When Program Analysis Meets Mobile Security: An Industrial Study of Misusing Android Internet Sockets

Wenqi Bu
East China Normal University, China
buwenqi@gmail.com

Minhui Xue
New York University Shanghai
East China Normal University, China
minhuixue@nyu.edu

Lihua Xu
East China Normal University, China
lhxu@cs.ecnu.edu.cn

Yajin Zhou
China
yajin@vm-kernel.org

Zhushou Tang
Pwnzen Infotech Inc., China
ellison.tang@gmail.com

Tao Xie
University of Illinois at Urbana-Champaign, USA
taoxie@illinois.edu

ABSTRACT

Despite recent progress in program analysis techniques to identify vulnerabilities in Android apps, significant challenges still remain for applying these techniques to large-scale industrial environments. Modern software-security providers, such as Qihoo 360 and Pwnzen (two leading companies in China), are often required to process more than 10 million mobile apps at each run. In this work, we focus on effectively and efficiently identifying vulnerable usage of Internet sockets in an industrial setting. To achieve this goal, we propose a practical hybrid approach that enables lightweight yet precise detection in the industrial setting. In particular, we integrate the process of categorizing potential vulnerable apps with analysis techniques, to reduce the inevitable human inspection effort. We categorize potential vulnerable apps based on characteristics of vulnerability signatures, to reduce the burden on static analysis. We flexibly integrate static and dynamic analyses for apps in each identified family, to refine the family signatures and hence target on precise detection. We implement our approach in a practical system and deploy the system on the Pwnzen platform. By using the system, we identify and report potential vulnerabilities of 24 vulnerable apps (falling into 3 vulnerability families) to their developers, and some of these reported vulnerabilities are previously unknown. The apps of each vulnerability family in total have over 50 million downloads. We also propose countermeasures and highlight promising directions for technology transfer.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; **Software and application security**;

KEYWORDS

Android security; Internet sockets; Vulnerability analysis

ACM Reference format:
Wenqi Bu, Minhui Xue, Lihua Xu, Yajin Zhou, Zhushou Tang, and Tao Xie. 2017. When Program Analysis Meets Mobile Security: An Industrial Study of Misusing Android Internet Sockets. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 6 pages.
https://doi.org/10.1145/3106237.3117764

1 INTRODUCTION

Mobile apps and their users have witnessed a massive growth over the last decade. As such, the security and privacy concerns are increasingly becoming the focus of great concern to various stakeholders. Program analysis, through its extensive applications on Android platforms [4, 7, 9–13], has demonstrated great potential in vulnerability disclosure. Despite sophisticated, static analysis explores the app behavior for all possible execution paths. In Android apps, static analysis usually requires to construct various dependency graphs from multiple entry points, elongating the process time with respect to graph construction and exploration.

Transferring academic research on vulnerability detection to an industrial setting requires significant adaption to account for practical realities of scale and cost. Modern software-security providers, such as Qihoo 360 and Pwnzen (two leading companies in China), are often required to process more than 10 million mobile apps at each run, for detecting potential vulnerabilities. Regardless of the number of Android apps in the target industrial environment, some of these apps are even too complex to analyze in its entirety. Furthermore, in security industry, it is inevitable to engage human inspections, *e.g.*, toward constructing vulnerability exploits to confirm the identified potential vulnerabilities.

To address such challenges in industrial environments, we present a novel approach for vulnerability detection. The goal of our work is to enable practical identification of vulnerable Internet-socket usage pertinent to a large-scale industrial setting. Our insights are three-fold. (1) Vulnerabilities, even all of those related to Internet sockets, typically carry different characteristics, and thus we should not unify strategies for analyzing different mobile apps as traditional analysis approaches did. (2) Mobile apps related to the same type of Internet socket usage tend to preserve similar features (*e.g.*, class names and permissions) introduced by the same SDK or library (used by the apps). (3) Typical static analysis techniques seek to analyze an app in its entirety; doing so is not only challenging given the nature of Android apps, but also time consuming.

To cope with the large number of apps to be processed in an industrial setting, we design a lightweight detection process to specifically avoid statically analyzing every app in the environment. Specifically, we leverage features to characterize similar usage patterns of apps and categorize them accordingly. Before sophisticated program analysis is applied, the whole set of apps are filtered, and only potentially vulnerable apps require further analysis.

To tackle with the complexity of some real-world apps, we integrate dynamic analysis with static analysis to identify vulnerable apps. One of our observations is that a typical vulnerable app opens a port by default at launch time. By dynamically observing the server socket instantiation, our approach is able to avoid traditional static analysis for reachability [10, 12], which identifies socket instantiation from all entry points of the app. Exploring all the possibly reachable paths from all entry points of an app would undoubtedly elongate the analysis process, sometimes even not realistic in an industrial setting.

Despite effective, our lightweight process may bring false positives to the detection result. To account for such false positives, we leverage human inspectors, whose efforts are inevitable in security industry. With our filtering and analysis techniques, the number of potential vulnerable apps that need examining by human inspectors is greatly reduced. Hence it is easier to identify the root causes of certain vulnerabilities, which, in our observation, are typically introduced by SDK or third-party libraries. Through examining the vulnerability families, an enriched set of features can be identified, and hence substantially boost the chances of identifying vulnerable apps.

We implement our iterative approach in a practical system and deploy the system on Pwnzen's automation mobile security platform, Janus [5]. The approach takes three main iterative steps, including human inspectors in the loop: Filter, Analyzer, and Feature Extractor. Our approach seeks to identify new vulnerabilities and categorizes potential vulnerability families with corresponding features. Features from every family are further refined and enriched for the next iteration to identify more vulnerabilities. To date, our filter is specifically designed for vulnerabilities related to Internet sockets, but the approach can be well generalized for detecting other types of vulnerabilities.

In summary, the paper makes the following main contributions:

- We discuss main challenges encountered and insights gained from vulnerability identification in an industrial setting.
- We propose a hybrid approach that integrates novel static and dynamic analyses to identify vulnerabilities based on Internet sockets in a large-scale industrial environment. Compared with prior work [10], our approach can identify vulnerabilities based on Internet sockets precisely with limited human effort. Based on the Janus platform with categorization, we can quickly identify vulnerable apps belonging to the same family and fix them promptly.
- For our initial experiment, we identify 24 vulnerable apps falling into 3 vulnerability families (the QQ family, Huya family, and ES family). The apps of each vulnerability family in total have over 50 million downloads.

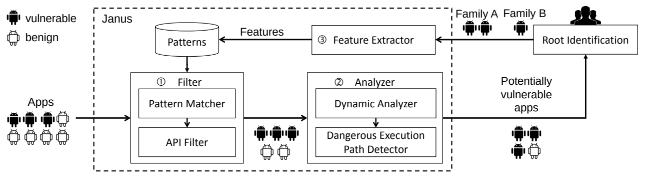

Figure 1: Overview of our approach

- We offer three lessons learned from our study. We also discuss countermeasures to prevent vulnerabilities based on Internet sockets.

2 OUR APPROACH

In this section, we propose a general approach to examine the use of Internet sockets of apps, and attempt to identify what apps are most likely vulnerable for validation. Figure 1 shows an overview of our approach, which contains three main steps: **Filter**, **Analyzer**, and **Feature Extractor**. The **Filter**, which contains two sub-steps: Pattern Matcher and API Filter, filters out irrelevant apps that are less likely to use Internet socket APIs; the **Analyzer** integrates dynamic analysis (Dynamic Analyzer) with static analysis (Dangerous Execution Path Detector), to further identify potentially vulnerable apps; the **Feature Extractor** extracts signature patterns for each identified vulnerability family, after manual root identification. The basic steps of our approach are detailed as follows.

Step 1. Filter. Step 1 filters out apps that do not match the identified patterns. To date, our filter is specifically designed for vulnerabilities related to Internet sockets. Hence, we start with the wormhole pattern [1], and also filter out the apps that do not use Internet socket APIs.

(a) Pattern Matcher. We first filter out apps that do not preserve the defined patterns. A pattern consists of different features, each of which represents an identified signature of specific vulnerabilities. A feature can represent either a class name, a string constant, a third library name and so on. The features are combined via *and*, *or* conjunctions as logic expressions to construct a pattern, which is then be processed over the database. The output of Pattern Matcher is a subset of apps, which are to be further filtered by API Filter.

(b) API Filter. At this step, we check whether an app requests permissions to use certain APIs and whether it actually uses the requested APIs. Take Internet socket APIs as an example. We check, at this stage, (1) whether remaining apps have `INTERNET` permissions. Apps without this permission are certainly not vulnerable because using Internet sockets on Android apps must require the `INTERNET` permission. We also check (2) whether these apps use Internet socket APIs to start server sockets. It is our observation that the usage of Internet socket APIs on the server side makes an app particularly vulnerable, because server sockets can arbitrarily accept random connection requests from client sockets. It is worth noting that our approach does not take client sockets into consideration, because a client socket is required to pinpoint a socket address that it connects to and then establishes the connection channel between a server socket associated with the designated socket address.

Moreover, rather than focusing on only the main *dex* file of an app, our detection extends to all *jar/dex* files that can be dynamically loaded in its *assets* directory. The reasons are two-fold. (1) Traversing all *jar/dex* files that can be dynamically loaded can quickly filter out apps that are definitely not vulnerable, but this coarse-grained filtering would introduce some false alarms. (2) Only the *assets* directory can contain *jar/dex* files according to the asset-packaging process in Android.

Specifically, our API Filter is implemented upon Androguard [4]. After pattern matching, the *AndroidManifest.xml* of each remaining app is extracted and checked for the `INTERNET` permission. We scan the main *dex* with two Dalvik instructions: *Ljava/net/ServerSocket;->\<init\>* and *Ljava/net/DatagramSocket;->\<init\>* to check whether it has the `INTERNET` permission. These two instructions represent the initialization of TCP and UDP server sockets, respectively. As mentioned earlier, the *jar/dex* files in the *assets* directory are also scanned. As a result, the Filter returns an app list in which apps use Internet sockets.

Step 2. Analyzer. Step 2 aims to identify apps that start server sockets at launch time and contain dangerous execution paths from the client socket connection. The outputs are potentially vulnerable apps with information of dangerous execution paths.

(a) Dynamic Analyzer. We conduct dynamic analysis to identify apps that start server sockets at launch time. Each app is installed and launched automatically, with its server-socket information collected. The server-socket information consists of the type of server socket and the socket address. We use the Xposed[1] framework to intercept server socket `get-client-socket` method and record its call location, which is then passed onto the Dangerous Execution Path Detector.

As mentioned earlier, we rely on only dynamic analysis to find potentially vulnerable server sockets of apps, for two reasons. (1) It is our observation that vulnerabilities based on server sockets that start listening at launch time have realistic significance, because an attacker is able to exploit these vulnerabilities within adequate time. If starting a server socket needs user interactions, we do not consider it vulnerable because such socket starting can be a normal functionality, *e.g.*, clicking to start a server socket to transfer a file to a computer. (2) Static analysis techniques have inherent limitations such as handling reflection and implicit invocation, as well as high time consumption.

(b) Dangerous Execution Path Detector. The Dangerous Execution Path Detector aims to check whether data accepted from client sockets can be used to invoke sensitive APIs, which are usually used to extract sensitive information or execute privileged commands. By leveraging a compound list of sensitive APIs [4, 7, 8], we use the `get-client-socket` method to create an entry point and build the inter-procedural data flow graph (IDFG). If there exists an execution path that travels from the entry point to a sensitive-API invocation, there is a high probability for data accepted from client sockets to trigger dangerous executions. The identified execution path with various authentication mechanisms serves as a prerequisite for manual exploitation.

Specifically, we implement the Dynamic Analyzer and Dangerous Execution Path Detector on top of Xposed and Amandroid [13]. Amandroid is a cutting-edge Android static analysis tool, which provides points-to information for all objects. We leverage Amandroid to construct the IDFG from apps' non-native part for performing app-layer analysis. The Xposed framework can record information of definite method invocations at runtime, and we use it to record call locations of the `get-client-socket` method of the server socket API.

In particular, we take as input potentially vulnerable apps after applying Step 1 Filter. First, we take the input apps as a set $\mathcal{A}$. For each $app \in \mathcal{A}$, we write a Python script to automatically install and launch each app, and collect information of server-socket address $s_{\text{add}} \in \mathcal{S}_{\text{add}}$, where $\mathcal{S}_{\text{add}}$ is a socket-address set. We use f_{socket} to represent a *getSocketAddress* function, that is, $s_{\text{add}} \leftarrow f_{\text{socket}}(app)$, for all $apps \in \mathcal{A}$. We utilize the Xposed framework to fetch locations when the server socket invokes the *accept()* or *receive()* method. We use client socket-connection location information $r_{\text{entry}} \in \mathcal{R}_{\text{entry}}$ as an entry point to construct the IDFG, where $\mathcal{R}_{\text{entry}}$ is an entry point set for all server sockets with multiple socket addresses. We use f_{entry} to represent a *getEntryPoint* function, that is, $r_{\text{entry}} \leftarrow f_{\text{entry}}(app, s_{\text{add}})$, for all $apps \in \mathcal{A}, s_{\text{add}} \in \mathcal{S}_{\text{add}}$. If r_{entry} is not null, we then initialize a *result* for each node $r_{\text{entry}} \in \mathcal{R}_{\text{entry}}$. Particularly, a *result* consists of four parts: (1) a string *apkname* a_{apk} that represents an app's name; (2) a boolean flag b_{vul} that forms a binary judgment of an app's vulnerability ($b_{\text{vul}} = 1$, it's vulnerable; otherwise); (3) a string `SocketAddress` s_{add} that shows an Internet server-socket address; (4) a *dangerous-paths* set $\mathcal{D}$ that shows dangerous execution paths. At the final step, we take each entry point $r_{\text{entry}} \in \mathcal{R}_{\text{entry}}$ to construct the IDFG if it is not null. The *find-path* function f_{path} takes in an entry point r_{entry} to identify all paths $p_{\text{path}} \in \mathcal{D}$ invoking the sensitive API files s_{api}. If there exists at least one path invoking the sensitive APIs, we label the app as vulnerable. We repeat this process until all the input apps are analyzed. We summarize the implementation in Algorithm 1.

ALGORITHM 1: Implementation of Step 2

```
Input: $\mathcal{A}$: Apps using Internet socket APIs
Output: results: Vulnerable apps with dangerous execution path information
 1: for $app \in \mathcal{A}$ do
 2:     $s_{\text{add}} \leftarrow f_{\text{socket}}(app)$
 3:     $r_{\text{entry}} \leftarrow f_{\text{entry}}(app, s_{\text{add}})$
 4:     for $r_{\text{entry}} \in \mathcal{R}_{\text{entry}}$ do
 5:         if $r_{\text{entry}} \neq$ null then
 6:             Build IDFG for $app$ from $r_{\text{entry}}$
 7:             $p_{\text{path}} \leftarrow f_{\text{path}}(r_{\text{entry}}, s_{\text{api}})$.
 8:             if $\# p_{\text{path}} > 0$ then
 9:                 $b_{\text{vul}} = 1$.
10:             end if
11:             ($results+ = a_{\text{apk}}, b_{\text{vul}}, s_{\text{add}}, p_{\text{path}}$)
12:         end if
13:     end for
14: end for
15: return results
```

Step 3. Feature Extractor. We understand that human inspection effort is inevitable in software-security industry. To better serve the inspectors, we leverage information of dangerous execution paths to guide manual exploitation to construct malicious client sockets. Furthermore, such information also helps understand the root causes of these vulnerabilities, such as exposing a function with weak authentication, importing a third-party library without evaluating its security, having back doors for collecting user information, and so on.

[1]http://repo.xposed.info/

Table 1: Apps using Internet sockets from two Android app markets

	# Apps using Internet socket APIs	# TCP Internet sockets	# UDP Internet sockets
Tencent Market	361	352	361
Google Play Store	176	228	125
Total	537	589	477
Percentage	36.7%	55.3%	44.7%

After reasoning about these vulnerabilities, we categorize vulnerable apps to different families according to different root causes. Then we extract different feature groups and have them installed to Janus [5] as different patterns. These patterns are used in the Pattern Matcher of Step 1. In summary, after an initial iteration, we obtain new vulnerable apps and new patterns. These new patterns represent root causes of these vulnerability families. When the next iteration starts, one of these patterns serves to be an input to the Pattern Matcher of Step 1. Another input is apps that are in need of analysis. The Pattern Matcher feeds all apps that match the pattern as input to the API Filter. Through the Filter, Analyzer, and Root Identification steps, we can further add new vulnerable apps and new patterns to the new iteration. The iteration for a pattern input stops until we cannot identify new vulnerable apps.

3 RESULTS

3.1 Usage of Server Internet Socket APIs

We evaluate our system on the 1, 464 top downloaded apps crawled from two Android app markets, the Tencent Android Application Market[2] and Google Play Store, of which 840 (resp. 624) apps crawled from the Tencent Android Application Market (resp. Google Play Store) are classified into 21 (resp. 32) categories. Table 1 shows the usage of server sockets of these Android apps. Note that an app can use multiple Internet sockets. Figures 2 and 3 show the server socket usage in different categories, as shown in decreasing order in terms of the number of socket APIs with respect to different categories. Each category of the Tencent Android Application Market (resp. Google Play Store) has 40 (resp. 20) top apps, except that the finance category of Google Play Store has only 6 top apps due to the market ecosystem. As shown in Figure 2, the top 5 categories, which take up more than 50% apps, are children, video, music, entertainment, and tools. As shown in Figure 3, the top 6 categories of Google Play Store, which take up more than 50% apps, are video, productivity, photography, lifestyle, sports, and music and audio.

3.2 Performance Evaluation

As mentioned earlier, static analysis seeks to analyze each app in its entirety, which is a time-consuming task. Take Amandroid [13] as an example, building the IDFG for most apps can take anywhere from 10s to 1000s each, with a median process time of 29s. It may even take more than 1 hour for each of some complex apps. It is unacceptable in our environment, given the Janus database containing more than 11 million apps. To alleviate the analysis burden, we filter out the irrelevant apps before entering the analysis phase, so that both the total analysis time for the Janus database and the number of apps required to be analyzed are tremendously reduced. In our experiment, we are able to filter the Janus database within

[2]http://android.myapp.com/

Table 2: Breakdowns of vulnerability families

Family	# Features	Efficiency	# Remaining apps	Infected apps
QQ family	2	57s for 11 million+	223	QQBrowser and QQHotspot
Huya family	2	14s for 11 million+	380	HuyaLive and HuyaHelper
ES family	3	3s for 11 million+	707	ES-File-Explorer

Table 3: Performance details

	Filter	Dynamic analysis	Dangerous exec. path detection	Root identification & Feature extractor
# Tencent Market remaining apps	361	75	24	20
# Google Play Store remaining apps	176	17	10	4
# Total remaining apps	537	92	34	24

1 minute. Specifically, as shown in column *Efficiency* in Table 2, depending on different features defined in each pattern, the filtering process time ranges from 3s to 57s. As filtering results, only a few hundreds of apps remain for further analysis, as shown in column *# remaining apps* in Table 2. Table 2 shows three families for the vulnerabilities that we mine out from the database. Take the QQ family as an example. We extract 2 features as a pattern to conduct the filtering process, which takes 57 seconds and results in 223 suspicious apps of the QQ family. Some of these suspicious apps are multiple versions of the same app. After manual exploitation, we confirm two vulnerable apps of the QQ family: QQBrowser and QQHotspot (see more details in Table 2).

To further evaluate our approach, again we use the top downloaded apps from two existing app stores (Table 1), and report the detailed analysis results in Table 3. After the filtering step, 537 possible vulnerable apps out of 1, 464 apps remain to be further analyzed. We then pass 537 apps to the Dynamic Analyzer and Dangerous Execution Path Detector. Although Dangerous Execution Path Detection is the most time-consuming intermediate step, we think it is indispensable. A decrease in the number of apps to be analyzed can boost the overall efficiency, as reflected by recent research that takes median process time 61.5s to identify dangerous execution paths [10]. Finally, we pass 34 apps (remaining after the steps of Dynamic Analysis and Dangerous Execution Path Detection) to Root Identification, and manually confirm in total 24 vulnerable apps in the end. These vulnerable apps expose sensitive API invocations without any authentication (or with weak authentication) and can be exploited by any remote attackers, such as requesting sensitive information and installing random apps.

3.3 Case Study

Next, we zoom in on the severe zero-day vulnerabilities pertinent to Internet sockets, and these vulnerabilities are included in popular apps installed by thousands of millions of users. Because of weak authentication mechanisms, an attacker can remotely exploit those vulnerabilities to (1) install arbitrary apps on target devices, (2) upload random files on target devices, (3) obtain all files and sensitive information from target devices.

3.3.1 ***Wormable QQBrowser: QQ family.*** QQBrowser is a major mobile app of Tencent, with up to 281 million monthly active users and more than 600 million downloads on the Tencent Android Application Market [3]. Other than its main browser functionalities, QQBrowser also offers reading, shopping, etc. In order to reuse

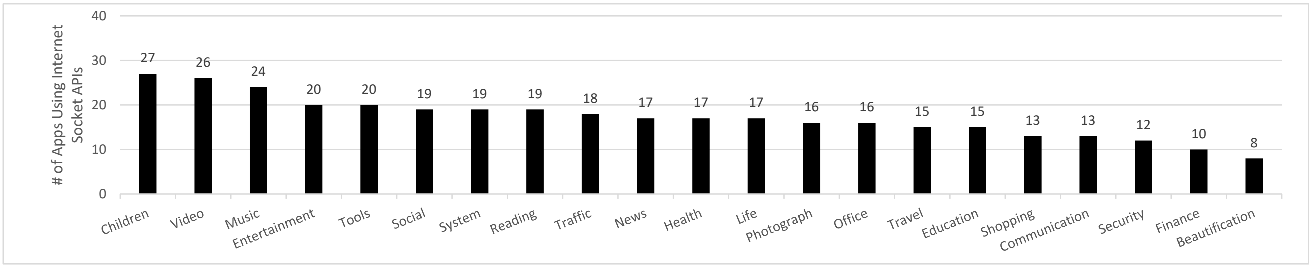

Figure 2: Usage of server Internet socket APIs from Tencent Android Application Market

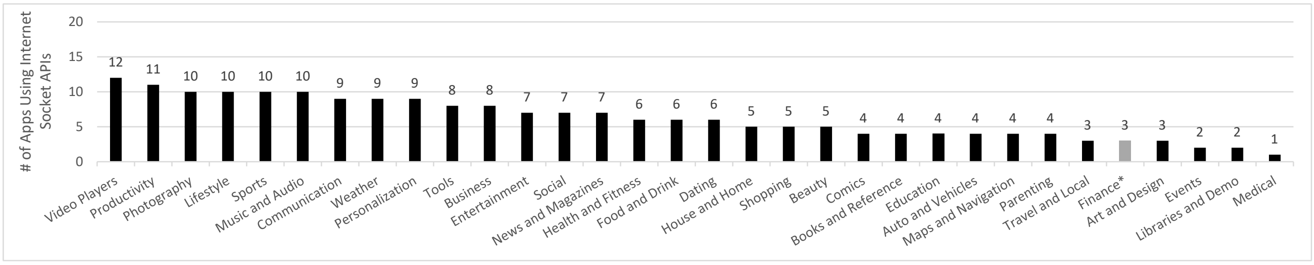

Figure 3: Usage of server Internet socket APIs from Google Play Store

some app modules and reduce the coupling dependency between different modules, QQBrowser separates different app modules from its browser. Each app module is encapsulated in a single *dex* file, which is packaged in the apk *assets* folder. QQBrowser uses a dynamic loading method to load different app models.

The QQBrowser vulnerability that we identify resides in the application module, used for transferring files from desktop computers to mobile devices through LAN. The vulnerable location is encapsulated in a *dex* file "com.tencent.mtt.sniffer.jar" under the *assets* folder. When devices connect to Wi-Fi, the QQBrowser will start a server socket to listen on the wildcard address 0.0.0.0 and port 8786. The QQBrowser vulnerability uses a certain HTTP data structure to transfer data to pass through the weak authentication. Only data accepted from client sockets associated with a certain structure can trigger sensitive API invocations. For example, a request method *post* and a request path *downloadandinstall* will allow an attacker to remotely have malicious apps installed. Other vulnerability exploits such as obtaining installed app records, peeking location privacy information (IP addresses), and having elaborate files uploaded to the target device SD card for performing colluding attacks can also be obtained. When the target and the attacker are not on the same Wi-Fi network, the attack can also be launched by distributing malicious wormable links. QQBrowser versions 6.4-6.9 and QQHotspot versions 1.0-1.2 all suffer from this vulnerability. We have reported this vulnerability to Tencent, and soon it has been acknowledged and fixed by Tencent.[3] The vulnerability sample has also been imported to the China National Vulnerability Database of Information Security, numbered *CNNVD-201609-627*[4] (see more details [2]).

3.3.2 ***Arbitrary file access: Huya family.*** HuyaLive is a very popular live app in China, with more than 72 million downloads on the Tencent Android Application Market.[5] After launching the app, we find that the app starts two TCP server sockets that can be remotely connected. These two server sockets listen on the wildcard address and ports 8082 and 8083. To simplify description, we term these two server sockets as *Server8082* and *Server8083*, respectively. Both server sockets are implemented based on an open-source web server, *NanoHTTPD* [6]. *NanoHTTPD* consists of a single Java file used to embed in apps. After passing through the Dangerous Execution Path Detector, we identify many sensitive API invocations, such as a File object's *list* method used for listing all files under a directory and a PrintWriter object's *print* method used for sending data back to client sockets. Through manual analysis, we find that both *Server8082* and *Server8083* are able to return HTML pages with little authentication. A remote attacker can access arbitrary files from victim phones through *Server8082*. The only authentication is the request path that can extract from the former HTML page. *Server8083* is designed for debugging, exposing some functionalities to allow app developers to invoke remote client sockets. These functionalities include causing garbage collection, watching live CPU usage, and triggering an exception to shut down an app, by which a remote attacker can launch DoS attacks. Two apps in the Huya Family (Huyalive and HuyaHelper) contain this vulnerability.

3.3.3 ***Random uploading: ES family.*** ES-File-Explorer is a very popular file management app, with more than 300 million downloads.[6] Besides basic functionalities of file management, ES-File-Explorer also provides file classification, garbage cleaning, file transfer between or across devices. Through our analysis, we find that ES-File-Explorer starts a server socket listening on the wildcard address and port 42135 when being launched. Through the Dangerous Execution Path Detector, we find a path for invoking *getExternalStorageDirectory()*, used to reach the SD card directory. Only data following a certain structure accepted from client sockets

[3]http://bbs.mb.qq.com/thread-1418941-1-1.html
[4]http://www.cnnvd.org.cn/web/xxk/ldxqById.tag?CNNVD=CNNVD-201609-627
[5]http://sj.qq.com/myapp/detail.htm?apkName=com.duowan.kiwi
[6]http://www.estrongs.com/

can trigger sensitive API invocations. We also identify that ES-File-Explorer customizes a protocol termed "*MYPOST*" similar to HTTP, and uses constructed data associated with the data structure to upload random files to target mobile devices. In fact, the vulnerability, existing in a component model *QuickTransfer*, is exploited to transfer data between different devices. The privileged component can be exposed to a remote attacker after launching the app through Internet socket channels, allowing the attacker to exploit the vulnerability to upload malicious apps for performing colluding attacks.

4 DISCUSSION

In this industrial study, we demonstrate that our proposed approach is largely effective and efficient. However, we also identify our approach's limitations as follows in such industrial setting.

- As we implement our current approach based on some off-the-shelf static analysis tools, our approach is inherently confined to the capability of these cutting-edge static analysis tools. Such limitations include handling Java reflection and dynamic loading.
- Our current approach does not take native code into consideration, but apps can embed the open-port functionality into native code for either disguising their stealthy behavior or performance purposes. Because analysis for capturing the control-flow jumps from native code to the Java layer is time consuming, our current approach does not include such analysis.
- App inspection cannot be fully automatically conducted. In this paper, we attempt to minimize the human efforts to examine and identify the root causes of vulnerabilities. Minimizing human efforts can facilitate technology adoption in practice.

In this industrial study, one of our research goals is to offer lessons learned from the security implication of Internet sockets. We list them as follows:

- When importing a third-party library or an internal library, developers need to do sufficient security testing to prevent wormable vulnerabilities such as those in QQBrowser.
- Developers need to remove debug code before releasing their apps. Not promptly removing the debug code can do great harm to target users, as demonstrated by the case of HuyaLive, which exposes files on the target mobile device to a remote attacker.
- The procedure of file sharing on the Wi-Fi network should be fully aware to users. Any client connections to the server sockets should also be notified and controlled by users.

As shown in our study, the misuse of Internet sockets on Android has resulted in many severe vulnerabilities. We discuss two possible countermeasures to alleviate the problem:

- Using a server Internet socket as an IPC channel should be bound to the local IP address 127.0.0.1 rather than using the default wildcard address. Another option is to use Android Unix domain sockets.
- If server Internet sockets accepting the remote connection are required, authentication logic should be better allocated to the remote server instead of being embedded in apps. Another remedy is to display a dialog window with an alert message to any connection. Furthermore, it would be more secure if users are allowed to authorize the connections; indeed how to translate deep security implications to common users remains important future directions.

5 CONCLUSION

In this paper, we have proposed a hybrid approach to detect vulnerabilities based on Internet sockets in large-scale industrial environments. We have integrated categorization with novel dynamic and static analyses to accelerate analysis of apps. Our approach helps identify 24 vulnerable apps (falling into 3 vulnerability families). The apps of each vulnerability family in total have over 50 million downloads.

As it is exciting to facilitate new industrial findings using real-world data, our research is the first to deploy a practical system for vulnerability detection on the state-of-the-practice Janus industrial platform. We expect that Android app vulnerabilities in the wild could be particularly amenable to the use of our proposed approach, and our general methodology can serve as a supplement to other detection schemes in practice.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China, under Grants 61502170 and 61673180, in part by the East China Normal University travel grant, and in part by the Shanghai Pujiang Program, under Grant 16PJ1430800. The work was also supported in part by NSF under grants no. CCF-1409423, CNS-1434582, CNS-1513939, CNS-1564274.

REFERENCES

[1] 2015. Baidu Wormhole Vulnerability. (2015). Retrieved May 10, 2017 from http://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/
[2] 2016. PANGU Wormable Browser. (2016). Retrieved May 7, 2017 from http://blog.pangu.io/wormable-browser/
[3] 2016. QuestMobile Top Chinese Apps Rankings. (2016). Retrieved May 7, 2017 from http://www.questmobile.com.cn/blog/en/blog_64.html
[4] 2017. Androguard. (2017). Retrieved May 7, 2017 from https://github.com/androguard/androguard
[5] 2017. Janus. (2017). Retrieved May 10, 2017 from http://cloud.appscan.io/
[6] 2017. NanoHTTPD. (2017). Retrieved May 7, 2017 from https://github.com/NanoHttpd/nanohttpd
[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. PLDI*. 259–269.
[8] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proc. CCS*. 217–228.
[9] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick Mcdaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. OSDI*. 393–407.
[10] Yunhan Jack Jia, Qi Alfred Chen, Yikai Lin, Chao Kong, and Z. Morley Mao. 2017. Open Doors for Bob and Mallory: Open Port Usage in Android Apps and Security Implications. In *Proc. EuroS&P*.
[11] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proc. ICSE*. 280–291.
[12] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z Morley Mao. 2016. The Misuse of Android Unix Domain Sockets and Security Implications. In *Proc. CCS*. 80–91.
[13] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. CCS*. 1329–1341.