vulnerabilities. However, it is challenging to track implicit flows (King et al., 2008).

5.3 Future Directions

Based on the above analysis, we list a few future directions to stir up research efforts into this important area. First, since static analysis and dynamic analysis have their own advantages and disadvantages, it is a promising approach to combine them together. For example, static analysis can quickly locate suspicious codes, guide the fuzzing test, generate GUI test cases, etc. Dynamic analysis can track the information flows, handle dynamic language features, etc. Second, since more and more apps employ various obfuscation and hardening techniques to protect themselves from being reverse engineered and analyzed, such techniques also make the detection of vulnerability more difficult. How to effectively and efficiently recover the original dex file is an interesting research problem. Third, although we summarize a number of vulnerabilities, there is a lack of formal approaches to represent them. Moreover, methods for discovering new vulnerability patterns are desirable. Leveraging machine learning techniques and incorporating more information in addition to code (Zhang et al., 2015a) would be a promising approach.

6 SUMMARY

In this chapter, we survey the vulnerabilities found in Android apps by collecting vulnerability reports from many sources, such as CVE. Besides introducing major vulnerabilities in Android apps, we model how to discover them as graph traversals following the definition in VulHunter Qian et al. (2015). We also review the approaches for discovering various vulnerabilities in apps, which could leverage static analysis, dynamic analysis, or the hybrid approach. Moreover, we discuss some limitations in existing approaches and suggest future directions of research.

References

Arxan, Inc, 2016. 5th annual state of application security report. https://goo.gl/mAqfx3.

Bhoraskar, R., Han, S., Jeon, J., Azim, T., Chen, S., Jung, J., Nath, S., Wang, R., Wetherall, D., 2014. Brahmastra: driving apps to test the security of third-party components. In: Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association. pp. 1021–1036.

Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y., 2015. Edgeminer: automatically detecting implicit control flow transitions through the android framework. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS).

CERT, 2015. Secure coding standards. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=111509535 (accessed August 15, 2015).

Drake, J.J., Lanier, Z., Mulliner, C., Fora, P.O., Ridley, S.A., Wicherski, G., 2014. Android Hacker's Handbook. John Wiley & Sons, New York.

Enck, W., Ongtang, M., McDaniel, P., 2009. Understanding android security. IEEE Secur. Privacy 7 (1), 50–57.
Fahl, S., Harbach, M., Muders, T., Baumgartner, L., Freisleben, B., Smith, M., 2012. Why eve and mallory love android: an analysis of android SSL insecurity. In: Proceedings of ACM CCS.
Gordon, M.I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., Rinard, M., 2015. Information-flow analysis of android applications in droidsafe. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). The Internet Society.
Grace, M., Zhou, Y., Wang, Z., Jiang, X., 2012. Systematic detection of capability leaks in stock android smartphones. In: Proceedings of NDSS.
Hay, R., Tripp, O., Pistoia, M., 2015. Dynamic detection of inter-application communication vulnerabilities in android. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis.
Heelan, S., 2011. Vulnerability detection systems: think cyborg, not robot. IEEE Secur. Privacy 9 (3), 74–77.
Hind, M., 2001. Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, pp. 54–61.
Jing, Y., Zhao, Z., Ahn, G.J., Hu, H., 2014. Morpheus: automatically generating heuristics to detect android emulators. In: Proceedings of ACSAC.
King, D., Hicks, B., Hicks, M., Jaeger, T., 2008. Implicit flows: can't live with 'em, can't live without 'em. Information Systems Security, Springer, New York, pp. 56–70.
Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G., 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of CCS.
Lu, K., Li, Z., Kemerlis, V., Wu, Z., Lu, L., Zheng, C., Qian, Z., Lee, W., Jiang, G., 2015. Checking more and alerting less: detecting privacy leakages via enhanced data-flow analysis and peer voting. In: Proceedings of NDSS.
OWASP, 2015. OWASP mobile security project. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks (accessed August 15, 2015).
Qian, C., Luo, X., Le, Y., Gu, G., 2015. Vulhunter: toward discovering vulnerabilities in android applications. IEEE Micro 35 (1), 44–53.
Rountev, A., Yan, D., 2014. Static reference analysis for GUI objects in Android software. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 143–153.
Sawers, P., 2015. Android users have an average of 95 apps installed on their phones, according to yahoo aviate data. http://goo.gl/cfs1bf.
Schrittwieser, S., Frühwirt, P., Kieseberg, P., Leithner, M., Mulazzani, M., Huber, M., Weippl, E.R., 2012. Guess who's texting you? evaluating the security of smartphone messaging applications. In: Proceedings of NDSS.
Seltzer, L., 2013. HP research finds vulnerabilities in 9 of 10 mobile apps. http://goo.gl/esxBkb.
Shao, Y., Luo, X., Qian, C., Zhu, P., Zhang, L., 2014. Towards a scalable resource-driven approach for detecting repackaged android applications. In: Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC).
Smaragdakis, Y., Kastrinis, G., Balatsouras, G., Bravenboer, M., 2014. More sound static handling of java reflection. Technical report.
Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., Khan, L., 2014. SMV-hunter: large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps. In: Proceedings of NDSS.
Wang, R., Xing, L., Wang, X., Chen, S., 2013. Unauthorized origin crossing on mobile platforms: threats and mitigation. In: Proceedings of the 2013 ACM SIGSAC of Conference on Computer & Communications Security. ACM, pp. 635–646.
Wei, F., Roy, S., Ou, X., et al., 2014. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1329–1341.
Wu, L., Grace, M., Zhou, Y., Wu, C., Jiang, X., 2013. The impact of vendor customizations on android security. In: Proceedings of ACM CCS.
Zhang, T., Jiang, H., Luo, X., Chan, A.T., 2015. A literature review of research in bug resolution: tasks, challenges and future directions. Comput. J. 59 (5), 741–773.
Zhang, Y., Luo, X., Yin, H., 2015. Dexhunter: toward extracting hidden code from packed android applications. In: Proceedings of ESORICS.
Zhou, Y., Jiang, X., 2013. Detecting passive content leaks and pollution in android applications. In: Proceedings of NDSS.

ABOUT THE AUTHORS

Xiaoqi Li received his B.S. from Central South University and is a graduate student at the Chinese Academy of Sciences. His research focuses on Android security and vulnerability.

Le Yu received his B.S. and M.S. from Nanjing University of Posts and Telecommunications. He is currently a research assistant at Hong Kong Polytechnic University. His research focuses on Android security and privacy.

Xiapu Luo is a research assistant professor in the Department of Computing at Hong Kong Polytechnic University. His research focuses on smartphone security, network security and privacy, and Internet measurement.