2018 42nd IEEE International Conference on Computer Software & Applications

A Security Assessment of HCE-NFC enabled E-Wallet Banking Android Apps

Ratinder Kaur, Yan Li, Junaid Iqbal, Hugo Gonzalez, Natalia Stakhanova
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
Email: {rkaur1, liyan.168, junaid.iqbal, hugo.gonzalez, natalia.stakhanova}@unb.ca

***Abstract*—E-wallets have started to grow in popularity, reaching a tipping point in some countries. This can be attributed to the worldwide use of payment-enabled devices and ubiquity of e-wallet acceptance by larger and smaller retailers. As more customers adopt e-wallets they may also become a big target of cybercrime. E-wallets facilitates financial transactions via smartphones which is a lucrative opportunity for cybercriminals. This paper presents a security assessment of the Android e-wallet apps of several Canadian leading banks.**

***Keywords*-E-Wallets; HCE-NFC Technology; Mobile Payments; Mobile Security; Android**

I. Introduction

The traditional smart card banking services are now moving to smartphones. This transition is facilitated by Near Field Communication (NFC) technology which enables a smartphone to emulate credit card using e-wallets apps. This emulation is typically provided through a hardware Secure Element (SE) or using Host Card Emulation (HCE). SE mobile payment solution relies on highly secure tamper-resistant chip, which stores credit card data locally on a device in encrypted form. HCE solutions on the other hand use only software and allow NFC applications to host data on the cloud. In this context, we define *e-wallet* as a software application that facilitates commercial transactions (over NFC channel) by storing customer's credit card information and other payment data either on device or on the cloud.

It is obvious that the nature of the e-wallet mobile apps requires stronger security as compared to other apps on the smartphone. While there are no specific regulations that guide the development of banking mobile apps, various financial institutions often provide advice to their customers on how to secure their transactions. In 2012, Canadian Bankers Association (CBA) published the NFC Mobile Payments Reference Model that offered a set of voluntary guidelines for mobile payments. In 2015, a more specialized guide titled "Mobile Payments Security White Paper" was offered to banking community [1].

As mobile payments started gaining popularity they also attracted attention of cybercriminals. In 2017, Trend Micro [2] spotted a unique increase of 94% in mobile banking malware samples. These samples are built in with more sophisticated capabilities to stay under the radar. They are often blended with legitimate processes or impersonate as one, to steal more than just a credit card data, and bypass security mechanisms.

In general security community only recently started focusing on risks associated with mobile payment systems as a whole. The main focus is to defend against transaction fraud in payment networks or to safeguard against NFC attacks. Unfortunately, the security efforts seem dispersed and it looks like the basic security of mobile payment apps is being often overlooked. The security of financial data on mobile devices is one of the major concerns and motivated by this fact, some researchers conducted security assessment of different mobile payments apps (country specific), and found that these apps are not developed with sufficient security in mind.

In this paper we propose a set of security recommendations for e-wallet apps. We focus specifically on Android platform. According to F-secure's estimation, 99% of all existing mobile malware target Android devices [3]. This is primarily due to the open nature of Android platform for app development. The proposed security recommendations are based on the security guidelines provided by CBA [1] and the OWASP Top 10 Mobile Risks[1]. We further assess the security of e-wallet banking apps from Canada's leading banks according to the proposed guidelines. All assessed app are found to be vulnerable in our analysis.

The reminder of the paper is organized as follows: Section 2 describes the related work; Section 3 discusses the security rules obtained for the analysis; Section 4 explains the proposed methodology and the dataset used; Section 5 highlights the findings of security assessment conducted; and Section 6 provides concluding remarks.

II. Related Work

This section briefly reviews the existing security assessments conducted on different types of mobile payment apps in different countries. Filiol and Irolla [4] performed static and dynamic analysis on 50 Android mobile banking apps. For static analysis a prototype is developed that detects malware by exploring behavioral similarities with known

[1]https://www.owasp.org/index.php/OWASP_Mobile_Security_Project

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.10282

IEEE computer society

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 26,2021 at 22:18:20 UTC from IEEE Xplore. Restrictions apply.

malware based on app characteristics like permissions, calls to APIs, entry points, malicious character strings, opcodes, etc. For dynamic analysis network communications are inspected during runtime. They reported that almost all apps endanger user privacy. Reaves et al. [5] performed an extensive manual analysis on seven Android mobile money apps - Airtel Money, mPay, Oxigen Wallet, GCash, Zuum, MOM, and mCoin. They checked these apps for SSL/TLS vulnerabilities, cryptography, access control and information leakage. To identify SSL/TLS vulnerabilities authors coupled their manual analysis results with Mallodroid[2] and Qualys SSL Server Test[3] tools. Chothia et al. [6] reported a security review of the mobile apps provided by the UK's leading banks. The authors analyzed Android and iOS apps from 15 UK banks. Their main focus was to identify the vulnerabilities in TLS implementations of these apps. They reported that the apps do not implement certificate pinning and hostname verification techniques correctly, and thus are vulnerable to man-in-the-middle and phishing attacks. Kim et al. [7] focused on application security, investigating runtime integrity protection mechanisms used in 76 financial Android apps in the Republic of Korea. The authors developed a tool named MERCIDroid to extract a mini call graph from the runtime API trace record to locate self-defense mechanism used by the apps for their protection. The API calls extracted belongs to the code used to check device rooting, app integrity and to stop an app's execution. Authors reported that self-defense mechanism employed in Android financial apps are not effective and could be easily bypassed. On the other hand, Nguyen-Vu et al. [8] paid attention to device security and investigated different methods to detect rooted device at both Java and native code level. They tested 18 banking and finance mobile apps in Korea. They found that 10 out of 18 apps could be evaded easily by just static file renaming. The apps were only using native code to check the existence of *su* binary file and other system package names. This check was bypassed by trivial renaming technique and without the need of intercepting any native calls. Yang et al. [9] examined mobile apps in China that embed third-party payment functionality. The authors studied four mainstream third-party mobile payment cashiers. The investigation revealed that third-party SDKs are improperly designed and have vulnerable sample codes. Rajchada et al. [10] conducted forensic analysis and security assessment of seven Android mobile banking apps in Thailand. The researchers found that the apps analyzed leaked personal user information like account number, account balance, citizenID, date of birth, user PIN code etc. Furthermore, it was reported that half of the apps investigated do not implement root device detection, do not encrypt user data and are susceptible to modification and repackaging.

[2]https://github.com/sfahl/mallodroid

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

The prior work has focused on security issues of Android mobile payment apps in different markets e.g. UK, Korea, China, Thailand. The assessment efforts were isolated, targeting, application level, device level or communication level. None of the previous studies offered a comprehensive view of security, which we focus. Moreover, this paper is the first one to assess e-wallet Canadain market. As per statistic information mobile payers across Canada are steadily becoming mobile wallet users[4]. To help mobile payment users with the transition to a mobile wallet user, the e-wallet app developers and owners should ease user concerns by showcasing proof of security and privacy incorporated in the mobile wallet. Therefore, our main goal is to evaluate the security of e-wallet banking apps available in Canadian market.

III. Security Characteristics

To perform security assessment of NFC-HCE e-wallet Android apps we first define specific sets of security rules. To start with, we extract a minimal set of characteristics from an app. This initial set defines an app as a possible e-wallet. If the main characteristics are not satisfied, we discard that app. Here we mainly look for the use of NFC-HCE permission. Then we extract rest of the characteristics to determine whether or not the e-wallet app complies to the best security practices. The proposed security sets and rules that forms the core of our assessment are obtained from the security recommendations provided by CBA and OWASP Top 10 Mobile Risks.

As per the whitepaper [1] by CBA, a secure HCE solution requires multiple security layers. A layered security design makes it difficult for the attacker to steal tokens from the device and reuse them once they're stolen. There are four security layers: application security, device security, communications security and dynamic data. First three layers are essential to protect the storage of tokens on the device and prevent token theft. In the event of token theft, dynamic data is critical to prevent tokens from being used and/or limiting how they may be used.

- *Application Security*: Ensures that an e-wallet app has not been compromised. It includes right usage of obfuscation, white box cryptography and Trusted Execution Environment (TEE). It aims at protecting payment keys and other sensitive data stored on the device.
- *Device Security*: Ensures that the device has not been compromised. Detection mechanisms include root, debug, and emulation detection. These scenarios compromise the security of the HCE solutions by exposing areas of the device that are normally protected.
- *Communication Security*: Ensures the security and confidentiality of communication between an e-wallet app and the payment network. If communications are

[4]https://www.statista.com/statistics/621166/mobile-wallet-usage-canada/

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 26,2021 at 22:18:20 UTC from IEEE Xplore. Restrictions apply.

Table I
SECURITY RULES TO ASSES AN E-WALLET APP.

Set	Derived Security Rules
Minimal: This set of rules define an app as a possible e-wallet.	• Android 4.4 or higher • Use of NFC-HCE permission • No third-party ads libraries
Device security: This set of rules assures if a device is compromised or not.	• No rooted devices {M8} • No emulator
Application security: This set certifies whether the app is trusted or not	• Self-verification integrity {M8} • Protected app (obfuscated or packed) • No debuggable flag {M10} • No legacy versions
Communication security and Dynamic data: This set of rules ensures the confidentiality of app data in transit.	• Https enforced {M3} • Proper certificate pinned {M3} • No weak cryptography {M5} • Proper key management process {M1}, {M9}
Device Storage: This set of rules covers unintended data leakage.	• No plain text logs {M2} • No sensitive information in backups {M2}
Memory: This rule checks for the data leak during run-time.	• No sensitive information in memory

intercepted and cracked, it is possible to steal the dynamic data provisioned to the device and perform fake transactions.

- *Dynamic Data*: Ensures protection from risks associated with possible theft and interception of payment credentials. HCE uses constantly changing payment data. So the dynamic credentials could be limited by number of uses, time period, or both, before they expire.

On the other hand, OWASP Top 10 Mobile Risks (M1 to M10) provides better security guidelines to follow during mobile app development. We thus, compiled the security layers from CBA and security practices from OWASP to obtain an extensive set of rules to review the security of e-wallet apps as shown in Table I. Some of the rules are marked with OWASP Mobile Risk number(s) {M1, M2, M3, M5, M8, M9, M10} to show that they directly correspond to a specific mobile risk.

IV. PROPOSED METHODOLOGY

After obtaining the rules for evaluating the security of e-wallet apps, we come up with an analysis methodology to perform the security assessment.

A. Analysis Process

The analysis process is mainly divided into two different phases - static and dynamic as depicted in Figure 1. To begin the analysis, the e-wallet apps are executed on Android LG Nexus 5X rooted device to check if they are running without any error. Some of them didn't work because the apps identified rooted device, while others crashed as they didn't find SIM card on the device. So, we inserted a SIM card on the phone and next we need to find out a way to bypass root device checks to proceed with our security analysis.

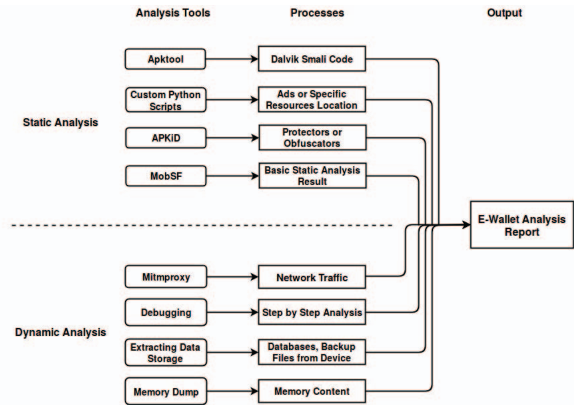

Figure 1. The Analysis Process

1) Static analysis: In static analysis phase, some existing analysis tools like apktool[5] , APKiD[6] and MobSF[7] and custom scripts are used to extract necessary information. With the help of apktool, smali code is extracted from the Android apps. From the smali code corresponding java classes and methods are found to reveal the logic of the app. APKiD is used to identify packers, protectors and obfuscators based on the pre-defined yara rules. APKiD also contains rules for anti-vm checks performed by the apps. Apart from these tools, the apps are also analysed using MobSF, a Mobile Security Framework which is an automated pen-testing framework capable of performing static and dynamic analysis. From the MobSF static analysis reports we are more interested in details from different categories, such as "java classes based on functions", "list of content" etc. This way, static analysis gave us a brief understanding about the apps.

2) Dynamic analysis: Before performing dynamic testing on the rooted device, we had to bypass some security checks done by the apps. We evaded anti-analysis defenses (for e.g., root, vm, emulator, integrity checks) to continue performing analysis. In an Android development, all layouts are connected with activities either statically or dynamically. In order to find the security functions, first apktool is run to generate the smali code and then the error message strings are searched and located in the activities with the help of custom scripts. After strings or layout content is located, the logic of app from smali code is analyzed to seek for the place where those security checking methods are called. The located security checks are then blocked by simply commenting that smali code. After modifying the smali code, apps are repackaged and resigned to run on rooted device. The dynamic analysis phase focused on four different tasks- intercepting network traffic between server and app, step by step runtime debugging, analyzing backup files and

[5]https://ibotpeaches.github.io/Apktool/
[6]https://github.com/rednaga/APKiD
[7]https://github.com/MobSF/Mobile-Security-Framework-MobSF

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 26,2021 at 22:18:20 UTC from IEEE Xplore. Restrictions apply.

memory contents. It is important to extract network traffic to check if communication between the server and app is secure or not. Mitmproxy[8] is used for this experiment. In order to analyze the apps step by step, we need to attach an Android debugger to the process that is running the app, we used Android Studio[9] to do the task. Step by step monitoring the running app gave us detailed understanding on the logic of the app. The backup files and memory contents of the app are also extracted and analyzed to locate any data leaks.

B. Dataset

The dataset comprises of e-wallet apps from large and leading banks in Canada. These types of apps are best candidates for a good case study. These are security critical apps with large user base and motivated attacker. They often use techniques to provide additional security such as two-factor authentication. We believe that unlike other apps, they have been thoroughly tested before being distributed on the Android market. So they are likely to be more secure. We crawled Google Play Market in November 2017 and filtered apps with NFC-HCE permission determined by a HostApduService manifest declaration in the Android Manifest File. The selected apps retained for analysis are[10] – (a) *Bank 1 app* is a pure e-wallet app for tap and pay at NFC POS terminals. Bank 1 a separate app solely for mobile banking. This e-wallet app is only for bank account holders. (b) *Bank 2 app* combines e-wallet tap and pay functionality in its mobile banking app and is only for bank account holders. (c) *Bank 3 app* also combines e-wallet tap and pay functionality in its mobile banking app. It is only for bank account holders. (d) *Android Pay* (now G Pay) is pure e-wallet app that accepts credit cards from any banks.

V. Findings

In this section we present and discuss the findings of security assessment conducted on the dataset.

No third-party ads libraries: An android e-wallet app dealing with financial data should be completely free from third-party ads. There is a higher probability that the third-party ads are using insecure http connections, which could expose the app to phishing attacks. Usually to include an ad in application, an ad activity is declared in AndroidManifest.xml file. To check if an e-wallet app is using some adds or not we wrote a Python script to search for ad activities in AndroidManifest.xml file. None of the banks have third-party ads.

No rooted devices: To determine if the app checks whether it is running on a rooted device or not there are many different ways. Like checking installed packages, files, configurations, directory permissions, processes, services, rooting traits using shell commands or system properties. We perform both static and dynamic checks. We looked for the use of root commands in the smali code and to confirm the fact, we just run the apps on a rooted device to find out if the app is running properly without any issues. Bank1-wallet checks for rooted device before running but we bypassed the root check by modifying the smali code. After modification the app worked well on the rooted device. Bank 2 and bank 3 apps do not perform root check, they got executed on rooted device after showing an alert- that the environment is not safe, which a user could just ignore. Android Pay uses SafetyNet to perform device checks and did not run at all on rooted phone.

No emulator: The e-wallet must be able to detect any emulated environment, so that it cannot be analyzed to reveal functionality or behavior. To identify an emulated environment the apps could use heuristics ranging from simple flag checks, unrealistic input to fingerprinting hypervisors. To spot this functionality in app, we used APKiD tool. It contains yara rules to identify anti-vm checks. For assurance we also verified manually by running all the apps on the Android Emulator. Bank 2 and Bank 3 apps do not check for VM presence. On the other hand, Bank 1-Wallet checks for ro.kernel.qemu, which we bypassed by modifying the smali code. Also, from the manual experiment we found that Bank 2 and Bank 3 apps are running fine on the Android emulator. Bank 1-Wallet did not run on emulator as the app also inspected for NFC and emulator does not come with NFC support. As expected, Android Pay checks for emulator presence and do not run on emulator.

Self-verification integrity: Self-verification integrity ensures that the app is not tempered and is the one originally distributed by the developer. There can be many ways to implement anti-temper checking capabilities in the app like verifying app's signing certificate at runtime or verifying the installer to ensure that the app is only distributed via the Google Play Store. To ensure self-verification integrity a simple tweak in smali code is done. We decompiled the app, made some changes in the smali code, re-compiled the app, signed with our own certificate and then installed it back on the device to see if it is running fine or not. Moreover, the app code changes in the process of decompilation and recompilation. Bank 1-Wallet app has self-verification check but that got easily bypassed by commenting few lines of smali code. Bank 2 and Bank 3 do not perform such checks as they worked fine after smali code modification. Android Pay has code integrity verification checks.

Protected app: If an app's binary code is not obfuscated it will be easy to inspect and/or do reverse engineering with freely available tools. To examine whether the app is obfuscated or not, we decompiled the app and look for classes and function names, if they are random like aa, bb,

[8]https://mitmproxy.org/

[9]https://developer.android.com/studio/index.html

[10]Names of financial institutions are not released due to ongoing efforts to improve app security. The apps and results of analysis can be shared upon request.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 26,2021 at 22:18:20 UTC from IEEE Xplore. Restrictions apply.

cc then the app must be obfuscated using layout obfuscation technique. We also checked the apps with APKiD yara rules. It has signatures for popular obfuscators like DexGuard, DexProtect, Bitwise AntiSkid, etc. By looking at the jumbled up variables, methods and classes names it is believed that all the apps are partially obfuscated. Unfortunately APKiD is not able to return the exact obfuscator name, potentially due to out-of-date signatures or no signature for any new obfuscator tool. Besides the apps being relatively obfuscated, we are able to grasp the app logic from the smali code and modified it according to our requirement.

Not debuggable: The e-wallet apps should not allow debuggers to get attached to perform step by step analysis through method calls. The best part of debugging is that root privilege is not required. To see if an Android app is debuggable or not, easy way is to look at the debuggable flag. The value of debuggable flag can be determined from android debuggable attribute (android:debuggable) in Manifest file. By default this flag is set to false when an apk is signed and generated in release mode. In order to debug the apps we manually set the debuggable flag in AndroidManifest.xml to true. As anticipated all the e-wallet apps have this flag set to false. For Bank 1-Wallet, we changed the value of android:debuggable flag, recompiled, installed it on unrooted phone, and received an error message. We then reset the value of flag back to false and found that the app is running fine. From this we concluded that Bank 1-Wallet has some anti-debug checks. Android Pay also shows error messages on changing the flag value. On the other hand, Bank 2 and Bank 3 apps work fine after changing flag value from false to true in the manifest file.

No legacy versions: To check this we installed the apps on old Android devices with NFC capability to see if legacy version of the app is provided. The risk with older apps is that chances are high that they may contain some unfixed errors and vulnerabilities that could be exploited. And at times fixing the bugs in legacy app version is overlooked. Bank 1-Wallet has old version of app available and it prompts user to install the latest version but the user can simply ignore the alert and continue with older version. All other apps comes with latest version.

Https enforced: Static and dynamic checks are performed to verify this feature. Firstly, we extracted urls from the apps to see if there are any HTTP urls embedded in the code. Later we captured the traffic between app and the server to check if the app downgrade its service from HTTPS to HTTP. These insecure connections could expose the apps to potential phishing attacks. Two of the apps mix HTTPS with HTTP traffic. Bank 1-Wallet has more http urls as compared to other apps. Bank 2 has some adverts for their own bank products and links to information for help. The adverts and information links are requested over the HTTP connections. Bank 3 app seems somewhat secure than Bank 1-Wallet and Bank 2 with few http connections. On the other hand, Android Pay has proper https enforced and uses https connection to request web resources. This has been confirmed from MobSF static reports.

Proper certificate pinned: We did investigation to look for TLS vulnerabilities and misuses. We set up mitmproxy on one machine, installed mitmproxy certificate on the rooted android device and then captured all data transmitted between app and the server. Bank 1-Wallet and Bank 2 apps do not implement certificate pinning. With Man-in-the-Middle (MITM) attack we could view user credentials and sensitive data being transferred between the apps and the server. The snapshots in Figure 2 and Figure 3 show data leak in network traffic for Banks 1 and 2, respectively. While Bank 3 and Android Pay use certificate pinning, so we couldn't capture traffic for them.

No weak cryptography: Here we check whether the apps implement additional cryptographic protection or they are only dependent on TLS security. The traffic between app and server is captured using mitmproxy to check HTTP POST requests containing JSON encoded data. In case of Bank 1-Wallet and Bank 2 app, the JSON data is readable, which is later encrypted by the app to store in the database. The snapshots 2d and 3c show clear plain text values (sensitive user info is blurred) in JSON format. Hence, these apps are not using any second layer of encoding/encryption to hide sensitive data being transferred.

Proper key management: We mainly focus on key storage as assessing entire key management process involves some testing at server-end, which is not in the scope of this assessment. We looked for cryptographic keys stored within the app i.e. in some key management libraries or local databases. Ideally, the keys should not be in plaintext format and should be stored isolated from the app data. All the three apps use Android keystore function i.e. they are storing cryptographic keys in a secure system level storage isolated from app data. But Android Pay do not use Android Keystore System.

No sensitive information in backups: We extracted the backup files from the rooted device and copied that on the system using Android Debug Bridge(adb). The database is then viewed and analyzed using SQLiteManager[11]. Banks 1, 2 and 3 apps have encrypted tables. If they are not fully encrypted then card numbers and keys are scrambled. No suspicious logs are identified in the backup files. Complete backup is not retrieved in case of Bank 3 as we did not have valid account to login, and no backup for Android Pay was captured as it didn't run on rooted device.

No sensitive information on memory: For capturing sensitive plain text data from the memory we used Fridump[12]. This tool requires rooted device to run frida server in order to dump the memory of a running process. Memory strings

[11] http://www.sqlabs.com/sqlitemanager.php
[12] https://github.com/Nightbringer21/fridump

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 26,2021 at 22:18:20 UTC from IEEE Xplore. Restrictions apply.

```
</ecats>
<client>
  <identity type="username"</identity>
  <identity type="cardNumber"></identity>
</client>
<pvq>
```


(a)


```
<pvqAnswer>
  <![CDATA[rbcxcq0_answer&]]>
</pvqAnswer>
<username>
```


(b)


```
<accountList>
  <RBCAccount>
    <id></id>
    <name>Signature RBC Rewards VISA</name>
    <accountNumber></accountNumber>
    <typeName>CL2</typeName>
  </RBCAccount>
</accountList>
```


(c)


```
{
    "active": true,
    "cards": [
        {
            "cardId": "437450",
            "cardRef": "1000416536",
            "cardState": "Active",
            "cardType": "HCE_IAC",
            "maskedPan": "",
            "productCode": "MDA",
            "psn": "00"
        },
        {
            "cardId": "438041",
            "cardRef": "1000429152",
            "cardState": "Personalized",
            "cardType": "HCE_VS_CVN43",
            "hashedPlasticPan": "",
            "maskedPan": "",
            "maskedPlasticPan": "",
            "productCode": "CL2",
            "psn": "06"
        }
    ],
    "ownerId": "",
    "ownerState": "New",
    "status": "OK",
    "type": "ListCardsResponse"
```


(d)

Figure 2. Bank 1-Wallet App Data Leak: (a) Login details, (b) Security Answer, (c) Account Number (d) Card Details


```
  "authenticationIdentity": {
      "accessAuthenticationIdentifierId": "",
      "accessAuthenticationIdentifierTypeCd": "loginid",
      "authenticationMethodSubtypeCd": "credential",
      "authenticationMethodTypeCd": "KBF",
      "secretVal": ""
```


(a)


```
{
    "authenticationIdentity": {
        "accessAuthenticationIdentifierTypeCd": "challenge",
        "authenticationMethodSubtypeCd": "challenge",
        "authenticationMethodTypeCd": "KBF",
        "secretVal": ""
    },
```


(b)


```
{
    "MobileDevice": {
        "AppVersionNo": "2.0",
        "DeviceName": "RGV2aWNlTmFtZTEyMzQ",
        "DeviceType": "PHONE",
        "Eligible": {
            "ReasonCode": "",
            "Status": true
        },
        "ICCID": "89302370313020417971",
        "IMEI": "",
        "ModelName": "SM-G903W",
        "ModelNo": "NRD90M.G903WVLU1CQH4",
        "OEMName": "samsung",
        "OSPlatformCd": "Android",
        "OSVersionNo": "7.0",
        "PhoneNumber": ""
    },
    "SvcCredentialRq": {
        "Credential": {
            "AlternateCredId": "VPM4CWA5MU#9XA7GVZQB",
            "CustomerName"",
            "ExpiryDate": "",
            "PAN": "",
            "Purpose": "Payment",
            "State": "ACTIVE",
            "SubTypes": [
                "VISA"
            ],
```


(c)

Figure 3. Bank 2 App Data Leak: (a) Login details, (b) Security Answer, (c) Device and Card Details

from Bank 1 and 2 apps contain login information like username, card number. Similarly, as mentioned in the above point we could not perform this experiment for Bank 3 and Android Pay apps.

Table II presents the summary of results of the security tests performed on all the e-wallet apps. It is clear from the results that Android Pay is quite secure and stable and is not vulnerable to common attacks. While the other banking e-wallet apps could be easily exploited with simple attack vectors. Thus, Bank 3 app is more secure while Bank 1 and Bank 2 apps are equally vulnerable.

Table II
SECURITY ASSESSMENT FINDINGS

Rules	**Bank 1**	**Bank 2**	**Bank 3**	**Android Pay**
No third-party ads	✓	✓	✓	✓
No rooted devices	✓, bypassed	✗	✗	✓
No emulator	✓, bypassed	✗	✗	✓
Self-verification integrity	✓, bypassed	✗	✗	✓
Obfuscated or packed	✓	✓	✓	✓
Not debuggable	✓	✗	✗	✓
No legacy app versions	✗	✓	✓	✓
Https enforced	✗	✗	✓	✓
Proper certificate pinned	✗	✗	✓	✓
No weak cryptography	✗	✗	✓	✓
Proper key storage	✓	✓	✓	✗
No sensitive information in backups	✓	✓	✓	-
No sensitive information in memory	✗	✗	-	-

VI. CONCLUSIONS

This paper presents the security assessment of e-wallet apps of some leading banks in Canadian market. We performed manually analysis on three apps and compared with the Android Pay, which is the most popular and quite secure e-wallet app. Our analysis targets basic device, application and communication security. It is found that e-wallet apps in the Canadian market are not well secured against trivial attack vectors.

REFERENCES

[1] CBA, "Payments security white paper," July 2015. [Online]. Available: http://www.cba.ca/payments-security-white-paper

[2] TrendMicro, "2017 mobile threat landscape," February 2018.

[3] F-secure, "The state of cyber security," January 2017.

[4] E. Filiol and P. Irolla, "(in)security of mobile banking...and of other mobile apps," in *Black Hat Asia*, March 2015, pp. 1–22.

[5] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler, "Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world," in *Proc. 24th USENIX Security Symposium, Washington, D.C.*, August 2015, pp. 17–32.

[6] T. Chothia, F. D. Garcia, C. Heppel, and C. M. Stone, "Why banker bob (still) cant get tls right: A security analysis of tls in leading uk banking apps," in *International Conference on Financial Cryptography and Data Security*. Springer, December 2017, pp. 579–597, doi: 10.1007/978-3-319-70972-7_33.

[7] T. Kim, H. Ha, S. Choi, J. Jung, and B.-G. Chun, "Breaking ad-hoc runtime integrity protection mechanisms in android financial apps," in *Proc. 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, April 2017, pp. 179–192, doi: 10.1145/3052973.3053018.

[8] L. Nguyen-Vu, N.-T. Chau, S. Kang, and S. Jung, "Android rooting: An arms race between evasion and detection," *Security and Communication Networks*, vol. 2017, August 2017.

[9] W. Yang, Y. Zhang, J. Li, H. Lui, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! finding flawed implementations of third-party in-app payment in android apps," in *Proc. Annual Network & Distributed System Security Symposium (NDSS)*, February 2017, doi:10.14722/ndss.2017.23091.

[10] R. Chanajitt, W. Viriyasitavat, and K.-K. R. Choo, "Forensic analysis and security assessment of android m-banking apps," *Australian Journal of Forensic Sciences*, vol. 50, no. 1, pp. 3–19, 2018.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 26,2021 at 22:18:20 UTC from IEEE Xplore. Restrictions apply.