Are Mobile Banking Apps Secure? What Can Be Improved?

Sen Chen[1], Ting Su[1,2*], Lingling Fan[1], Guozhu Meng[4,2], Minhui Xue[2,3], Yang Liu[2], Lihua Xu[1,3*]
[1]East China Normal University, China [2]Nanyang Technological University, Singapore
[3]New York University Shanghai, China [4]Chinese Academy of Sciences, China
{ecnuchensen,tsuletgo,ecnujanefan,minhuixue}@gmail.com,{gzmeng,yangliu}@ntu.edu.sg,{lihua.xu}@nyu.edu

ABSTRACT

Mobile banking apps, as one of the most contemporary FinTechs, have been widely adopted by banking entities to provide instant financial services. However, our recent work discovered thousands of vulnerabilities in 693 banking apps, which indicates these apps are not as secure as we expected. This motivates us to conduct this study for understanding the current security status of them. First, we take 6 months to track the reporting and patching procedure of these vulnerabilities. Second, we audit 4 state-of-the-art vulnerability detection tools on those patched vulnerabilities. Third, we discuss with 7 banking entities via in-person or online meetings and conduct an online survey to gain more feedback from financial app developers. Through this study, we reveal that (1) people may have inconsistent understandings of the vulnerabilities and different criteria for rating severity; (2) state-of-the-art tools are not effective in detecting vulnerabilities that the banking entities most concern; and (3) more efforts should be endeavored in different aspects to secure banking apps. We believe our study can help bridge the existing gaps, and further motivate different parties, including banking entities, researchers and policy makers, to better tackle security issues altogether.

CCS CONCEPTS

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

KEYWORDS

Mobile Banking Apps, Vulnerability, Empirical Study

ACM Reference Format:
Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, Lihua Xu. 2018. Are Mobile Banking Apps Secure? What Can Be Improved?. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275523

1 INTRODUCTION

"There are two big opportunities in the future financial industry. One is online banking, where all the financial institutions go online; the other is internet finance, which is purely led by outsiders." — Jack Ma.

[*]Corresponding authors.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11...$15.00
https://doi.org/10.1145/3236024.3275523

A few years ago, Jack Ma argued the differences between FinTech (financial technology) and TechFin (technological finance). FinTech [13] is known as a new industry where cutting edge technologies are applied in financial services. It is used to help companies manage the financial aspects of their businesses. TechFin [10], on the other hand, is the third-party company that offers new technological solutions (e.g., security service). For example, as one of the most popular contemporary FinTechs, *mobile banking applications* (apps), have been widely adopted by banking entities to provide instant financial services (e.g., money transfer, peer-to-peer payment). Undoubtedly, securing such FinTech as banking apps is important and crucial to our interests.

However, as revealed by our recent large-scale study, *these financial services (e.g., banking apps) are not that secure in the real world* [23]. Specifically, we built an automated security analysis tool Ausera [23], and assessed 693 banking apps across over 80 countries. Ausera unveiled 2,157 vulnerabilities, many of which could cause serious sensitive data leakage (e.g., PIN code, user name).

To investigate such an alarming phenomenon, we take three steps to investigate the current security status of banking apps. First, we report 335 vulnerabilities to the 60 corresponding banking entities for confirmation, and track the patching process by scanning the new app versions. However, the reporting process is not as smooth as we expected. Due to lack of online contact information or security supporting services, we only receive a few valid responses. We keep tracking the email responses from August 1, 2017 to January 31, 2018 (6 months in total), but only receive responses from 16.67% (10/60) banking entities.

During the 6 months, 30% (18/60) banking entities patched 41.27% (52/126) of our reported vulnerabilities. On average, it takes 75 days for a bank to patch one vulnerability, which is obviously far from satisfactory. We find several reasons that account for such a long patching time: (1) banking entities may not be aware of or emphasize specific vulnerabilities in practice; (2) they do not have standard severity rating criteria to prioritize vulnerability patching; (3) they usually rely on penetration testing [19] to guarantee security, which however may not be able to detect vulnerabilities of their concerns.

Second, we explore whether the state-of-the-art security tools can uncover vulnerabilities that the banking entities most concern. To this end, we choose four representative tools often used by developers in practice, i.e., Qihoo360 [21], AndroBugs [1], and QARK [20] and MobSF [18], to assess those banking apps that have been patched according to our reports. Based on the analysis results, we find two obvious weaknesses of these tools: (1) they output a large number of false positives due to the syntax-based checking strategies; (2) they may fail to handle different malicious scenarios due to the same vulnerability (e.g., using invalid server verification).

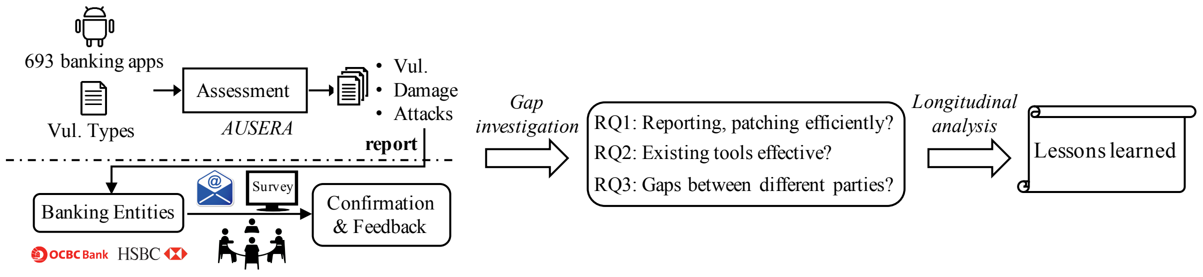


Figure 1: Workflow of our study

Third, we set up several in-person or online meetings with the banking entities from UK, India, China, Singapore, and HongKong, e.g., *HSBC, OCBC, DBS, and BHIM*, to understand the policies they follow. We surprisingly find banking entities may have inconsistent understandings with academic researchers or even with themselves. They emphasize more on data leakage-related and invalid authentication vulnerabilities than the other types. Furthermore, we conduct an online survey for app developers to familiarize with their security considerations and vulnerability understandings during development. We find developers may choose insecure APIs or implementation methods due to their preference or expertise. Based on the experience, we distill several lessons and recommendations for different parties, including banking entities, researchers and policy makers, to better secure banking apps.

In summary, we make the following contributions:

- We conduct the first study to track and investigate the vulnerability patching process of mobile banking apps, and communicate with banking entities on the discovered vulnerabilities by our security risk assessment tool AUSERA.
- We unveil several problems in industrial practice, and also evaluate the abilities of the state-of-the-art vulnerability detection tools on the patched vulnerabilities that the banking entities most concern.
- We distill several lessons and recommendations from our investigation, and show the gaps that banking entities, researchers and policy makers that need to close.

We believe our study can provide useful insights for different parties to secure mobile banking apps. We here also emphasize that our study has received positive feedback from the banking entities in question and already driven several improvements in their policies, as well as our tool, AUSERA, has done its upmost for boosting their products' quality.

2 STUDY DESIGN

In this section, we first briefly introduce the workflow of our study and summarize three research questions we aim to answer in this paper. Next, we introduce the procedure of vulnerability collection. Last, we give a real vulnerability case to show its damage and briefly illustrate how AUSERA can detect it.

2.1 Workflow of Our Study

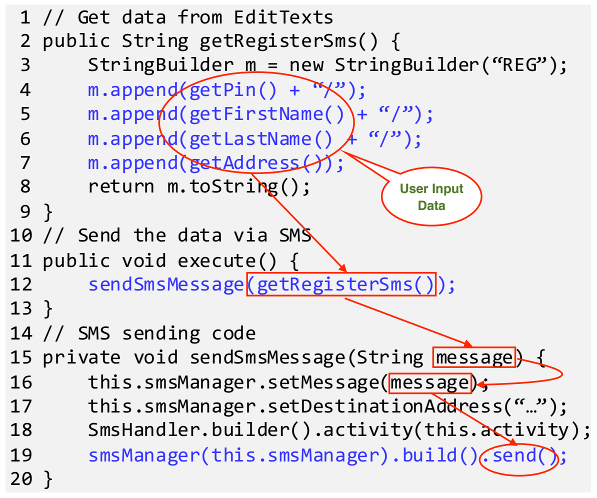
As shown in Figure 1, our study contains 4 phases:
(i) **Vulnerability collection**. We apply our automated security risk assessment tool, AUSERA, on 693 mobile banking apps to collect vulnerabilities. Specifically, AUSERA employs two strategies to detect vulnerabilities in mobile banking apps: a *forward data flow analysis* to determine whether there exists sensitive data flowing into insecure sinks (e.g., `sendTextMessage`); a *backward control flow analysis* to check whether the vulnerable API invocation patterns in communication infrastructure are truly reachable (e.g., invalid server authentication). After assessing all banking apps, we manually confirm vulnerabilities by inspecting the corresponding vulnerable code. We adopt several reverse-engineering and analysis tools (e.g., APKTOOL [5], JADX-GUI [11]) to ease our analysis. The manual analysis takes us about one and half a month, including the demonstration of how to exploit several severe vulnerabilities to get users' sensitive data.
(ii) **Progress tracking**. We track the procedure of vulnerability reporting and patching via emails and in-person meetings. Specifically, we send the detected vulnerabilities, potential damages and attacks to the banking entities, and set up in-person meetings if necessary. Further, we conduct an online survey[1] to collect feedback from app developers about their security consideration during development and understanding of vulnerabilities. To achieve this, we crawled the email addresses of over 2.5k Android app developers and invite them to participate the survey. All of them have the experience of developing apps in the "FIANANCE" category on Google Play. We have received 20 responses in total until now.
(iii) **Gap investigation**. We compile the feedbacks from banking entities and industrial developers, observe the problems during the vulnerability reporting and patching procedure, and investigate the security gaps between different parties.
(iv) **Longitudinal analysis**. We summarize the lessons learned from our study. We also provide some recommendations to banking entities, academic researchers and policy makers on how to shorten the vulnerability patching time and bridge existing gaps.

2.2 Research Questions

In this paper, we aim to answer the following research questions by tracking the procedure of vulnerability reporting and patching.

- ***RQ1***: If some vulnerabilities were found in the mobile banking apps, can we efficiently reach the corresponding banking entities? Can the vulnerabilities be efficiently patched?
- ***RQ2***: Are the state-of-the-art security analysis tool effective for detecting the vulnerabilities that the banking entities concern?
- ***RQ3***: What are the differences of understanding vulnerabilities and rating the severity between different parties, e.g., banking entities, third-party security companies and researchers?

[1]https://goo.gl/forms/5XWSzuBqA4G4aY6E3



```
1 // Get data from EditTexts
2 public String getRegisterSms() {
3     StringBuilder m = new StringBuilder("REG");
4     m.append(getPin() + "/");
5     m.append(getFirstName() + "/");
6     m.append(getLastName() + "/");
7     m.append(getAddress());
8     return m.toString();
9 }
10 // Send the data via SMS
11 public void execute() {
12     sendSmsMessage(getRegisterSms());
13 }
14 // SMS sending code
15 private void sendSmsMessage(String message) {
16     this.smsManager.setMessage(message);
17     this.smsManager.setDestinationAddress("…");
18     SmsHandler.builder().activity(this.activity);
19     smsManager(this.smsManager).build().send();
20 }
```

Figure 2: Pseudo code of banking app that leaks credentials

2.3 Vulnerability Collection

We collected 693 mobile banking apps from Google Play, which cover more than 80 developed and developing countries across 5 continents. We then apply Ausera to identify vulnerabilities. Ausera has integrated 16 vulnerability types [23] in total. These types are collected from prior research [26], best industrial practice guidelines (e.g., Google Android Best Practice [15] and OWASP), online secure reports, and security weakness and vulnerability databases (e.g., CWE [9], CVE [8]). Finally, we collected 2,157 vulnerabilities in total, 592 of which are related with private data leakage (affecting 470 banking apps), such as *Preference Leakage, Logging Leakage, SMS Leakage, and SD Card Leakage*. Additionally, invalid authentication is also more likely to be exploited (affecting 222 apps), but always gains little attention from app developers.

2.4 A Real Vulnerable Case

To better understand what a data-related vulnerability is and how Ausera detects it, we give a real vulnerable case. The case is taken from the app of a famous banking entity in Southeast Asia with 10M-50M downloads on Google Play. It discloses users' credentials and may cause severe financial loss. Specifically, after users register by entering their names, passwords and addresses, the app packages users' credentials into an SMS message and sends them to the bank server via `ServiceManager` on Android. It is extremely astonishing that the SMS message with credential data is also stored into the outbox. Assume the user device runs a daemon service of a malicious app at the backend, which could extract the credentials inside the outbox and thereby send out to its remote server controlled by a cybercriminal. Consequently, the cybercriminal is able to use the stolen credentials to access the victim's account.

Figure 2 presents the pseudo code of this vulnerability. The confidential data is extracted from the `EditTexts` (Lines 4-7); the method `execute()` is called to send a message (Line 12) containing the extracted confidential data, and `SmsManager` sends the credentials (Line 19). To detect this leakage, Ausera first tags sensitive data by applying NLP techniques, and then uses control- and data-flow analysis to track the data flow (the red arrows in Figure 2 indicate the flow of the user input data) until it sends out the data. Ausera extracts the leakage path of user inputs and validates the app does send sensitive user data via SMS.

Table 1: Tracking information of 7 banking entities we have successfully got in touch with till now.

Banking Apps	#Email	Duration (days)	In-person meeting	Country	Download
HSBC	20	30	02/17/2018	UK & China	5M-10M
OCBC	7	45	01/12/2018	Singapore	5M-10M
DBS	14	35	01/15/2018	Singapore	5M-10M
MyAadhar	15	42	11/03/2017	India	50M-100M
BHIM	18	42	11/03/2017	India	10M-50M
ICICI Netbanking	20	47	11/19/2017	India	100M-500M
ICICI Pockets	20	47	11/19/2017	India	50M-100M

3 SECURITY STATUS QUO

3.1 Vulnerability Reporting

When reporting vulnerabilities, we find many banking entities do not provide contact information or security supporting service on their official websites. As a result, we have to contact them by using the emails of app developers found on Google Play. This phenomenon indicates that banking entities have not pay enough attention to the security of their apps. They do not provide an efficient channel or a bounty program to receive security feedback from third-parties (e.g., users, researchers).

We contacted 60 banking entities by sending the corresponding vulnerabilities. However, we only received responses, including 114 emails in total, from 10 banking entities. Some banking entities gave us positive responses, which confirmed the reported vulnerabilities. They would like to cooperate with us to patch the reported vulnerabilities in their new versions. However, as shown in Table 1, 7 out of 10 banking entities gave us real replies, while the other 3 only gave auto-replies, saying that:

- *"We proceeded to forward your request to the competent office; in case the proposal will be positively evaluated, you will be contacted at the addresses that you kindly provided us."*
- *"Your ticket number is 553331, Our Support team are currently reviewing your concern. Please expect a response from us as soon as possible."*

We explained the reported vulnerabilities in detail to the 7 banking entities via email, and sent 16 emails on average to each of them. But this communication method is proved to be time-consuming and ineffective. Therefore, we decided to set up in-person or online meetings with these banking entities. We successfully set up the meetings with 7 banking entities. In each meeting, the number of participants is more than 10, and the duration is more than 2 hours. At least three people from our research team participated in the meeting. We extensively discussed the reported vulnerabilities in these banking apps, and exchanged our idea on how to ensure app security and other related topics. In particular, we visited some banking entities from Singapore and China in person. More results are discussed in Section 4.

Table 2: Tracking the status of vulnerability patching

Types	#Apps	#Patched Apps	#Vuls	#Patched Vuls	Patching Rate	#New Vuls
Actively responded	10	7	56	24	43%	0
Silently patched	11	11	70	28	40%	5
Total	21	18	126	52	41.27%	5

3.2 Vulnerability Patching

As shown in Table 2, we find that 21 out of 60 banking entities have confirmed the reported vulnerabilities in two different ways. 10 banking entities confirm the vulnerabilities by emailing us the positive feedback. Specifically, 7 of 10 have fixed the vulnerabilities, and the rest 3 claim they will patch the vulnerabilities soon in the next release versions but need our assistance. However, another 11 banking entities confirm the vulnerabilities by silent patching — we haven't received any responses from them, but they indeed patch the corresponding vulnerabilities in their new versions. To further confirm the patching status, we recheck the latest release versions of these 60 banking apps, and find 18 banking entities have already patched 52 vulnerabilities in their apps according to our vulnerability reports.

There are 126 vulnerabilities detected in the 21 banking apps. However, the patching rate of the reported vulnerabilities is low either for banking entities with "Actively responded" (43%) or with "Silently patched" (40%). We explore the reasons behind in Section 4. On the other hand, during the patching procedure, 5 new vulnerabilities are introduced in new versions. It indicates banking entities do not have effective security assessment mechanism for their apps. For example, the banking app (*C*∗) has 6 vulnerabilities. In its updated version, two reported vulnerabilities (i.e., Only Uses HTTP Protocol and Logging Leakage) are patched by employing `SSL/TLS` over `HTTPS` communication. However, new vulnerabilities are introduced, i.e., the app does not verify the identity of bank server when using `SSL/TLS`.

3.3 Auditing Vulnerability Detection Tools

According to the discussion with banking entities, we find the top priority of banking entities is to patch vulnerabilities related to data leakage and invalid authentication. Among the 52 patched cases from 18 banking apps, 29 cases are relevant to data leakage, and 6 cases are using invalid authentication. Banking entities patched 67.31% of these two types of vulnerabilities. Based on the observation, we intend to investigate whether the state-of-the-art security tools can effectively detect such vulnerabilities.

We apply 4 state-of-the-art industrial and academic tools, Qihoo360, AndroBugs, QARK and MobSF on the 18 patched banking apps to investigate their abilities of detecting data leakage-related and invalid authentication vulnerabilities. Qihoo360 and AndroBugs are the representative tools in industry and academy, respectively. QARK and MobSF are both open-source tools and often used in practice according to our online survey. In Table 3, we show 7 different types of data leakage-related vulnerabilities (e.g., Preference Leakage, SMS Leakge, and Screenshot) and invalid authentication. We tick the types of vulnerabilities these tools can detect. We can see all tools can handle invalid authentication via syntax- or pattern-based checking methods, but we find they can be negatively affected by the existence of dead code (incurring false positives).

On the other hand, these tools cannot detect most of data leakage-related vulnerabilities. Only MobSF and AndroBugs can detect sensitive data disclosure through logging, encryption key hard-code and screenshot. MobSF achieves this by simply matching such APIs as `Log.e()`, `Log.d()` and `Log.v()`. However, it may incur a number of false positives, since it is very common for developers to output some insensitive information of their apps via `Log.d()` to ease debugging. We find MobSF reports 167 logging-related cases for the banking app *GCash*, 165 of which are all false positives. Meanwhile, it reveals 17 false positives related to encryption key hard-coded, but they only indicate which Java files are relevant to encryption functions, such as `Cipher` objects. Actually, they use "Files **MAY** contain hard-coded keys." as a declaration for their detected security cases, which is not actionable for patching. Moreover, we find these tools cannot detect all the vulnerabilities that they declare to be able to detect. For example, QARK and Qihoo360 cannot detect all the 6 vulnerabilities of invalid authentication.

QARK is often used by developers or third-party security companies to guarantee app security according to our online questionnaire. However, as shown in Table 3, it cannot detect data leakage-related vulnerabilities that banking entities most concern. In addition, developers also use the tools like Xposed [22] and Frida [14] to assist their security assessment for apps. Xposed, a hook framework, can change the original behaviors of the system and apps. Frida, a dynamic instrumentation toolkit, can inject scripts to explore native apps on Android. Strikingly, several developers rely on functional testing tool like JUnit [17] to guarantee app security.

4 LESSONS LEARNED

This section summarizes the lessons learned from our study, including 114 email exchanges, 7 in-person meetings, and 20 valid online questionnaires. It helps us understand the gaps between different parties, including banking entities, researchers, and third-party security companies.

4.1 Different Understanding of Vulnerabilities

Understanding of reported vulnerabilities between banking entities and researchers. From the view of academic researchers, we usually recommend all of the reported vulnerabilities should be patched as soon as possible. However, from the view of industrial practice, banking entities think that only parts of them are vulnerable and should be patched. We provide three cases to demonstrate this divergence as follows.

- Hash functions `MD5` and `SHA1` have been proved insecure many years ago [27]. However, they are still widely used in app development. The online questionnaire unveils that 60% developers are still using `MD5` or `SHA1` for cryptographic use (e.g., randomization and integrity validation), although they have been aware of the weakness. In addition, some banking entities don't care whether the imported third-party libraries use the message digest even if they are aware of it. "In case the instances (i.e., vulnerabilities) are in the third party libraries we do not consider such instances as these do not fall within the scope of the assessments." as the team from *BHIM* and *MyAadhar* said in the email.

Table 3: Patched vulnerability types that banking entities concern.

Concerned	Preference Leakage	Logging Leakage	SD Card Leakage	WebView Leakage	SMS Leakage	Encryption Key Hard-coded	Screenshot	Invalid Authentication
#Patched Vuls	4	7	4	3	2	2	7	6
Qihoo360 [21]	-	-		-	-	-	-	✓
AndroBugs [1]	-	-	-	-	-	-	✓	✓
QARK [20]	-	-	-	-	-	-	-	✓
MobSF [18]	-	✓	-	-	-	✓	-	✓

- Logging sensitive data is very common in industry. Some sensitive data is leaked as debugging outputs, such as payee name, transfer money, and transfer time. According to the results from online questionnaires, we find there are still 50% of developers logging sensitive data in their delivered apps. Some banking entities (e.g., *ICICI Netbanking* and *Pockets*) assume "if the minimum SDK is 16, no application can read the logs of other applications even installed on the same device, which is a secure environment." But this is not always true, especially when the devices have been rooted or under privilege escalation attacks. The appropriate mitigation is to remove these debugging outputs before release, which has been widely acknowledged in academia [26].
- The last one is invalid authentication for banking apps. Some banking entities (e.g., *HSBC China*) used two types of authentication before login and after login, respectively. The problematic authentication phase is before login: before logging in the banking app, they use an invalid authentication to set up the communication between the app and remote server. That is, they do nothing in the function of `checkServerTrusted` on invalid certificates, it is at risk for users' data and behaviors.

Understanding of same vulnerabilities between different banking entities. Actually, as for a certain vulnerability, different banking entities may hold different understandings. Here, we give a real case to explain the phenomenon. We regard screenshot as a vulnerable behavior for banking apps, because malicious apps can extract users' credentials and other sensitive data (e.g., transfer data) via screenshots. The developers in *OCBC* from Singapore thought "screenshot is a basic function for banking users, since users may want to share information to their friends". However, the developers in *BHIM* from India confirmed the damage of screenshots although they don't know how to patch it at first. The results from online questionnaire show that 75% developers think allowing screenshot in a security-related app is vulnerable, however, the rest of developers hold the opposite opinion. We don't think that screenshots of all app pages have damage, but it is at least vulnerable to sensitive pages (e.g., pages of login, registration and transfer). To avoid this screenshot attack, developers should set `WindowManger.LayoutParams.FLAG_SECURE` by calling `getWindow()#setFlags` in the page that should be protected.

4.2 Severity Criteria and Concerns

Severity criteria of reported vulnerabilities. Due to high vulnerability patching cost (time and human efforts), banking entities prefer to patch the vulnerabilities of high risks. Specifically, *OCBC* and *DBS* rely on CVSS [7] to determine the severity of found vulnerabilities, while the other 5 banking entities check whether the vulnerabilities belong to sensitive data leakage. If so, such vulnerabilities are considered of high risks. But the standard of sensitive data leakage is too coarse-grained since the definition of data is unclear, and the criterion CVSS is not that complete and perfect. Actually, there is also no standard criteria for vulnerabilities in academia. But researchers prefer security metrics, such as the severity of damage, the cost of exploits, and CVEs, to decide the severity of vulnerabilities.

Concerns of banking entities about vulnerabilities. Banking entities do concern about reported security issues: "HSBC *takes the security of its customers, systems and data very seriously and we are always interested in any security issues raised by users and researchers of our web sites and welcome any relevant information that you may have."*, said by *HSBC* security team from the UK in the email. According to the patching results, received emails, and in-person meeting, we find all banking entities concern more about sensitive data leakage and invalid authentication vulnerabilities, such as `SMS` and `Preference` leakage.

4.3 Collaboration and Responsibility

Collaboration mechanism between researchers, banking entities and third-party security companies. When we first reported the vulnerabilities to banking entities, their developers cannot patch them directly due to lack of enough security knowledge. Banking entities handed them over to third-party companies for seeking further confirmations and supports. It was stated that the third-party companies used penetration testing (e.g., QARK [20], Frida [14] and Drozer [12]) to check the banking apps before delivery. However, penetration testing cannot fully guarantee the security of apps since it relies on manual analysis. The whole process of security check is transparent to banking entities, and that is why they have to resort to third-party companies for confirmation of newly discovered vulnerabilities. Actually, the vulnerability patching time includes the communication time and actual fixing time, which explains the long patching duration. Therefore, the collaboration between researchers, banking entities and third-parity companies is not very efficient.

Who should be responsible? In some cases, banking entities refused to take the responsibility of introducing vulnerabilities. One real case is as follows. One banking app stores user name and password to an `XML` file via `SharedPreference`, an internal storage. If a malicious app is installed on the same device, users' sensitive data can be stolen easily. However, some banking entities think users should be responsible for such risk, since they should not download and install malicious apps on their devices for whatever reasons.

In other cases, the banking entities think the third-party security companies should be responsible. For example, the developers use `setSeed` when implementing `SecureRandom`, which is vulnerable due to lack of randomness. They require third-party companies or their own security team to add the corresponding detection methods when inspecting newer versions. Additionally, in some cases, they only recognize the severity after we report the vulnerabilities (e.g., invalid authentication before login). Under this situation, it is the banking entity itself that should be responsible for the vulnerability.

4.4 Recommendations

What can be improved for banking entities?

- Banking entities should provide various channels to respond to vulnerabilities, especially for the researchers who are focusing on the relevant research topics. For example, they can provide the contact information of security supporting service on the prominent position of their company websites.
- App packing [28] is one of the effective protection methods for banking apps, such as ApkProtect [4], Bangcle [6], and Ijiami [16]. It can significantly complement penetration testing. Even if one app has vulnerabilities, app packing can increase the difficulty of exploitation.
- Banking entities should pay more attention to security issues rather than only functional bugs [24, 25]. It is better to build a security team by themselves or work with third-party security companies, e.g., Alibaba Juanquan [3] and Ijiami [16].
- Banking entities should carefully choose third-party libraries, and evaluate the risk imposed by the incorporation of these libraries.

What can be improved for academic researchers?

- Researchers should pay more attention on the research topics of Android applications which have practical scenarios in industry.
- When reporting vulnerabilities, researchers are advised to provide more information such as the potential damage, exploitation methods, and patching methods. It makes banking entities instantly grasp these vulnerabilities so as to take duly measures.
- According to our online questionnaires, we summarize the vulnerability types that should receive more attention. The results show that 78% of developers concern more about sensitive data-related issues (e.g., privacy of users, info leaks, database), which need further study and exploration.

What can be improved for policy makers?

- Explicitly designating the responsibility of each party can effectively improve vulnerability patching and security awareness. GDPR [2] (General Data Protection Regulation EU) already made the first step.
- The policy makers should help standardize the third-party reporting channels.

5 CONCLUDING REMARKS

5.1 Impact of Our Study

The security status quo of the 7 banking entities that closely communicated with us has been changing in different aspects. (1) They have accepted our reported vulnerabilities and actively collaborated with us to improve the app security. In addition, the core developers explored and learned the techniques used in Ausera from us. (2) They asked us for help to conduct an automated security assessment for their new versions with Ausera before release. (3) They further explored penetration testing from third-party companies to shorten the patching duration, such as the *OCBC* team. (4) They took charge of the outdated versions in the wild by forcing updates to reduce the impact of our reported vulnerabilities. (5) Some secure hints are pushed to mobile users frequently by banking entities, such as the banking app from *DBS*.

5.2 Conclusion

This paper represents the first study to track the reporting and patching process of vulnerabilities in mobile banking apps, which lasts more than 6 months. We received numerous positive feedback from banking entities on the reported vulnerabilities. We also audited the state-of-the-art vulnerability detection tools for those confirmed vulnerabilities. Finally, we provided many practical recommendations for different parties. In all, our study provides useful insights to bridge the security gaps, and also helps banking entities and third-party companies to better tackle security issues.

ACKNOWLEDGMENTS

We appreciate the reviewers' constructive feedback. This work is partially supported by NSFC Grant 61502170, the Science and Technology Commission of Shanghai Municipality Grant 18511103802, NTU Research Grant NGF-2017-03-033 and NRF Grant CRDCG2017-S04. Sen Chen is partly supported by ECNU Project of Funding Overseas Short-term Studies.

REFERENCES

[1] 2015. AndroBugs. https://github.com/AndroBugs/
[2] 2016. General Data Protection Regulation. https://www.eugdpr.org/
[3] 2018. Alibaba Juanquan. https://angel.co/projects/112838-alibaba-juanquan
[4] 2018. APKProtect. https://github.com/CvvT/ApkProtect
[5] 2018. Apktool. https://ibotpeaches.github.io/Apktool/
[6] 2018. Bangcle Security (bangbang). https://www.crunchbase.com/organization/bangbang-security
[7] 2018. Common Vulnerability Scoring System. https://www.first.org/cvss/
[8] 2018. CVE: Common Vulnerabilities and Exposures. https://cve.mitre.org/
[9] 2018. CWE: Common Weakness Enumeration. https://cwe.mitre.org/
[10] 2018. Daily Fintech: From Fintech to TechFin. https://dailyfintech.com/2018/03/16/from-fintech-to-techfin-three-trends-that-banks-will-be-worried-about/
[11] 2018. Dex to Java decompiler. https://github.com/skylot/jadx
[12] 2018. Drozer. https://github.com/mwrlabs/drozer
[13] 2018. Financial Technology. https://en.wikipedia.org/wiki/Financial_technology
[14] 2018. Frida. https://github.com/frida
[15] 2018. Google Best Practices for Security. https://developer.android.com/training/best-security.html
[16] 2018. Ijiami. http://www.ijiami.cn/enindex
[17] 2018. JUnit. https://fr.wikipedia.org/wiki/JUnit
[18] 2018. MobSF. https://github.com/MobSF/Mobile-Security-Framework-MobSF
[19] 2018. Penetration Test. https://en.wikipedia.org/wiki/Penetration_test
[20] 2018. QARK. https://github.com/linkedin/qark
[21] 2018. Qihoo360 (Appscan). http://appscan.360.cn/
[22] 2018. Xposed. http://repo.xposed.info/module/de.robv.android.xposed.installer
[23] Sen Chen, Guozhu Meng, Ting Su, Lingling Fan, Yinxing Xue, Yang Liu, Lihua Xu, Minhui Xue, Bo Li, and Shuang Hao. 2018. AUSERA: Large-Scale Automated Security Risk Assessment of Global Mobile Banking Apps. *arXiv preprint arXiv:1805.05236* (2018).
[24] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* 486–497.
[25] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* 408–419.
[26] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. 2015. Mo (bile) Money, Mo (bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *USENIX Security.* 17–32.
[27] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Eurocrypt*, Vol. 3494. Springer, 19–35.
[28] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on.* IEEE, 358–369.