Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps

Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, Dawu Gu
Shanghai Jiao Tong Univeristy
{yangwenbo1990810, yyjess, jarod, ice_wisdom, dabyto, shengdexinqing, dwgu}@sjtu.edu.cn

***Abstract*—The massive growth of transaction via third-party cashier has attracted numerous mobile apps to embed in-app payment functionality. Although this feature makes the payment easy within apps, transactions via current third-party in-app payment involve more sophisticated interactions between multiple participants compared to those using traditional payments. The implementations in mobile apps also lack security considerations. Therefore, such transaction exposes new attack vectors and could be exploited more easily, leading to serious deceptions such as payment forging.**

To investigate current third-party mobile payment ecosystem and find potential security threats, we conduct an in-depth analysis on world's largest mobile payment market–China's mobile payment market. We study four mainstream third-party mobile payment cashiers, and conclude unified security rules that must be regulated by both cashier and merchant. We also illustrate the serious consequences of violating these security rules, which may cause up to four types of attacks against online and offline transactions. Besides, we detect the seven security rule violations to the payment in Android apps. Our detection result shows not only the prevalence of third-party in-app payment, but also the awful status quo of its security. Over 37% Android apps with at least 100,000 users embed third-party payment functionality. Hundreds of them violate security rule(s) and face with various potential security risks, allowing an attacker to consume almost every aspect of commodities or services in life without actually purchasing them or deceiving others to pay for them. Our further investigation reveals that the cashiers not only have improperly designed SDK, which may expand the attack effects, but also release ambiguous documents and even vulnerable sample codes, directly leading to the mistakes committed by merchants. Besides the cashiers' ignorance for security, our successful exploits to several apps show that these flawed implementations can cause financial loss in real world. We have reported these findings to all the related parties and received positive feedbacks.

I. Introduction

The past few years has witnessed the extraordinary developments in mobile payment. The significant growth of smartphone based transaction promotes the usage of third-party mobile payment services in mobile apps. Compared to transaction processes with traditional payment channels (e.g., via credit card), transaction with third-party in-app payment is settled within mobile app conveniently. Users can pay their bills directly without switching to another app or web browser. Moreover, third-party cashiers are willing to provide such functionality for popular apps to fulfill in-app payment. To help an app access their mobile payment service, cashiers provide SDKs, leading a straightforward integration of in-app mobile payment functionality. As a result, more and more apps are using third-party in-app payment as their major payment channel. Nonetheless, implementing secure in-app payment is not easy. In-app payment is still in its incipient stage and is especially error-prone due to both the misunderstanding of app developers, the improper designed services, and the ambiguous documents or code samples released by cashiers. It also involves more participants and interaction steps compared to traditional payment processes. Therefore, the potential attack surface is much wider.

Previous studies [25][27][22][18][21] mainly focus on the security of e-commerce of web application other than mobile apps. Although numerous security flaws of e-commerce web applications have been revealed when integrating services of third-party cashiers. On mobile platform, however, the trust boundaries are redefined. Client apps are considered untrusted since all the data handled by apps can be manipulated by the attacker. Moreover, workflow of e-commerce in web applications is unable to cover the entire transaction of in-app payment, since the introduced mobile client plays an important role in this multi-party model. Thus, it is inadequate to directly employ traditional flaw detection of web applications on mobile apps.

To the best of our knowledge, there exists neither unified specification to regulate in-app payment process nor assessment approach to validate the security of them so far. Documents and samples of transaction provided by most in-app payment cashiers are partial or even incorrect. Analysts must reverse-engineer the binary code of an app and its integrated in-app payment SDKs to ensure the process. Before finding potential security flaws, it is required to first summarize the status quo of current third-party in-app payment. It's also urgent to conclude the basic security rules which should be obeyed by both cashiers and merchants throughout the transaction process as well as their violations detection methodology.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23091

To investigate how widespread is insecure in-app payment in apps, in this paper we make a study of the world's largest smartphone and mobile payment market–China's mobile market. A distinguishing feature in China's mobile market is that most apps support third-party in-app payments **only**. User cannot use other payment channels such as credit card or online bank to pay the bills in app. On the other hand, users in China also prefer to choose these third-party cashiers to manage their finances because these cashiers provide more services than traditional banks. Users can not only do shopping or make investment conveniently, but also transfer to each other for free through their accounts in cashiers. Once these third-party payments are vulnerable, every in-app transaction is inevitably suffering severe security threats. Our study tries to answer the following questions for our research target: **a)** What exactly should be done to implement a secure in-app payment? **b)** Which kind of attack can be conducted and who suffers financial loss? **c)** What is the ratio of app with insecure in-app payment and how to detect them? **d)** What factors affect the insecure implementation?

To answer the above questions, we first conduct an in-depth analysis on services of four mainstream third-party cashiers. The results of our analysis include: **1)** we unveil the details of their payment processes through reverse-engineering relevant SDKs and apps; **2)** we conclude a series of security rules which regulate the security requirement of in-app payment; **3)** we illustrate the severe consequence of violating these rules including four attacks against both online and offline transactions. Besides, we develop the methodology to detect the seven violations of our proposed security rules in Android apps and their servers as well as the third-party payment SDKs. We first prove the prevalence of in-app payment through scanning 7,145 Android apps with at least 100,000 users for each and pinpointing at least one in-app payment SDK in above one-third of the analyzed apps. We then detect the flawed implementations of the 2,679 apps and find hundreds of them are vulnerable. Also none of the four cashier's SDKs reach the requirement regulated by our proposed security rules. Though the improper designed SDKs do not directly lead to vulnerabilities, they would expand the effects of user deception attacks. Through combining these flaws, we can perform various attacks targeting both merchants and users, including shopping for free or with others' money. The affected area covers almost all aspects of daily life.

Finally, we analyze the root cause of the flawed implementations of in-app payment. We find the documents and sample codes provided by cashiers are often not examined and confuse developers. Some of them are incorrect and even vulnerable, which directly leads to these flaws of merchants. Different from previous work [25][27][22][18] which focus on the security analysis of merchants, our findings reveal that the cashiers also contribute to the flawed third-party payment on mobile platform. We have reported all the problems to the affected cashiers and obtained their credits.

II. In-app Payment Demystified

Although in-app payment is pervasive in Android apps, the process of how an app fulfils a transaction via third-party payment service is often obscure due to several reasons. First, implementation variation of in-app payment is significant. Unlike well-developed multiple-party communication processes such as OAuth[6], in-app payment has no unified specification to follow. Different third-party payment service providers (cashiers) regulate different in-app payment processes and release their own SDKs for app to integrate. Implementation aspects of third-party payment services such as used web APIs, integration style of SDKs, and the parameters required differ greatly from each other. Second, cashiers often release documents and samples to app developers. Developer relies on those documents and code samples to integrate in-app payment SDKs and implements the payment process. However, our review illustrates that most of these documents are ambiguous and may confuse the app developers. Some code samples even conflict with the process regulated by the documents. Only by reading cashiers' documents and studying their sample code is not adequate to conclude the exact payment process, and we will give our result in Section V-E. Third, testing the in-app payment not only involves actual payment with money expending, but also requires some franchises and relevant documents only granted to verified identity such as registered companies. Many analysis efforts lack such qualifications are therefore impeded.

To demystify the details of in-app payment process, we first give a brief description of participants involved in payment process. Then, we choose four popular cashiers to analyze their documents and code samples. And we reverse-engineer popular apps with in-app payment to understand the details of payment implementation. After this reverse engineering work, we gain a panoramic view of in-app payment process: two representative payment process models that cover necessary transaction steps for four cashiers are concluded.

A. Definitions

In a typical in-app payment process, user browses, selects and buys commodities in a merchant app (***MA***). Implemented by the merchant, the ***MA*** and the merchant server (***MS***) interact with each other. Information such as users information, commodities provided, and order information are stored in databases on the ***MS***.

To support payment in app, an ***MA*** integrates one or more third-party payment SDK (TP-SDK) released by the third-party cashier. In a checkout process, user chooses a third-party cashier in the ***MA*** and makes a payment to the cashier. The cashier server (***CS***) records the payment information and status, and informs the merchant the completion of the payment. The complete payment information is then stored to the ***CS***.

B. Unveiling Payment Process

To unveil the payment process, we first choose four popular cashiers as our research targets: WexPay (in-app payment service provided by Wechat Wallet) [9], AliPay (Alipay Wallet) [1], UniPay (Unionpay Wallet) [8] and BadPay (Baidu Wallet) [4]. Each of them has at least 100 million users. Merchant can register to all four cashiers as long as it owns a legitimated company registered to the Chinese Commerce and Industry Bureau. For every cashier, we get the TP-SDK and auxiliary materials including code samples and relevant documents. The documents describe not only interfaces of TP-SDK but also the suggested payment process and Web APIs of cashier server. Code samples illustrate simplified implementation for client app and server. We also download 7,145 apps with at least 100,000 users for each from *Myapp* [5], the largest Android APP market in China. Through studying the documents of four cashiers, reverse-engineering the TP-SDK and downloaded APPs with static and dynamic analysis, monitoring the network traffic of the transaction process, and implementing sample code to real app and server, we have two observations about the in-app payment: 1) how prevalent is the in-app payment in Android APP; 2) which payment process model does merchant need to comply with when integrating third-party in-app payment function. We detail these observations in the following sections.

C. TP-SDK Identification

In order to find out which app uses third-party in-app payment, we adopt feature based identification strategy to detect apps with TP-SDK. We reverse-engineer TP-SDKs of four cashiers and extract their unique features. We observe that if an ***MA*** uses a TP-SDK, it needs to invoke a specific interface and passes parameters, hence we make use of these interfaces as the feature of TP-SDKs. For instance, if an ***MA*** uses TP-SDK of AliPay, it must pass the payment order information to AliPay SDK through a certain interface [1]. For other three TP-SDKs, there are also similar features. Notice that the name of interface is not always an available feature. Developers may use code obfuscation tools such as *ProGuard* [7] to obfuscate the function name in app. Therefore, we manually pick a combination of special strings in every TP-SDK that are seldom used elsewhere as extra features. Utilizing those features, we build a static analysis tool based on *AndroGuard* [3] to scan all 7,145 apps. The result is listed in Section V.

D. Process Analysis

After studying the documents and sample code of four cashiers, along with static and dynamic analysis to merchant apps and servers, we find that the payment processes adopted by four cashiers are somewhat different. However, they can be concluded as two in-app payment process models (in Figure 1 and Figure 2). Among the four cashiers, WexPay and UniPay follow the process model I (in Figure 1), while AliPay and BadPay follow the other. We first choose process model I as an example to illustrate a complete third-party in-app payment process in detail. It is a simplified model only including essential steps and parameters of a transaction. The whole process of the model contains nine steps in general.

1) The ***MS*** receives a merchant order ($order_m$) and the type of cashier after a user selects the commodities and chooses a third-party cashier in the ***MA***. $order_m$ contains order information only related to merchant (e.g., the type and the amount of commodities user wants to buy), since cashier has not involved yet by now.
2) The ***MS*** generates a payment order ($order_p$) according to $order_m$, and sends it to the cashier by invoking the Web API defined. The $order_p$ should contain information about this payment, generally including four essential parameters: *order ID*, *merchant ID*, *total amount*, *notify URL address*. *Order ID* refers to the identifier of the payment order. It is usually generated by the merchant and should be unique. *Merchant ID* is used by the cashier for uniquely identifying the merchant. *Total amount* refers to the total amount of money involved in the payment that cashier receives from user on behalf of the merchant. *Notify URL* is an URL address of the ***MS***. After a payment finishes, cashier needs to inform the merchant of the result by sending the notification to the *notify URL*.
3) After receiving and verifying the $order_p$, the ***CS*** will store payment information into its database, and returns a signed message (*TN*) that contains a *transaction number*. Notice that the *transaction number* identifies the payment order and is generated by the cashier. It does not contain any unnecessary information of the order, such as the *notify URL* or *Total amount*.
4) After the ***MS*** receives and verifies *TN*, ***MS*** should sign *TN* and send it to ***MA***. Notice that *TN* now contains the merchant's signature.
5) ***MA*** deals with received *TN*, and passes it as parameters to the interface defined in TP-SDK.
6) TP-SDK prompts its payment UI (an *Activity* in Android) to accept user's confirmation. The payment Activity in TP-SDK shows the detailed information of the payment order acquired from the ***CS*** through its own network channel (omitted in the Figure). After user confirms the payment order and enters the account password, TP-SDK sends the pay request to the ***CS***. The ***CS*** checks the request, and then pays for the order with money in user's account.
7) The ***CS*** sends a notification of payment to both TP-SDK (the step with *apostrophe*) and the ***MS*** (the step without *apostrophe*).
8) The ***MA*** shows payment result to user according to the notification received by TP-SDK.
9) The ***MS*** validates the signature of the notification, and makes an extra query of the notified payment order to the ***CS*** to confirm details of the order including *order ID*, *merchant ID*, *total amount*, *etc*.

After all the above steps, the transaction is settled and the merchant can ship commodities or provide services to user.

When adopting process model I, WexPay and UniPay implement similar process with nuance differences. Both cashiers require different extra parameters for $order_p$ and $order_m$. Also, UniPay does not require the *TN* message to be signed and does not include Step 8.(in Figure 1) as a necessary step in its suggested process.

When adopting process model II, however, AliPay and BadPay have relatively larger differences to WexPay and UniPay. The main difference occurs in Step 2. The ***MS*** just sends the generated signed payment order ($order_p$) back to the ***MA*** other than to the ***CS*** after receiving the merchant order ($order_m$) request from the ***MA***. Compared with model I, in which the ***MA*** can only receive *TN*, the ***MA*** in Figure 2 receives the complete payment order information including order ID, total amount of the payment, the notify URL address of the ***MS***, etc. And it transfers all the information to the

[1] *com.alipay.sdk.app.PayTask->pay()*

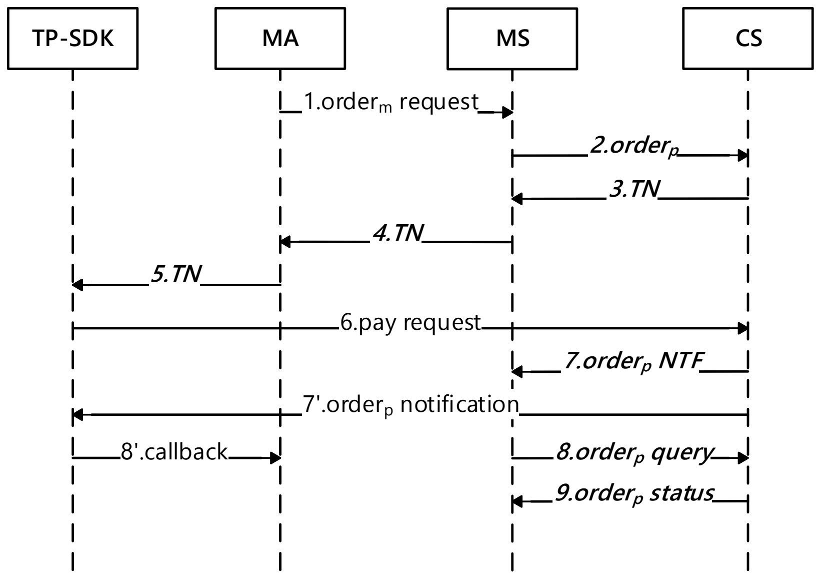

Fig. 1: In-app Payment Process Model I adopted by WexPay and UniPay

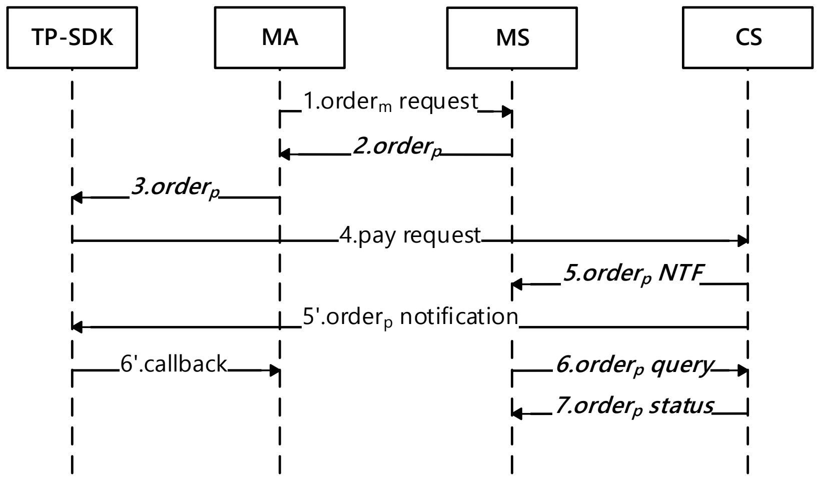

Fig. 2: In-app Payment Process Model II adopted by AliPay and BadPay

integrated TP-SDK, which is responsible for dealing with all the detailed parameters of the payment order in this process.

In Figure 1 and Figure 2, messages with bold and italics text need to be **signed** by the sender to prevent being tampered. So another important factor in the transaction process is the signing method of messages adopted by cashiers. AliPay and UniPay regulate the *SHA1-RSA* as their signing method. Merchant generates its RSA private key and public key, and sends the public key to the cashier. Also, cashier informs every merchant its public key. The ***MS*** verifies the received signed message with the cashier's public key, and sends message signed with its private key to cashier or to the ***MA***. However, WexPay and BadPay adopt hash function (e.g., MD5) with a secret key (as the ***salt*** of the hash function) to generate the signature. The *secret key* is shared between the merchant and the cashier. In the later part of this paper, we denote both the secret key of hash function and the merchant's RSA private key as *KEY*.

III. Security Analysis

We conduct further security analysis to the process models we concluded above in this section. The security of third-party payment has been studied before in previous work [25][18][27][21]. However, all of them focus on Web service. In the prevailing mobile platform, the *in-app* payment introduces new multi-party models and thus, faces new security challenge. The merchant client application and the embedded TP-SDK play more significant roles which do not exist in traditional Web model. So it's necessary to re-consider the security threats of the in-app payment on mobile platform.

Although the payment process models that regulated by cashiers have been vetted before releasing and are supposed to be secure, such multi-party models still struggle against various unexpected security threats due to the information asymmetry in the transaction process. Moreover, the whole transaction process involves multiple parties, not only cashiers, but also merchants and users. Due to the ambiguous documents and confusing sample code released by cashiers, developers of merchants often disobey the process model regulated by cashiers and implement diversified payment processes, which may lead to potential security flaws. Any mistake committed by any party in the multi-party model may lead to the whole process vulnerable. Therefore, it's necessary to conclude security rules to regulate all parties in the model.

In this section, we first define the adversary model. Then we give the security rules which must be complied, and what the cashier and the merchant should pay extra attention to throughout the entire transaction process of in-app payment. Finally, we describe four attacks in detail under our reasonable adversary model if the cashier or merchant violates the security rules, which may lead to the loss of multiple participants in the model.

A. Adversary Model

Before proposing security rules and the attacks caused by rule violation, we first define the adversary model as followed. We assume that the attacker can reverse-engineer ***MA*** and the embedded TP-SDK, since it can be easily acquired in Android Application market (even if the app is protected, techniques have already developed to circumvent it [28]). When the attack targets cashier or merchant, the attacker plays the role of a malicious user and is assumed to be able to manipulate execution or data of local app and system, and to tamper or forge the network communication. But when the attack involves other users of the ***MA***, attacker is assumed only to control the data transmission between them (e.g., perform MITM attack with the ARP spoofing or deceive users to attacker's malicious WiFi), but not to control other users' devices, i.e., not able to install malware or repackaged ***MA*** on them by subterfuge. Though the attacker is not able to sniff or tamper the network traffic between ***MS*** and ***CS*** under any circumstances, it can forge request or message to either ***MS*** or ***CS*** in our model.

B. Security Rules

According to the two types of process model adopted by four cashiers and the adversary model, we conclude the following security rules that must be obeyed throughout the whole process involving both cashiers and merchants, no matter how cashier regulates the process model or which cashiers ***MA*** chooses to use. Otherwise, the process will be breached.

1) Payment orders must be generated (Figure 1) or signed (Figure 2) by the ***MS*** only.
2) *Never* place any secret (e.g., private key for signing) in the ***MA***.
3) TP-SDK must inform user *detailed* information of the payment order.
4) TP-SDK must verify the transaction belonging to the ***MA***.
5) *Always* use secure network communication between client and server.
6) *Always* verify the signature of received messages.
7) ***MS*** should make an *extra* query to confirm notified payment's details.

There are four types of attack that the payment process may suffer if one or more violation of security rules occur, and the victims involve normal users of ***MA*** and the merchant. Then we will describe them in details.

C. Order Tampering

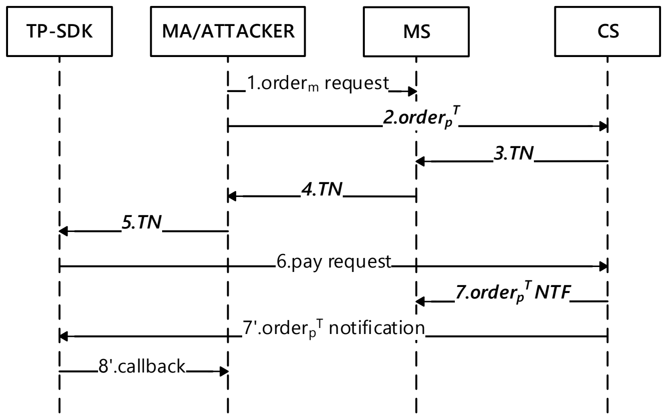

Fig. 3: Order Tampering Attack to Process Model I

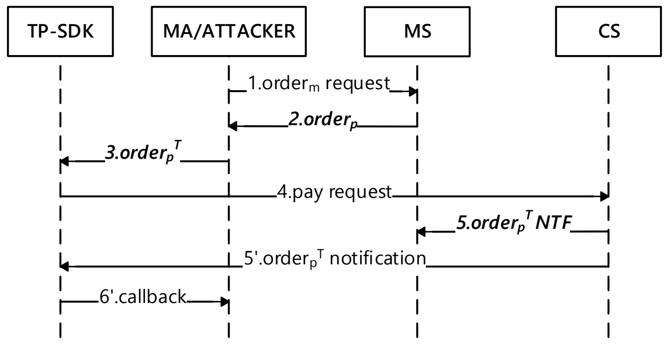

Fig. 4: Order Tampering Attack to Process Model II

In this type of attack, the attacker acts as a malicious user. If the merchant fails to obey the **Security Rule 1** and **Rule 7**, then attacker can cheat the merchant by sending a payment order ($order_p$) to the cashier different from the actual merchant order ($order_m$). In this situation, the attacker could tamper the content in the payment order such as the total amount and thus pay less money for the ordered commodities without the merchant's awareness.

The attack for process model I is shown in Figure 3. In model I (Figure 1), the signed payment order is generated and sent by the ***MS*** to the ***CS***. A local attacker can only obtain the (*TN*) message which does not include any detailed information (e.g., the total amount of the payment) of the payment order. Thus, it's impossible to tamper the payment order information. However, if the ***MA*** incorrectly implement the payment order generation step in the app rather than its server, the attacker can succeed in tampering the payment information. Since the attacker can take full control of local app and system, we merge the attacker with the ***MA*** in the Figure 3. The $order_p^T$ in the figure indicates that the payment order has been tampered already and so does Figure 4.

In model II (Figure 2), though the complete payment order information can be achieved by the attacker ($order_p$ need to be returned to ***MA***), he can not tamper it since the payment order is signed by ***MS***. A payment order with wrong signature will be rejected by the cashier. However, if the merchant signs the payment order or leaks the *KEY* in the ***MA***, then the attacker with full control of the local app and system could easily intercept all information appeared in the ***MA***, and is then capable of tampering the received order information (e.g., modifying the price), and re-signing it as a legit one to TP-SDK (as shown in Figure 4).

To fulfil this attack, another implementation flaw is required: the merchant fails to confirm the notified payment order information to ***CS*** (**Security Rule 7**). Otherwise, ***MS*** can get every details of the notified payment order including total amount, merchant ID, etc after make the query to ***CS***. It can refuse the tampered order after verifying every details of it and does not ship commodities.

In all, the violation of **Security Rule 1** and **Rule 7** will make the merchant become victim, and attackers as malicious users can buy commodities in any price.

D. Notification Forging

If the merchant fails to obey the **Security Rule 2** (or **Rule 6**), and **Security Rule 7**, then it suffers notification forging attack, allowing attackers to purchase commodity without paying it. In the attack, a normal payment process is performed until the TP-SDK requires user to confirm the order and enter password to pay for it. At that time, an attacker does not pay for it, but instead sends a fake payment result notification to notify the ***MS*** that the order is paid successfully. The attack model to process model I (Figure 1), for example, is shown in Figure 5. If the order is not paid (Step 6. in Figure 1), it still remains *'pending'* status, and the ***MS*** will not receive the notification from the ***CS*** (Step 7. in Figure 1). However, afterwards attackers can forge the notification and send it to the ***MS*** (Step 6. in Figure 5). If the merchant trusts this fake notification and does not confirm the order's details to the ***CS*** (Step 8. in Figure 1), the payment is successfully forged. The attack can also be performed to model II (Figure 2) with the same way, which we omit it here.

Attackers need to exploit several mistakes committed by the merchant to make a forged notification available. First, the notify URL address of ***MS*** that receives the payment notification from ***CS*** should be known beforehand. So the attack requires ***MA*** to contain the notify URL address, which would be placed by the developers accidently. Actually, as we illustrated before, ***MA*** who adopts process model II (AliPay and BadPay) certainly contains the notify URL, since all order information including notify URL is used as input of the TP-SDK (Step 3. in Figure 2). Second, the attack needs to construct a forged payment order notification of the cashier and cheats ***MS*** to accept it. Attackers can obtain the data format of the notification message from documents released by the cashier, and then forge it with a signature which labels the identity of the sender. The *KEY* used here is often extracted from the ***MA***, in which the merchant's developers place this shared secret key by mistake (**Security Rule 2**). Notice that among four cashiers, only the notification of those who adopt hash-function as their signing method (WexPay and BadPay) can be forged because the cashier and the merchant share the same *KEY* as their signing *KEY*. For those using *SHA1-RSA* to sign the messages, the RSA private key of cashiers can be hardly leaked, thus, forging the cashier's message with legal signature is quite impossible. Moreover, we observe that some ***MS***s even ignore the validation of the signature of the received messages (**Security Rule 6**). Thus, the fake notification even with wrong signature is unconditionally accepted. Finally, similar to *Order Tampering* attack, notification forging also needs the ***MS*** to ignore the order re-confirmation step. Otherwise the merchant can find out that the notified payment order is still remain 'pending' status in ***CS***.

In all, if the merchant 1) fails to check the signature of notification message at the ***MS*** or leaks the signing *KEY*, and 2) misses notified payment confirmation at the ***MS***, then the attack will succeed.

E. Order Substituting

Different from the two attacks above, the victim of order substituting attack becomes the normal user of ***MA*** rather than the merchant. The cause of this type of attack involves multi-parties' violation of security rules including both cashier (**Security Rule 3** and **4**) and merchant (**Security Rule 5**). In this attack, the attacker substitutes an order of one transaction to another, and misleads a victim user to pay for the attacker's order unconsciously.

Figure 6 shows the order substituting attack to process model II (Figure 2). The attack is available when the message returned from ***MS*** is transferred with an insecure network communication channel. Thus, the attacker can act as a man-in-the-middle between ***MA*** and ***MS***. Attackers can intercept the message and substitute signed payment order ($order_p$) with another one ($order_p^A$) of a legal transaction, and send it to the ***MA*** on victim's device. The victim will then pay for the attacker's order rather than his own order. Notice that the attacker uses a legal payment order to replace the original one. This message usually belongs to a normal trade performed by the attacker beforehand (steps between Step 2. and Step 2'. in Figure 6), so it is reasonable to cheat the victim's TP-SDK and finish the transaction with this message successfully. The attack to process model I (Figure 1) is similar. The only difference is that the attacker needs to substitute the *TN* message (Step 4. in Figure 1) rather than the $order_p$ (Step 2. in Figure 2) returned from ***MS***.

The root cause of this attack includes the lack of secure communication channel as well as the inadequate prompt information showed by TP-SDK. We discover that the payment Activity (Activity is a component of Android application, acting as an user interface) of TP-SDK generally does not show enough information about current payment order, thus the victim will confirm and pay for another order without being aware of it. For example, if the payment Activity only shows the total amount of the order, then the attacker could make an order with the same price of the victim's order. Even if some TP-SDKs show the commodities and the merchant

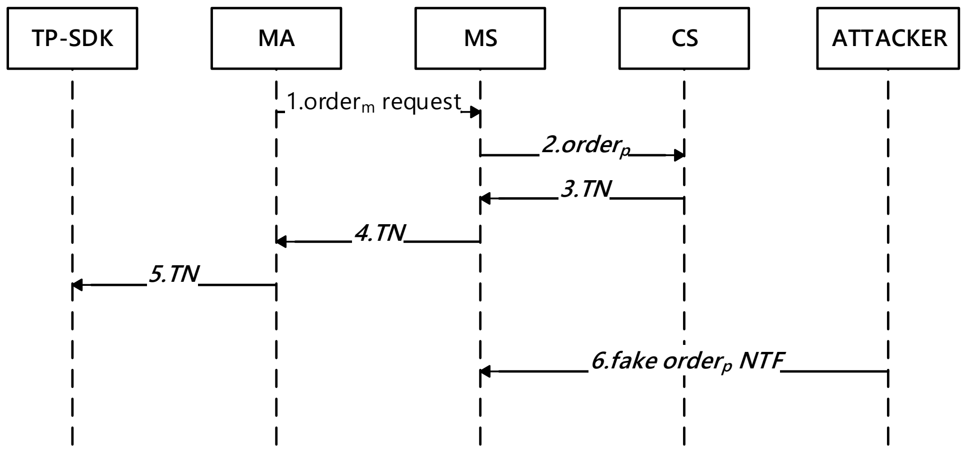

Fig. 5: Notification Forging Attack to Process Model I

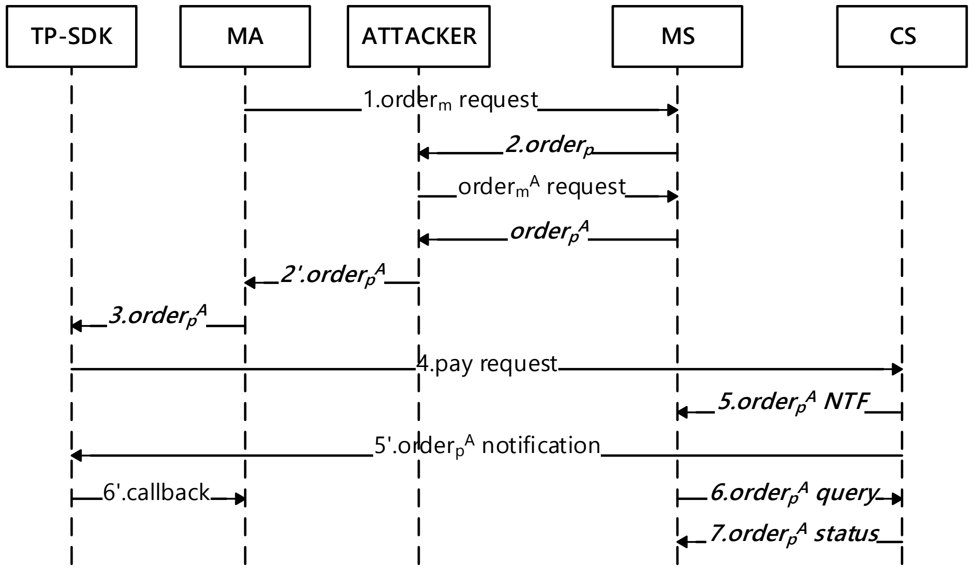

Fig. 6: Order Substituting Attack to Process Model II

name of the order, the attacker could make an order with same commodities while modifying the consignee since it is not difficult for attacker to know what victim is going to buy through eavesdropping the merchant order request (Step 1. in Figure 6) via insecure network connection between the ***MA*** and the ***MS***. What's worse, if the TP-SDK accepts the $order_p$ (or *TN*), whatever it is generated by the host ***MA*** or not, this attack can be expanded that even a transaction from another ***MA*** can be substituted to that from one ***MA***. In other words, if attackers substitute the original $order_p$ (or *TN*) with anther $order_p$ (or *TN*) of malicious merchant registered to cashier by attackers themselves, the money paid for the transaction will be transferred to attackers directly. Nevertheless, our investigation indicates that some TP-SDKs do not verify *TN* carefully, allowing attackers to substitute the original one easily.

In a word, as long as 1) ***MA*** adopts insecure network channel to communicate with ***MS*** (**Security Rule 5**), 2) and the TP-SDK in ***MA*** does not display clear information about the payment order on its payment Activity (**Security Rule 3**), attackers can perform MITM attack and deceive users to pay for orders not belonging to themselves. Also, missing $order_p$ (or *TN*) verification in TP-SDK (**Security Rule 4**) will expand the impact of the attack.

F. Unauthorized Querying

If the merchant violates the **Security Rule 2**, leaking its *KEY* to attackers, it will also suffer the unauthorized querying attack. An unauthorized querying attack allows attacker to query every transaction recorded in ***CS***, acquiring secret business information which should only be shared by cashier and merchant. The root cause of this attack is due to the leaking of merchant's authentication credential. Cashiers provide several Web APIs for merchant to query various information, such as every payment order's status and details, the merchant's history bill of everyday, etc. Furthermore, cashiers make use of the signing *KEY* to authenticate the identity of each merchant.

However, the *KEY* may be accidently placed in the ***MA*** by the developers of the merchant. So the attacker could utilize the leaked *KEY* to query transaction information illegally.

IV. Detecting Flawed In-app Payments

The violation of the seven security rules causes exploitable attacks and leads to serious consequences. In this section, we will describe how to convert these rules into detectable forms in the payment process. Detecting these violations is helpful to find flawed in-app payments to actual loss. Furthermore, we discuss the feasibilities and details of detecting such flaws.

A. Local Ordering

According to the **Security Rule 1**, ***MA*** is prohibited to generate payment orders for those adopting process model I. Local ordering refers to the incorrect ordering behavior implemented by the ***MA*** rather than the ***MS***. It allows the attackers to tamper the payment order. Notice that this flaw only appears to apps with WexPay or UniPay, since in their regulations, placing the payment order must be enforced by the ***MS***.

To detect this violation of **Rule 1**, we search the existence of a relevant *destination URL* used by the merchant to place a payment order. In detail, app will visit destination URL[2] for WexPay and UniPay, respectively. The visit indicates the incorrect behavior of generating order locally. Therefore, we first scan all strings in DEX file and resource file of an APK to find whether the above two strings exist. If so, the app is then manually tested to confirm the security flaw.

B. KEY Leakage

In the two payment process models, several messages transmitted need to be signed. According to our proposed **Security Rule 2**, sensitive information, especially the *KEY* is prohibited to appear in app. Otherwise, attackers can tamper or forge messages with legal signature and camouflage to be certain party and cheat others in the multi-party model.

We combine pattern matching and dynamic testing techniques to detect *KEY* leakage in apps. We develop an automatic detecting tool based on *AndroGuard* to search leaked *KEY* in app adaptively against specific cashier.

For WexPay, it adopts hash function with secret key to generate the message signature. The secret key for message signing is a 32-byte string with arbitrary content shared with merchant and cashier. The ***MA*** uses this key to sign message so we would like to search such hard-coded key in app. However, simply searching 32 bytes length string in an ***MA*** often gives a huge amount of candidates. To effectively determine the potential key, we utilize a Web API provided by WexPay as an oracle to substantiate the key identity accurately. The Web API offered by WexPay allows merchant to download the history bill of one day with three necessary parameters: *appid*, *mch_id*, and *secret key*. Therefore, we could leverage the *appid* and the *mch_id* to help identify the *secret key*. Notice that the features of these two parameters are apparent: the *appid* is a 18-byte string with a *wx* prefix, and the *mch_id* is a 10-byte string comprised of digits only, and both two parameters are uniquely allocated to merchant. We can first locate strings with similar features in DEX file and resource file (strings.xml), and query the Web API for the identity of the found parameters. If any of the input parameter is incorrect, the response of the query gives a corresponding notification. For instance, if the first *appid* parameter is incorrect, the Web API would directly return a "*wrong appid*" notification without considering the following parameters. Thus we could check each parameter individually until its correctness is identified, which significantly improve the efficiency. And if all three parameters are correct (which means we find a leaked key in app), the Web API responds either the merchant's real bill data, or "*no bill exists*" if no transaction happened on that day. Using this testing approach, we can effectively find leaked WexPay key in an app. Similar to WexPay, BadPay uses a shared secret key to sign its messages. However, no Web API is provided by BadPay for us to verify the potential key candidates. Considering that far fewer ***MA***s use BadPay, we could confirm the key through manual reverse engineering.

For AliPay, merchant uses an RSA private key within a Based64-encoded standard ASN1 certificate to sign the order information. The certificate format contains remarkable feature (A string with *'MI'* as prefix and at least 300 bytes long) and can be easily located. However, the app may also contain such certificates to fulfil other functionalities. To confirm the application of found certificates, we adopt the following two heuristics. We first check whether the variable name of the candidate certificate contains *ali* or *alipay*. Second, we make use of the cross reference searching to find the Java class that refers to the candidate certificate. Since the private key in a certificate is used to sign the order of AliPay, the order information is often generated in the same class that uses the private key. This generated order information contains specific feature strings ("**&service=mobile.securitypay.pay**" for example) and can be easily identified. If a certificate corresponds to one of the above properties, we regard it as the signing key of AliPay. Similar to AliPay, UniPay also uses RSA private key to sign its messages but the private key is encapsulated in a CER format. We also adopt similar detection methodology to UniPay.

C. Incomplete Prompt

When an ***MA*** invokes the TP-SDK and shows the payment Activity to users (e.g., between Step 5. and Step 6. in Figure 1), users need to confirm the order and decide whether to pay for it. As the **Security Rule 3** implies, detailed order information should be prompted to user in the payment Activity completely. Otherwise, user may suffer deception, resulting in an attack that what user pays is not what he/she really buys (*Order Substituting Attack*).

We detect this security flaw by checking whether TP-SDK displays enough information about the payment order to user during the payment. In detail, the following fields are checked: 1) payment order ID that represents the order uniquely in both merchant and cashier. 2) what commodity or service that users are going to pay. 3) user that the order belong to in merchant app. 4) merchant that the order belongs to. 5) total money of the payment.

[2] https://api.mch.weixin.qq.com/pay/unifiedorder for WexPay; https://gateway.95516.com/gateway/api/appTransReq.do for UniPay

D. Transaction Verification Missing

In a secure payment process, TP-SDK integrated in ***MA*** need ensure that the received payment order (Step 3. in Figure 2) or *TN* (Step 5. in Figure 1) actually belongs to the ***MA*** according to **Security Rule 4**. Otherwise, malicious merchant can expand the *Order Substituting* attack and directly get money from users as we mentioned in Section III-E.

We detect this security flaw through testing whether the TP-SDK accepts a payment order that does not belong to the ***MA***. First, we place a order using a normal ***MA***. Then we intercept the $order_p$ / *TN* message from the ***MS*** and substitute it with $order_p$ / *TN* message generated from another ***MA***. And we check whether the order belonging to another ***MA*** can be accepted successfully by the TP-SDK. If so, the violation of **Rule 4** is confirmed.

E. Insecure Communication

According to **Security Rule 5**, network communication between ***MS***, ***CS***, and ***MA*** (including its integrated TP-SDK) should adopt secure transmission (e.g., via TLS channel). Otherwise, attackers can intercept, eavesdrop or tamper what users want to buy (e.g., in Step 1. of Figure 1), the payment order information (e.g., in Step 2. of Figure 2), or the transaction information (e.g., in Step 4. of Figure 1). It can also directly cause the *Order Substituting* attack, as we mentioned in Section III-E.

According to the adversary model, we mainly concern how to detect the insecure network communication employed between the ***MA*** (including TP-SDK) and the remote server. We set a proxy to conduct MITM attack against HTTP and HTTPS connection to detect the potential flaw. The insecure communication between TP-SDK and ***CS*** may cause wide and serious consequence. Since TP-SDK is integrated by a large number of APPs, all the ***MA***s with this kind of TP-SDK will suffer vulnerabilities (such as payment information leakage, transaction interception and tampering, etc), if the network communication is insecure. So we adopt a refined policy to detect the flaw in TP-SDK. We try to sniff and attack the network communication during a manually conducted payment process. If the connection between TP-SDK and ***CS*** is an HTTP connection, we regard it as insecure. Furthermore, if the connection is HTTPS, we will check whether it verifies the SSL/TLS certificate properly, or implements the certificate pinning. If the TP-SDK uses private protocol communicating with its server, we further audit the security of this protocol (since there are only four TP-SDKs, we could audit it manually).

For the communication between ***MA*** and ***MS***, we only consider the situations of HTTP, insecure HTTPS (without certificate validation), and secure HTTPS. Our purpose is to find out the network connection of the exact step when ***MS*** returns $order_p$ / *TN* to ***MA*** is secure. Since our result needs high accuracy, we manually trigger the ***MA*** to the step and monitor the network traffic.

The limitations of automated analysis methodology to detect the security flaw is so serious that lead to high inaccuracy. The key difficulty is that how to find the URL connection related to the step that ***MA*** and ***MS*** transmit the transaction information accurately. Among a variety of URL strings in apps, it's quite impossible to decide which URL is responsible for transmitting the order/transaction information only by name. In addition, it is common for apps to join several substrings to the ultimate URL address, or even use code obfuscation to sensitive URL, which also raise the difficulty of automated detection. Previous work like MalloDroid [15] only gave coarse detection result without identification of target URL's logic function, and [19] even indicated the inaccuracy of such automated analysis, which further prove the difficulty of this work. Furthermore, finding the target URL with dynamic analysis involves deep human interaction, including registering and login account, clicking products, choosing in-app payment and paying for the order, which is also impossible to be automated and large scale. As a result, we can only do it manually to achieve accurate detection result.

F. Notified Payment Confirmation Missing

As **Security Rule 6** implies, ***MS*** needs to make an extra payment order query (e.g., Step 8. of Figure 1) to confirm every details of the payment order, even if it receives the payment notification. Since this part of implementation is on ***MS***, we can only apply an indirect detection approach to detect the violation. We try to tamper a payment order information which is different from the original payment order but with legal signature, and pay for it. If the ***MS*** accepts the payment order and ships the commodities, then we can conclude that the ***MS*** does not re-confirm the notified payment order. Notice that the tampered order message should be with correct signature, which means the samples here need to be based on the result of *KEY Leakage*. We perform the dynamic detection manually, because the process involves much human interaction such as placing orders and checking the payment's status.

G. Signature Validation Missing

Security Rule 7 implies that ***MS*** is supposed to check the integrity of every received message (e.g., in Step 3., Step 7., and Step 9. in Figure 1). Otherwise, ***MS*** would accept messages even with incorrect signature. To detect this flaw, we try to place an order but without actually paying for it, and then forge an order notification to the ***MS*** with incorrect signature. If the merchant accepts the payment order, Then we can conclude that ***MS*** fails to check the signature properly. Here the samples are based on the apps that commit *Notified Payment Confirmation Missing*. Because we need to exclude the negative result caused by successfully confirming the notified payment.

V. Empirical Study

We make an empirical study of the world's largest smartphone and mobile payment market–China's mobile market. We choose it as a representative example for the following reasons: First, nearly 400 million users in this market use mobile devices to purchase goods and services (by the end of 2015, according to research from the China Internet Network Information Center). In the first quarter of 2016, China's third-party mobile payment tools handled transactions worth more than 5.9 trillion yuan (885.8 billion us dollar). Second, unlike the mobile payment market in the U.S., where most mobile transaction is settled via credit card through web or Apple Pay,

TABLE I: TP-SDK Distribution

Cashier	Number
WexPay	2260
AliPay	1299
UniPay	574
BadPay	34
Total	2679
Sample	7145

Cashier	KEY leakage	Local Ordering
WexPay	155	104
AliPay	398	/
UniPay	0	0
BadPay	7	/

TABLE II: Flaws in Merchant Apps

in China's market most popular apps mainly use in-app third-party payment services. Third, instead of a single payment standard, a variety of payment schemes are provided by different third-party cashiers. Each payment scheme is unique and one app may support many schemes simultaneously. To investigate the flawed in-app payment implementations, we first conduct our TP-SDK identification to the 7,145 most popular apps from *Myapp* market. As Table I shows, 2,679 apps integrate at least one TP-SDK, and most of them contain more than one TP-SDK. The proportion of apps supporting in-app payment is as high as 37.5%, which proves the prevalence of third-party in-app payment.

Then we detect each security flaws we mentioned in Section IV. We classify these flaws into four categories involving ***MA***, TP-SDK, ***MS*** and network communication. Besides, we further investigate the official documents and analyze sample codes released by four cashiers in-depth and gain some interesting and unexpected findings, which may imply the root cause of these flaws. Then we choose representative vulnerable apps based on the result of detection and then exploit their security flaws to prove the validity of our analysis. We provide them as case studies to illustrate the complexity of conducting concrete attacks against real world transactions. After our detection, we find that hundreds of the merchants violate at least one security rule and none of the four TP-SDKs strictly obey these security rules.

A. Flaws in MA

We first detect those flaws in the 2679 ***MA***s. The detection result is shown in Table II. We can see that hundreds of the merchants leak their *KEY*s in ***MA***s. Nearly one hundred merchants using WexPay generate and send payment order in ***MA***s.

Notice that our *KEY Leakage* result of WexPay has no false positive since it is based on the response messages from WexPay's Web API as we mentioned in Section IV-B. However, there may be false negatives because our large scale application analysis is mainly based on static analysis, which means this is even an underestimated result. If the *KEY* or the URL is encrypted in the ***MA***, or sent by the ***MS*** through network to app, then we can not detect it with the string feature. The result of detecting WexPay destination URL is over 130. However, after our manually confirmation, 104 of them really do *Local Ordering*, and the rest just hard-code the URL without invoking it. All the *Local Ordering* apps of WexPay leak the *KEY* since they generate and sign the payment order in ***MA***. However, another 51 apps either use the *KEY* to sign the received *TN* message or just hard-code the *KEY* without actually use it, both violating the security rules. Also we detect that nearly 500 apps integrated AliPay contain strings with RSA private key features. Among them, we find out that 398 apps actually leak their AliPay private keys using the locating techniques mentioned in Section IV-B, and the rest are the keys of other SDKs. Since only 34 apps integrated BadPay, we do the *Key Leakage* detection manually. Among seven flawed apps, five of them hard-code its *KEY*. One of them encrypts the *KEY* and stores it in ***MA***. However, it uses symmetric encryption and hard-codes the key, so the *KEY* of its BadPay can be decrypted and retrieved. And one app receives its *KEY* of BadPay from its server after it places its merchant order, which is unnecessary and also leads to *KEY* leakage. In addition, we find out that several apps share the same *KEY* of WexPay and AliPay, which means that they are either using the same checkout account or developed by the same company.

B. Flaws in TP-SDK

Since TP-SDKs are provided by the cashiers and integrated by the ***MA***, flaws in specific TP-SDK directly affect the host ***MA***. We evaluate the four most popular TP-SDKs provided by AliPay, WexPay, UniPay, and BadPay, respectively. The result is shown in Table III. Only WexPay verifies *TN* correctly. *TN* accepted by WexPay SDK includes parameters of *merchant ID*, *transaction number*, etc. WexPay SDK achieves the ***MA*** certificate through system API in Android, and checks the consistency of the APK certificate and *merchant ID*. It also checks whether the *transaction number* belongs to the *merchant ID*. In contrast, we succeed in invoking the other TP-SDKs (AliPay, UniPay, BadPay) integrated in the ***MA*** by the transaction order of another ***MA***. Also, we find that both WexPay and AliPay require the registered merchants to submit their certificates of ***MA***, while UniPay and BadPay do not. Obviously, only WexPay makes use of the certificate to verify *TN*.

For *Incomplete Prompt*, we manually check every elements presented on the payment Activity of every TP-SDK. We find out that all four TP-SDKs do not present the order's owner in the Activity, leading to the risk of phishing. BadPay only shows the total amount of the payment order, which is obviously insufficient. WexPay and AliPay both show the order description submitted by merchants. But they do not require the merchant to submit necessary information about the order such as the order ID, the order owner, etc. UniPay and WexPay show the merchant name of the order while AliPay and BadPay do not. Also for UniPay, order ID and payment time will be shown to users only if a spinner on the Activity is clicked. In all, every TP-SDK lacks necessary information more or less on payment Activity, which may lead users to be deceived.

Cashier	Transaction Verification	Information Prompt					Network Communication
		orderID	commodity	owner	merchant	money	
WexPay	✓	×	✓	×	✓	✓	secure private protocal
AliPay	×	×	✓	×	×	✓	HTTPS pinning
UniPay	×	✓	✓	×	✓	✓	HTTPS pinning
BadPay	×	×	×	×	×	✓	HTTPS validation

TABLE III: flaws in TP-SDKs

We manually check the implementation of network communication of four TP-SDKs. SDKs of AliPay and UniPay use HTTPS to connect to their servers and adopt certificate pinning. WexPay SDK use a proprietary protocol to communicate with its server. After we reverse-engineer it, we find that it implements its key agreement algorithm based on ECC with hard-coded secure parameters in the SDK. The protocol is then audited manually and is proved to be secure enough against MITM attack. The SDK of BadPay validates the SSL certificate properly, thus, is secure. However, compared to the other three TP-SDKs, BadPay does not adopt SSL-pinning, which means it can not be protected against a compromised CA.

C. Flaws in MS

We tampered the payment order of 15 *KEY leaked* apps with correct signature and paid for it to see whether ***MS*** would accept them. Since the action need really exploit the *KEY Leakage* vulnerability, involving a lot human interactions like ***MA*** account registration, placing a merchant order, tampering the payment order, and paying for it, it's unrealistic to be automated and large-scale. So we did it manually and found that 9 of the 15 apps that finally accepted the tampered order, which means that their ***MS***s miss the notified payment confirmation to ***CS***. Among the nine vulnerable apps, there are ***MA*** that use WexPay, AliPay or BadPay.

We further checked whether the ***MS***s of the nine apps verify the signature properly. We got the notification message format according to cashiers' documents, forged the message with incorrect signature and sent it to the *Notify URL* address of the ***MS***. The result is that two of the nine apps' servers still accept the payment. It indicates that even if the *KEY* is not leaked, attackers can still buy products without paying for it.

D. Flaws of Network Communication

We manually test 87 most popular ***MA***s chosen from the 2,679 apps with embedded TP-SDKs to evaluate the security of their connections to the ***MS*** during the payment. The result is that 45 apps use HTTP connection and 42 apps use HTTPS connection. Among 42 apps who use HTTPS connection, four of them fail to validate SSL certificate properly. In addition, although we do not find proprietary protocol used by ***MA*** to communicate with the ***MS***, some apps adopt home-brewed encryption schemes to protect the content in HTTP connection. Since those encryption schemes generally lack a mature session key management, we regard them as insecure without further investigating the security of their encryption. In all, these 49 vulnerable apps increase the risk of suffering *Order Substituting* attack.

Though we did not test all the 2679 apps due to the inaccuracy of automatic analysis, the result of the 87 samples through manual work shows that a large proportion of merchants are still not cautious in implementing secure network communication. It is an astonishing result that even in 2016 there are still so many popular apps use insecure HTTP channel even if all of them are related to financial transaction. Notice that the tested samples are the most popular apps with larger user amount and stricter security audit. We believe that those samples with less user amount may perform worse on building secure communication. Moreover, we find that cashiers only request the merchant to adopt HTTPS communication in the ***MA*** as an optional requirement rather than a mandatory enforcement. So merchants may ignore the request and implement insecure network communication.

E. Root Cause Inquiry

As we mentioned above, cashier is mainly to be blame for the various mistakes committed by merchant. They release ambiguous, confusing and self-contradictory documents and sample codes, which directly result in the security flaws in ***MA***. Contrary to common sense, we find out that even the official sample code violates several security rules and is apparently vulnerable. Merchants who follow these samples codes suffer our proposed attacks.

After reviewing the sample codes as well as manually checking the documents of the four TP-SDKs, we have some interesting findings. Even though all of the four TP-SDK documents claim that the *KEY* needs to be kept in secret, their sample codes implement the process of message signing in their client apps, leading to the *KEY* leakage, except UniPay. It can be used to explain that so many ***MA***s commit vulnerabilities when using WexPay, AliPay or BadPay (while become secure using UniPay). For example, the sample code released by WexPay directly commits *Local Ordering*, which obviously conflicts with its official process. We also find that the figure describing the process of the payment released by AliPay defines that the order should be signed in client app and so does the sample code, but the code comment in the sample says that the signing step should be in ***MS***. We hypothesize that the contradiction between documents and sample code confused merchant developers a lot, leading those developers who follow the sample during their development to commit these mistakes. Only the sample code of UniPay implementation keeps consistent to their documents, making the order generating and signing in the ***MS*** code, so in our detection none of the APPs are flawed when using UniPay. It shows that the cashier is the key factor to the security of merchant implementation. Also UniPay SDK does not need the signature of *TN* as parameters, while others need. So some ***MA***s of the three TP-SDKs just sign the payment and send it to TP-SDK, also leading the *KEY Leakage*. The correct implementation, by contrast, is that the signature of *TN* or payment order should be received directly from ***MS***. In addition, since the *KEY* of WexPay can be modified to any string as long as merchant notifies cashier, we find that some leaked keys, which are supposed to be random strings, are modified to a weak key such as *12345678912345678912345678912345*, or just the name of merchant, which may suffer brute-force dictionary attack, or social engineering.

We also try to discover the incentive of flawed ***MS***s. We find that not all cashiers release the sample code of ***MS***, thus merchant needs to implement it without example by themselves. Even if there is sample code for ***MS***, server implementation varies a lot compared with the client. Merchant may implement their servers by using Java, PHP, .NET, Python or even native language, which are out of the language scope of sample code released by cashier. Even though cashiers suggest merchants to do these validations in documents or some even implement them in sample code if it has, merchant may also ignore it during the code transplant or without existing correct code examples. Besides, the incomplete prompt and transaction verification missing in TP-SDK are mainly caused by business convenience, user experience or UI design. Even if these flaws do not directly lead to attack, they will expand the attack effect along with other flaws in ***MS*** or ***MS***.

Although the third-party in-app payment involves financial transaction and should have high security level, none of the cashiers emphasize the security in particular. Some cashiers just mention it (e.g., suggesting the merchant to implement the network communication in HTTPS in the end of the documents), which is easy to be ignored by merchants. Not to mention the fact that improper designed TP-SDK and incorrect documents/sample codes released by cashiers. Previous work [25][27][22][18] mainly focus on the the security of the merchant. And [11] ascribes Android code insecurity to informal documentation such as Stack Overflow, while official API documentation is secure but hard to use. However, our work shows that when it comes to the third-party in-app payment on Android, even official documents/sample code released by cashiers lead to the code insecurity, which may be helpful to improve the security of the whole ecosystem of third-party payment.

F. Case Studies

We choose several flawed merchant apps to perform real attacks. It shows that these detected violations of security rules can directly lead to serious consequence including financial loss in real world.

1) Order Tampering: We conducted ethical hacking against two apps to show how attacker tampers the order and purchases any commodities with an arbitrary price. The first case is an app used by several restaurants in China to display electronic menu to customers and allows them to pay for what they order in app via WexPay or AliPay. The second case is an app for customer to order laundry pickup and delivery service.

For the first app, we went to a local restaurant and downloaded the app to order food and drinks. We installed the app on our penetration test Android phone with Xposed [10], a popular function hooking framework. After we finished ordering and entered the table number in the app, we chose to pay the order via AliPay. The app violated the **Security Rule 1**, signing the order information in client app. So before the invoking of the AliPay SDK in the ***MA***, we hooked the Java function used to execute SHA1-RSA signing of the order information in the ***MA*** with the help of Xposed. We have implemented an Xposed module to tamper the price parameter to only one *yuan* (CNY, Chinese Yuan, about 0.16 US Dollar) in the order information and re-signed the message. Then the TP-SDK successfully accepted the order information and prompted its payment Activity, asking for paying this order with one *yuan*. After we paid this order, the order Activity in the ***MA*** showed that the order is successfully paid with **the original price**. And the waiter served the dishes to us after a while without any doubts. Even after our dinner, our "*less paid*" bill did not attract any awareness (we informed the restaurant later and paid for the meal in the original price, and informed the vulnerabilities to the app developer), which indicates that the merchant fails to confirm the notified order details.

For the second app, we leveraged the flaws similar to that in the first app. The only difference is the app leaks its Badpay shared secret key instead of the private key of AliPay. We observed that this app allowed user to charge for his/her account beforehand, and each time the user wanted to order the service, he/she could directly pay a certain amount of fee for each clothes using the pre-charged cash. So this time our penetration test aimed at the transaction of charging. Like the pentest in restaurant, we used the similar way to perform the attack and successfully charged one hundred *yuan* into the account with only one *yuan* paid from our Badpay account. And then we spent the money in the account to place an order and succeeded in washing our clothes without the merchant's awareness or doubts. After that, we informed the merchant the vulnerabilities and paid for the real price of the order.

The attacks against those two apps prove that order tampering causes money loss and makes influence not only to online e-stores but also to offline store in real life. Also, we find that the security flaws of the two apps (leaking their keys in app, failing to re-confirm the details of notified order to ***CS***) are pervasive in many other ***MA***s. In other words, such attacks can be performed to many other merchants.

2) Order Substituting: We proved that *Order Substituting* attack can be **automated** and let user pay for the attacker's order without awareness. We employed the attack on a wireless router of our local area network. Since we control the router, we can conduct MITM attack to the devices in the same LAN. We set a MITM proxy on the router to replace self-signed HTTPS certificate to the original one and decrypt the content of HTTPS connection.

The victim app in this case is a popular e-Book Reader with over 20 million downloads. Users can purchase non-free e-books in this app via payment channels of all four cashiers. We take the BadPay for the case. The app use HTTP connections when users browse book lists in the app. When users want to pay for a book then the connection will turn to HTTPS but the app fails to check the HTTPS certificate correctly. So our proxy can intercept, eavesdrop and tamper the connection. Once a user orders a book and is about to pay, our proxy extracts from the network traffic which book the user want to buy and the order information including price (10 *yuan* in the case). Then our proxy places a new order request using **another** app (a take-out food order app) to buy a burger which is also 10 *yuan*, and get the order information from the ***MS*** without paying for the burger in the latter step. Instead, the proxy intercepts the payment order response of the book and substitutes it with the attacker's burger order. As a result, the ***MA*** receives the replaced order response and prompts the user with its payment Activity of the TP-SDK. Notice that information on the payment Activity of BadPay only including price, are exactly the same as the price of the book. User cannot distinguish this replaced order and is cheated to pay for it. Thus after the payment, the burger is paid and delivered to us while the e-book is still kept unpaid. When they paid for the attacker's order and found their own orders are still unpaid, they just believe it is the delay of the ***CS*** that leads to a temporary unpaid status. Imagine that if the take-out food order app here becomes a malicious ***MA*** controlled by the attacker, then the attacker can easily generate an order in arbitrary price according to the victim's order and substitute it. Thus, after user pay for it, the money is directly transferred to the attacker's account. Even with those TP-SDKs that display the merchant name (WexPay and UniPay), the attacker can still cheat users to pay for attacker's order in the same app.

3) Unauthorized Querying: KEY Leakage directly leads to *Unauthorized Query* attack to merchants as we describe in Section III-F. A typical case is the unauthorized querying of arbitrary orders. We take WexPay as an example. Hundreds of ***MA*** leak their secret keys, and we make use of the leaked key to download all bills of each day of the merchant by simply making a request to a specific URL address[3] provided by WexPay, along with traversing the *'bill_date'* parameters. Our result includes various merchants whose cash flow of everyday ranging from millions of *yuan* (online luxury goods store) to tens of *yuan* (social network app). The bill also contains the details of every transaction in that day, including payer, paying bank, discount, etc. Obviously, all these information should be confidential to any unauthorized visitors.

4) App with Multiple Vulnerabilities: Merchant may violate multiple security rules and the relevant transaction is vulnerable to not only a single type of attack. To illustrate, we demonstrate how to acquire free movie tickets in different ways by exploiting a movie ticket ordering app with an approximate 10 million users. The chosen app allows users to select the cinema, the movie, and the seats they want, and then buy the movie tickets online via in-app payment. After the payment users will get a ticket code and when they get to the cinema, they can enter the ticket-code on an automated ticket machine to get the physical movie tickets.

Several security flaws are detected in this app. The problem occurs when users pay for tickets via WexPay in the app. The app commits *Local Ordering* mistake and thus, also exposes the secret key and notify URL in app. Moreover, the app is also proved to miss the notified payment confirmation and signature validation. We perform three kinds of attacks to buy movie tickets. In the first attack, we hook the order-generation function based on Xposed in the app and tamper the price to a particular value to one *yuan*. Thus we pay for it via WexPay with only one *yuan* (or even cheaper if we wish). In the second attack, we do not tamper the order but follow all the normal step until the app invokes the WexPay's payment Activity to ask us for the payment password to paying for the ticket. Then we terminate the following steps and directly forged a legal payment notification of the WexPay to it server. In the third attack, we repeat the second attack but just send a forged notification message with incorrect signature to the ***MS*** and the server accepts it. All attacks lead us to available ticket-codes successfully. We did use the acquired ticket-codes to fetch the physical tickets at different cinemas and watched the movie.

Moreover, before informing the merchant and repaying the tickets we have bought, we wait for a certain period (thus the merchant could check the collection of all past orders periodically sent from the cashier) to check whether the merchant verifies it. In spite of this, until we explain our behavior to the merchant, they still have no idea about what has happened. It proves that the attacks can be performed repeatedly without the merchants' awareness by an intentional attackers in real life.

VI. Ethical Consideration

We carefully designed our experiments to avoid ethical problem. First, we reported all our findings and the behaviors we performed during the experiments to the related parties and did what we could do to help them improve the systems. Our effort was appreciated by these organizations. In detail, we reported the mistakes in documents/sample codes to WexPay, AliPay and BadPay. All of them have fixed and updated it. For instance, [2] shows the updated payment process figure. The original figure told the developers to generate and sign the payment order in client app (as we mentioned in Section V-E), which is obviously insecure. All the three cashiers expressed their gratitude to us. Also we detected flaws (such as missing order signature validation) of several merchant servers described in Section V-C. We reported these flaws and explained our behaviors to these influenced merchants as soon as we carried out our experiments, and helped them to fix these vulnerabilities. Since hundreds of merchants suffer flaws in their APPs as we mentioned in Section V-A, it would be too much work for us to inform all of the merchants. We decided to report the vulnerable ***MA*** list to the Security Response Center of Tencent, Ant Financial and Baidu, who are responsible for the security of their whole payment ecosystem (WexPay, AliPay, and BadPay). They informed all the related merchants of their security risks, revoked leaked *KEY* and renewed them. We use a Web API provided by WexPay as oracle to help finding the leaked *KEY* in ***MA***, which need to brute-force the parameters of the API and may induce potentially heavy load to WexPay's server. We did restrict the frequency and times of invoking the API to avoid potential denied of service attack against the server. Also if the candidates of three parameters (mentioned in Section IV-B) are too many, we first filtered it manually to reduce the test set. After we described our detecting method to WexPay, they confirmed the issue and planned to impose some constraints to invoking the API in future. They even expressed their appreciation to our effective and efficient *KEY Leakage* detection method for large-scale APPs. Second, we ensure no financial damage was inflicted

[3]'https://api.mch.weixin.qq.com/pay/downloadbill

upon the merchants by returning items or re-paying the unpaid orders, etc. For the victim user in the *Order Substituting* attack (described in Section V-F2), we later paid for the e-book order for him, who is actually a colleague of us. We made use of the result of downloaded history bills of merchants to evaluate the feasibility of *Unauthorized Query* attack, and help to detect *KEY Leakage* in ***MA*** using WexPay. We not only described our detecting method in detail to merchants, but also deleted all these data at once to avoid further exposures.

VII. Discussion

Although our large scale analysis reveals that the ratio of flawed in-app payment implementations is surprising, we have to point out that we underestimate the actual danger. During the TP-SDK identification, our methodology is based on static code feature. We classify many apps protected by code packing techniques as not using any TP-SDK, while they do integrated TP-SDKs. In the detection of flaws in ***MA***, we adopt static analysis mainly focusing on analyzing DEX and resource files in APK. Those implementations with native *.so* are not analyzed. Since our work presents a systematic approach to detect those vulnerabilities, we believe the analysts of merchants and cashiers could adopt our approach to audit their products before releasing.

Our security analysis mainly focuses on the interfaces of multi-party involvement in the third-party in-app payment. We pay less attention to the attacks or flaws only involving single party (traditional user-to-merchant payment model), for instance, the merchant order tampering, or denied-of-service attack on order ID or transaction ID.

VIII. Related Work

A. Insecure Android Third-party SDK

Meanwhile, vulnerabilities or threats introduced by third-party SDKs in Android applications have also been studied by many researchers. Chen *et al.* [13] studied on potentially-harmful libraries across Android and iOS through clustering similar packages to identify libraries and analyzing them using AV systems to find those libs. Wang *et al.* [26] identified serious authentication and authorization flaws in applications that integrate Single-Sign-On SDKs. Li *et al.* [16] aims to understand and analyze the security hazards imported by Push service in Android applications. Wang *et al.* [24] and [12] demystified and assessed the vulnerabilities of OAuth protocol on mobile platform, which often introduced by third-party providers as SDKs in Android applications. However, the payment SDK in Android applications has never been studied before. We propose a comprehensive methodology to detect various security rule violations in those apps who embed third-party payment SDKs. All of these flaws lead to serious consequence and result in financial loss for different parties involved.

B. Android App Vulnerabilities

The security analysis of vulnerabilities in Android application has also become hot spot these years, with the dramatic growth in mobile users. It's common that misuse of security libraries leads to flaws in apps. Egele *et al.* [14] studied the misuse of cryptographic API in Android applications and summarized basic rules in using cryptographic libraries. Fahl *et al.* [15] reveal several types of flaws in the use of SSL/TLS by Android application and detect potential SSL vulnerabilities. SMV-Hunter [20] combined static analysis and dynamic validation to improve the precision of detecting unsafe SSL use. Reaves *et al.* [19] performed security analysis on SSL/TLS certificate verification, non-standard cryptography, access control and information leak of branchless banking applications, and found significant vulnerabilities. Different from all of these work, our analysis of security flaws caused by apps integrating third party in-app payment libraries. We reveal that these flaws during in-app payment caused both by merchant apps and third-party payment SDK providers. Among these flaws, *KEY Leakage* has been studied before. Both PlayDrone [23] and CredMiner [29] try to detect token exposure of AWS and OAuth in Android applications. However, we adopt a more efficient and accurate methodology, combining local program analysis and a remote Web API, to detect such flaws in third-party payment. Besides, our work covers the security threats and flawed implementations throughout the whole payment process, rather than focusing on a particular type of vulnerability detection.

C. E-Commerce Vulnerabilities

The security analysis of e-commerce and payment has attracted the attention of researchers in recent years, since vulnerabilities may cause great impact and financial loss. As far as we know, the only similar work is implemented by VirtualSwindle [17], which can perform an automatic attack against in-app billing service. However, it seems to be just a small part of our work. First, the in-app billing service described in the paper is just one scenario of the in-app third-party payment. The service is simpler and less popular compared to our research targets. Second, VirtualSwindle can only launch one type of attack and the adversary model assumed in the paper is too limited. However, our work describes four types of threats and attack model is more diversified. Third, only 85 Android apps were studied in the paper compared with the thousands of samples in our work. Overall, we perform a more large-scale and systematic analysis to third-party in-app payment. Our work focuses on finding all flawed implementations throughout the whole payment process, not only launching one type of attack.

Wang *et al.* [25] are the first to analyze logic vulnerabilities in Cashier-as-a-Service based web stores, and found several logic flaws manually. Sun *et al.*[22] propose to detect logical vulnerabilities in e-commerce application through static analysis of available program code. Pellegrino *et al.* [18] proposed the idea of black-box detection of logical vulnerabilities in e-shopping applications Sudhodanan *et al.* [21] propose an automatic technique based on attack patterns for black-box, security testing of multi-party web applications. InteGuard [27] offers dynamic protection of third-party web service integrations including cashier service in merchants' websites.

All of the work mainly target on e-commerce in web application, while we focus on mobile platform. Our detection object is mainly flawed implementations in Android applications using third-party in-app payment SDKs, since apps in mobile OS redefine the trust boundaries. Different from third-party payment of Web service, client application with TP-

SDK embedded plays an important role in mobile payment scene. Thus our work involves large scale detection on flawed app according to mobile application's characteristics. while previous work focus on server flaw detecting, which can hardly be large scale. Besides, we also present security rules to every party of in-app payment scene, analyze and question the cause of these payment flaws in-depth.

IX. Conclusion

Insecure In-app payment is becoming a main threat to mobile ecosystem as more and more online transactions are transferring from website to app. Different from traditional web payment, in-app payment involves more sophisticated implementation details and the process is often obscure. To demystify processes of popular in-app payments and reveal potential security risks, we conduct a comprehensive analysis on mainstream third-party in-app payment schemes in Android Apps. Our analysis investigates implementations of four in-app payments and concludes a series of security rules that should be obeyed. We not only pinpoint the serious consequence of violating security rules, but also detect these flawed implementations. Our statistics paint a sobering picture–hundreds of apps integrated with third-party in-app payment SDKs are vulnerable. Besides, our further investigation indicates that cashier is mainly blame for these flawed implementations. We hope our study can remind and guide developers of both merchants and cashiers to build more secure in-app payments.

Acknowledgment

We would like to thank our shepherd, Kevin Butler, and the anonymous reviewers for their insightful comments that greatly helped improve the manuscript of this paper. This work is partially supported by the Technology Project of Shanghai Science and Technology Commission under Grants No.15511103002, and the Key Program of National Natural Science Foundation of China (Grants No. U1636217). The corresponding author of this paper is Yuanyuan Zhang.

References

[1] AliPay. https://open.alipay.com.

[2] AliPay payment process. https://doc.open.alipay.com/doc2/detail?treeId=59articleId=103658docType=1.

[3] AndroGuard. https://github.com/androguard/androguard.

[4] BadPay. https://b.baifubao.com.

[5] Myapp Android Market. http://android.myapp.com/.

[6] OAuth. http://oauth.net/.

[7] ProGuard. http://proguard.sourceforge.net/.

[8] UniPay. https://merchant.unionpay.com/join/index.

[9] WexPay. https://pay.weixin.qq.com.

[10] Xposed. http://repo.xposed.info/.

[11] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You get where youre looking for: The impact of information sources on code security. In *Proc. of IEEE Symposium on Security and Privacy, 37nd*, 2016.

[12] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[13] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, and Y. Zhang. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *Proc. of IEEE Symposium on Security and Privacy, 37th*, 2016.

[14] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proc. of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[15] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[16] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proc. of the 21st ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[17] C. Mulliner, W. Robertson, and E. Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[18] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS*, 2014.

[19] 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 *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[20] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS*, 2014.

[21] A. Sudhodanan, A. Armando, R. Carbone, and L. Compagna. Attack patterns for black-box security testing of multi-party web applications. In *Proc. of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[22] F. Sun, L. Xu, and Z. Su. Detecting logic vulnerabilities in e-commerce applications. In *Proc. of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.

[23] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, 2014.

[24] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[25] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *Proc. of IEEE Symposium on Security and Privacy, 32nd*, 2011.

[26] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security*, 2013.

[27] L. Xing, Y. Chen, X. Wang, and S. Chen. Integuard: Toward automatic protection of third-party web service integrations. In *Proc. of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[28] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Research in Attacks, Intrusions, and Defenses*. 2015.

[29] Y. Zhou, L. Wu, Z. Wang, and X. Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.