Inferring Implicit Assumptions and Correct Usage of Mobile Payment Protocols

Quanqi Ye[1], Guangdong Bai[2(✉)], Naipeng Dong[1], and Jin Song Dong[1,3]

[1] National University of Singapore, Singapore, Singapore
yequanqi@u.nus.edu, {dcsdn,dcsdjs}@nus.edu.sg
[2] Singapore Institute of Technology, Singapore, Singapore
guangdong.bai@singaporetech.edu.sg
[3] Griffith University, Nathan, Australia

Abstract. Although mobile shopping has risen rapidly as mobile devices become the dominant portal to the Internet, it remains challenging for a developer of mobile shopping Apps to implement a correct and secure payment protocol. This can be partly attributed to the misunderstanding, confusion of responsibility and implicit assumptions among multiple separate participants of the payment protocols, which involve at least users, merchants and third-party cashiers (e.g., PayPal). In addition, the documentation of the payment SDK which is written in informal natural languages is often inaccurate, ambiguous and incomplete, such that the developers might be confused. In this paper, we seek to infer the correct usage and hidden assumptions of the most commonly used mobile payment libraries, i.e., PayPal and Visa Checkout. Our approach starts with building mobile checkout systems strictly following the documents of PayPal SDK and Visa Checkout SDK. Afterwards, we propose an algorithm to automatically generate test cases embedding different attacker models to check the correctness and security of the payment procedure. During the testing, our algorithm analyzes the security violations so as to infer the correct usage of these payment libraries. Using our approach, we have successfully found several non-trivial hidden assumptions and bugs in these two payment libraries.

Keywords: Mobile payment · Payment protocol · Protocol extraction

1 Introduction

Mobile shopping is becoming increasingly popular as it brings great convenience to people and it has become an indispensable part of their daily lives [9]. Numerous merchants start providing mobile shopping Apps as their main portals [12]. Mobile payment, which allows users[1] to pay remotely on their mobile devices, is a critical procedure in mobile shopping. A small vulnerability in the payment

[1] User of the merchant App, i.e., customer.

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 469–488, 2018.
https://doi.org/10.1007/978-3-319-78813-5_24

protocol may cause severe financial lose for users and merchants, as revealed by previous research [14,15,17].

Existing studies on online payment mainly focus on the desktop platform rather than the mobile platform. We highlight that online payment protocols intended for desktop platform cannot be directly applied to mobile platform, due to the disparity of these two platforms, especially w.r.t. security [13]. First, mobile devices have limited computation capability and battery power, and thus it is hard to deploy a malware detection system as powerful as on the desktop. Second, mobile devices are small in screen size, such that particular information on security may be omitted for the sake of usability. For example, the users may not realize that the website they are browsing is not the intended one as the browser often hides the address bar to save space. Third, desktop has deployed well-evolved security mechanisms to control access to security-critical resources, whereas few similar mechanism has been built on mobile platform.

Mobile payment normally involves multiple parties, including at least customers, merchants and third-party cashiers (TPC for short hereafter) such as PayPal. These parties interact with each other following the underlying payment protocol, which is typically designed by the TPC. The problem is that the merchant App[2] developers and the protocol designers are usually different parties. Misunderstanding to certain steps in the protocol, confusion of responsibility and wrong assumptions on the responses from other parties are unavoidable. For example, to facilitate use of the payment protocol, the TPC usually provides App developers with an SDK encapsulating the protocol implementation. This eases the use, but it may exacerbate misunderstanding because the details of the protocol are hidden. Even worse, the documentation of the SDK, which is often written in natural languages, may be inaccurate, ambiguous and incomplete. Consequentially, it is highly likely that the merchant developers fail to correctly implement the payment protocol.

In this paper, we propose a systematic approach to identify the correct usage and the implicit assumptions which the developers of merchant Apps must follow and be aware of to implement a secure payment system. To this end, for each payment SDK, we first build a testbed shopping system which includes both a front-end merchant App embedding the SDK, and a back-end merchant server which processes the payment issued from the App. To minimize the bugs caused by our mis-integration, the testbed system is built by strictly following the official documents and TPC's sample code. By applying protocol extraction techniques [6,20] on the testbed systems, we infer implementation-level payment protocols. These protocols are used to automatically generate test cases for dynamic testing. During the testing, we check whether the payment is secure by observing the integrity of four key elements in a payment, given that the integrity is the key property of a payment protocol [17]. Whenever the integrity is violated, we manually study the test cases and execution traces to learn the cause of the violation. Through the analysis, we are able to infer the correct

[2] In this paper, we use *merchant App* to indicate the front-end App running on customer's mobile device and *merchant server* the back-end server.

usage and hidden assumptions to build a secure payment system. By applying our approach on the Android SDKs of two widely-used TPCs, i.e. PayPal and Visa Checkout, we have successfully found several non-trivial usage rules and hidden assumptions. Our approach detects three bugs in these two payment libraries.

We summarize our contributions as follows.

- We extract PayPal and Visa Checkout's mobile payment protocols which can be a reference for other researchers.
- By applying our approach, we have found and reported **three** confirmed bugs in PayPal SDK.
- We summarize **three** rules and **five** implicit assumptions in using PayPal and Visa. These are beneficial to the merchant App developers in building a secure payment system.

2 Background

To ease the understanding, this section briefly introduces the background in mobile payment.

2.1 A General Process of Mobile Payment

Although different TPCs may have different payment protocol implementations, they generally follow a similar process in terms of mobile payment. In this section, we use PayPal payment as an example to introduce such a mobile payment process, as shown in Fig. 1.

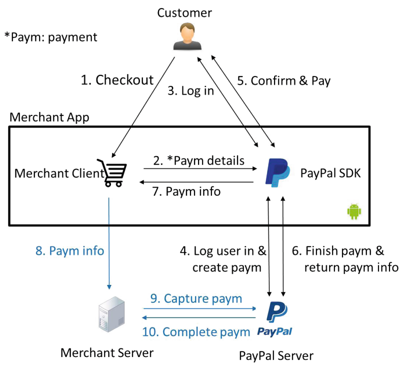


Fig. 1. Correct process to capture payment

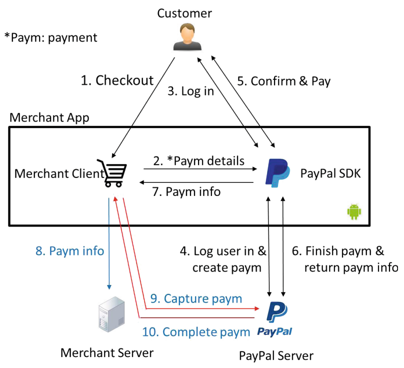


Fig. 2. Dangerous process to capture payment

S1. After ordering, the user clicks on the "Checkout" button to initiate a checkout process (step 1).

S2. The merchant client invokes the PayPal SDK and passes it the payment details (step 2).
S3. PayPal SDK shows a login dialog for the user to login (step 3).
S4. After receiving login credentials, PayPal SDK sends them along with the payment details to PayPal server for verification. After verification, PayPal server creates the payment and sends it back to the PayPal SDK, which then shows the user a button for payment authorization (step 4).
S5. The user confirms the payment details (e.g., amount) and authorizes the payment (step 5).
S6. Upon receiving authorization, PayPal SDK forwards it to PayPal server and the PayPal server sends the payment result back to PayPal SDK (step 6).
S7. The PayPal SDK sends the payment info to the merchant client (step 7).
S8. The merchant client sends (optional) the payment information to its merchant server (step 8).
S9. The merchant server captures the payment with PayPal server using the payment information from merchant App (step 9).
S10. PayPal server replies merchant server with payment completed response (step 10).

2.2 Special Features of PayPal SDK

Despite of the general payment process, each TPC may have special features. In this section, we introduce such special features in PayPal SDK which are relevant to the security of the protocol. PayPal's mobile payment process can be further divided into the following three types, depending on the timing of authorizing the payment and the timing that the merchant captures[3] the payment.

Single Payment. It represents a one-time payment. The single payment can be further divided into three categories.

– Immediate payment, where the user authorizes the payment immediately and the merchant captures the payment immediately.
– Authorization payment, where the user authorizes the payment immediately and the merchant may capture it later.
– Order payment, which is used in the case that the user authorizes the payment in advance when the actual item for sale is not ready yet. Once the item is received by the user, the merchant can capture payment at any time.

Future Payment. It allows the user to authorize the merchant to create and capture payment in the future. In other words, once authorized, the merchant can create and capture payment for multiple times.

Profile Sharing. It is used to share user's profile information in PayPal server to the merchant App. This seems not a payment feature. However, as we show in Sect. 8, this feature actually allows the merchant to capture payment from user's account.

[3] *Capture* is a term used in the PayPal documentation, meaning that the merchant completes/cashes the payment.

2.3 An Example of Dangerous Usage

Although the processes of the all three types of single payment in PayPal's Android SDK are the same, there are subtle differences among them. For example, in both authorization payment and order payment, after the user authorizes a payment, the protocol requires the merchant to immediately capture the payment, whereas in the immediate payment, the merchant does not have to do so. Therefore, to guarantee the payment is captured successfully, the following rule must be complied by the merchant.

#1. *For authorization payment and order payment, the merchant server* ***must*** *subsequently capture the payment from the* ***merchant server***[4].

Following this rule, the merchant server needs to actively perform step 9 and step 10 as shown in Fig. 1 to ensure the authorization or order payment is completed correctly.

A dangerous usage of the protocol example is shown in Fig. 2. In that scenario, the payment capturing request is performed by the App, while it should be done by the merchant server as shown in Fig. 1. The reason is that the environment in which the merchant App resides is out of the control of the merchant, and thus it should be considered as insecure. This is a case that developers without security domain knowledge may not be aware of. If rule #1 is not followed, an attacker could intercept the messages sent from the merchant App to TPC and forges a response from the TPC. Compared to the App side, the merchant server is normally under control of the merchant, and thus performing the payment capturing request on the server side is relatively more secure.

The cause of this security issue is as follows. PayPal may assume that it is the merchant's responsibility to ensure the capturing request is sent from merchant server, while the merchant may assume that the protocol is secure and he/she may not realize it is dangerous to capture payment from merchant App.

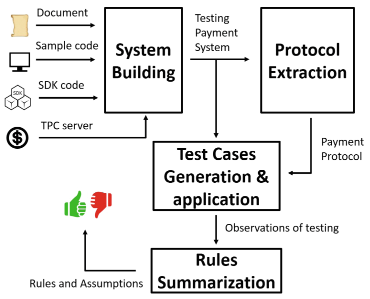


Fig. 3. Method overview

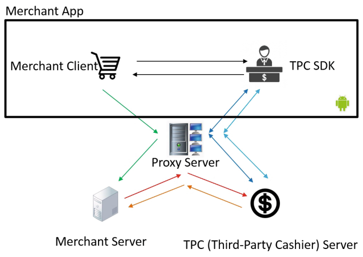


Fig. 4. Testbed mobile checkout system with proxy server

[4] We find that this rule also applies for Visa Checkout, in which there is no immediate payment, and the merchant is required to actively capture the payment.

This example demonstrates that, because of such hidden assumptions and confusions of responsibility among participants of the mobile payment protocol, security problems in this scenario are inevitable. This motivates us to identify the hidden assumptions and the correct usage of a payment protocol.

3 Method Overview

In this section, we introduce our overall method. As shown in Fig. 3, our method includes the following steps.

System Building. Taking the documentations from TPC and the sample code (with SDK provided by TPC) as input, we first build the testbed payment systems following the instructions from documents. We mainly need to incorporate two parts - the merchant and the TPC server. For the TPC server, we use the sandbox environment, for example [3], to avoid finance cost to any real merchants during testing. We remark that the sandbox environment is a separate server that provides mirrored functionalities of the live environment that a TPC server uses for real-world applications. All the functionalities needed in this work from live environment can be found in the sandbox environment. Hence, the rules inferred in the sandbox environment are also applicable to the live environment. For the merchant, following the work flow introduced by the official documents, we build both merchant App and merchant server with essential functions needed to accept payment.

Protocol Extraction. In order to understand how the TPC SDKs create a payment and what information is necessary for creating a payment which the SDKs' documents do not cover, we need to perform the protocol extraction to infer the underlying payment protocols. These protocols specify the exact actions of each participant. They are used for generating test cases under different attacker models.

Test-based Rules Summarization. In this step, we infer rules during the dynamic testing. To this end, we propose an algorithm to guide the testing process. The algorithm generates different test cases incorporating various attacker models. It then drives execution of the system by feeding it the generated test cases. The essential idea of the algorithm is to enumerate what an malicious participant can do. When executing each test case, the protocol may either terminate or end normally. In the former case, the attack may have been prevent by the protocol, so we do not further examine it. For the latter case, we check the integrity after the execution finishes. If the integrity is breached, there may be a flaw in the system, and we manually examine it to figure out the cause of the problem and then summarize protocol usage rules or assumptions.

4 System Building

In this section, we introduce the testbed system building. The architecture of the testbed system is shown in Fig. 4. It includes a merchant App including the merchant client and a TPC SDK, a merchant server and a TPC server. We set up two sets of testbed systems integrating respectively PayPal and Visa Checkout SDK.

Merchant App. For each of the merchant Apps, we reuse most of the code from the samples provided by TPC. To simplify the merchant App, we omit the item selection process and provide just two buttons representing two different items with different prices. When one of the buttons is clicked, the user is redirected to TPC SDK to finish the rest of payment protocol. After that, the merchant client[5] receives the payment information returned from TPC SDK and it can either send the information to the merchant server, or perform capture directly depending on the test case. For example, if it is the single payment in PayPal, the merchant client transmits the payment ID back to the merchant server, whereas if it is the future payment, it transmits the authorization code back to the merchant server.

Merchant server. In different test case, the merchant App may send different payment information to the merchant server, which then accordingly perform one or more of the following actions.

For PayPal:

– *Doing nothing.* This action represents that the merchant server does not need to perform any further action. This may happen if the merchant client has captured the payment.
– *Retrieving payment details.* This action represents that the merchant server queries the detailed payment information from the PayPal server, such as amount and capturing status.
– *Verifying payment information.* This action represents that the merchant server validates the payment information retrieved from the PayPal server.
– *Capturing payment.* This action represents that the merchant server captures the payment with PayPal server by providing the payment information received from the merchant client.

For Visa Checkout:

– *Doing nothing.* This action represents the same as in PayPal.
– *Retrieving payment details.* This action represents that after receiving payment ID from merchant client, the merchant server uses it to retrieves the encrypted payment information from Visa server.
– *Decrypting payload.* This action represents that the merchant server decrypts the encrypted payment information returned from Visa server.
– *Updating payment information.* This action represents that the merchant server updates the payment information to the Visa server after validation.

[5] The portion of code that is implemented by merchant developers which is representing with a carte label in Fig. 1.

We create a profile for each of the two merchant Apps in the respective TPC servers. The TPC servers generate two unique artifacts for each of the Apps: shared secret and merchant ID (They may be named differently in different TPCs). The shared secret is used to authenticate the merchant and the merchant ID is used to identify the merchant App. In summary, we build two sets of systems which incorporate PayPal SDK and Visa Checkout SDK, respectively. We remark that these testbed systems are representative as we build them based on the official documentations and sample code which can reflect the actual situations where developers are facing as they develop Apps that integrate TPCs.

5 Protocol Extraction

In order to generate test cases for the testbed system, we need to extract the baseline payment protocol from PayPal SDK and Visa Checkout SDK to understand how the payments are created and completed by the protocols. Our approach extracts the protocol from the messages exchanged by the participants during the protocol execution. The messages we take as input include application-layer messages, such as HTTP messages and HTTPS messages. In a nutshell, our extraction approach works as the following steps.

– **Protocol Message Capturing.** During protocol execution, messages are exchanged through the network channels. We capture these messages as traces from our testbed systems for our analysis.
– **Trace Refinement.** The raw traces captured are typically complicated and contain many redundant parameters which are not relevant to our analysis. Therefore, in this step, we remove redundant parameters to get refined traces.
– **Protocol Interpretation.** After trace refinement, we get the baseline payment protocols. However, the concrete semantics of the messages are still unclear for us to understand the precise behaviors of the SDKs. For example, some messages stand for payment creation while some stand for payment update. Therefore, in this step, we aim to identify the semantics of these messages by manual analysis.

5.1 Protocol Message Capturing

To capture the raw protocol messages in the network channels, we need to deploy a proxy server in the network channels intercepting the messages coming in and going out from merchant App. The proxy is not part of our testbed mobile checkout systems, but it facilitates protocol refinement and can simulate the network attacker during the dynamic testing.

Figure 4 shows the testbed mobile checkout system with the proxy deployed. The proxy server is deployed between the merchant App and the two servers (merchant server and TPC server) such that all messages sent out by merchant App can be captured and even changed (for trace refinement). In this

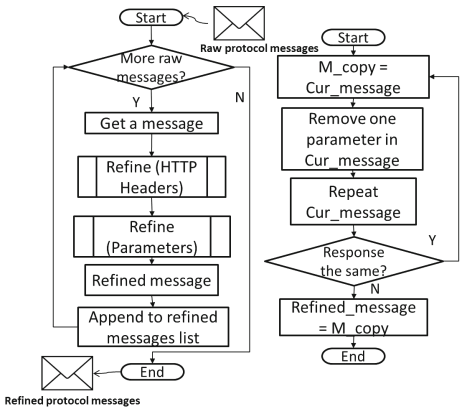


Fig. 5. Trace refinement procedure (The sub-procedure on the right side is the detailed procedure for the "Refine" procedure on the left.)

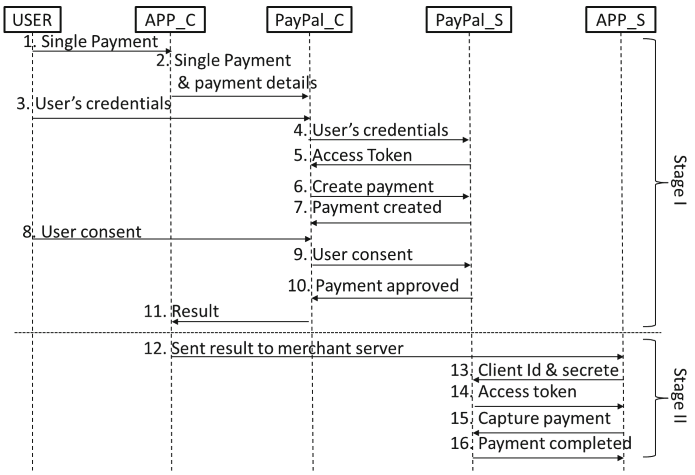


Fig. 6. Single payment protocol (I: The first stage of creating and user consenting the payment. II: The second stage that merchant server verifies or captures the payment.)

work, we only consider the attacks which can control the client-side Apps and communication channels. Therefore, we skip the communication between the merchant server and the TPC server.

After the deployment of the proxy server, once the protocol is executed, all the communications coming in and going out of the mobile device can be recorded by our proxy server. We execute all possible payment methods in PayPal and Visa Checkout such that enough information regarding the protocol can be preserved in the captured traces.

5.2 Trace Refinement

The trace refinement procedure is shown on the left hand side of Fig. 5. The procedure takes the raw protocol messages as input and then outputs the refined traces without redundant parameters. The concrete message refinement procedure is shown on the right hand side of Fig. 5. By using our proxy, we keep replaying every message with one parameter temporarily removed. If the modified message leads to the same response as the original message, the removed parameter is a redundant parameter to the protocol. Hence, we can remove permanently that parameter. We keep doing it until we cannot remove any remaining parameter. The final message therefore is a concise message which excludes all redundant parameters while still produces the same response as the original message.

We iterate the refining procedure on all the raw messages and obtain their refined versions. Eventually, all the messages refined make up the refined trace. In some cases, a replayed message is not accepted when the message carries a parameter that can only be used for once, e.g. timestamp. To address this, we repeat the whole protocol in order to fuzz for the single non-repeatable message.

5.3 Protocol Interpretation

After refining the protocol messages, we then analyze the purpose of each message. Messages sent to different url endpoints with different parameters correspond to invoking different APIs/commands in TPC server to log user in, create or update payment.

We summarize the identified TPC API endpoints from messages of TPC SDKs and messages of the merchant server. We find that different API endpoints serve as different purposes/commands in the protocols.

From the communication trace between PayPal SDK and PayPal server, we observe that different payment methods in single payment use the same set of API endpoints and follow the same sequence when invoking the APIs. We also observe that future payment and profile sharing share the same set of API endpoints and also follow the same sequence. The difference is the intent of the final consent made by future payment and profile sharing. For future Payment, the consent is to authorize the merchant to make payment in the future, whereas profile sharing authorizes the merchant to retrieve personal information from PayPal.

The final outputs of protocol inference are the baseline protocols for different payment methods in PayPal and Visa Checkout. We summarize them as follows. In the protocol, we denote merchant App as `APP_C`, PayPal SDK as `PayPal_C`, PayPal server as `PayPal_S`, the merchant server as `APP_S`, Visa checkout SDK as `Visa_C`, Visa server as `Visa_S`.

– **Single Payment.** As shown in Fig. 6, the first stage of authorization payment and order payment are the same. There is subtle difference at the second stage. In the immediate payment, the merchant server does not have to capture the

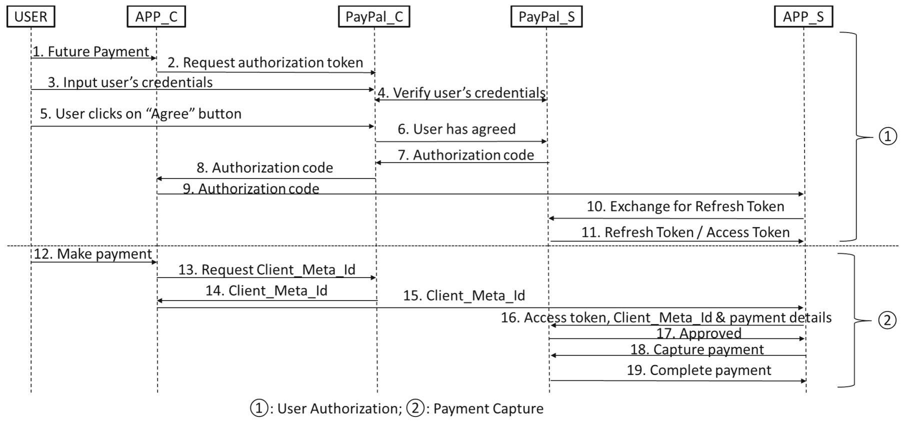

USER
APP_C
PayPal_C
PayPal_S
APP_S
1. Future Payment
2. Request authorization token
3. Input user's credentials
4. Verify user's credentials
5. User clicks on "Agree" button
6. User has agreed
7. Authorization code
8. Authorization code
9. Authorization code
10. Exchange for Refresh Token
11. Refresh Token / Access Token
12. Make payment
13. Request Client_Meta_Id
14. Client_Meta_Id
15. Client_Meta_Id
16. Access token, Client_Meta_Id & payment details
17. Approved
18. Capture payment
19. Complete payment
①
②
①: User Authorization; ②: Payment Capture

Fig. 7. Future payment protocol

payment. Rather, it only has to verify whether the payment details are correct. To this end, it uses its merchant ID and the shared secret to obtain an access token and makes a direct server-to-server API request to check if the payment details are exactly the same as the one returned from the merchant App.

– **Future Payment.** The procedure of future payment is shown in Fig. 7. We highlight that whenever the refresh token which the merchant obtains using the authorization code is still valid, it can be used by the merchant to create and capture payment. From the official document on PayPal SDK, we know that although the authorization code is short-lived, the refresh token is long-lived and lasts for 10 years [2]. That means that when the refresh token is obtained, the merchant can create and capture the payment within 10 years.
– **Profile Sharing.** The procedure for profile sharing is highly similar to that of future payment. The difference only occurs at the last step. Merchant server makes request to different API endpoints to retrieve user's profile information rather than to create and capture payment as in future payment.
– **Visa Checkout.** As shown in Fig. 8, most steps of Visa Checkout are similar to PayPal's immediate payment. However, in the last two steps (step 9 and step 10), apart from the *callID*, Visa also returns the *encKey* and *encData* which are encrypted data containing the payment details. Merchant needs to first decrypt the *encKey* using the shared secret and then uses the decrypted *encKey* to decrypt the *encData* to get the payment details.

6 System Testing

Based on the extracted protocols, we can generate test cases to dynamically test our testbed systems. During the test case generation, we consider two types of attackers, i.e., the malicious user and the malicious merchant, each of which has specific attack capabilities. Given that the integrity is the predominant property in payment protocols, our dynamic testing mainly targets this property.

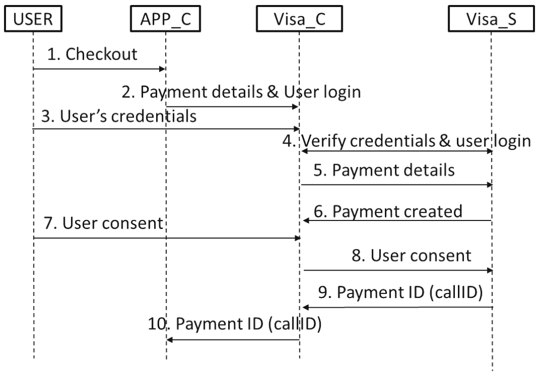


Fig. 8. Visa checkout protocol

6.1 Attacker Models

During the test case generation, we consider the following two attacker models.

Malicious User. The malicious user stands for such a attacker that controls the mobile device where the merchant App is running on. This attacker attempts to shop for free or pays less for the order[6], and the victim of this attacker model is the merchant. We list the capabilities of this attacker as follows.

- To control the network channels of the merchant App such that it can change parameter(s) in protocol messages coming in and going out from the device.
- To record, interrupt and replay the messages sent and received by the merchant App.
- To forge a message to merchant server, merchant App or TPC server.

Malicious Merchant. The malicious merchant stands for such a attacker that controls the merchant App and the merchant server. This attacker attempts to overcharge the user, charge the user without authorization and obtain the profile information of user from the TPC. The victim of this attacker model, thus, is the user. We list the capabilities of this attacker as follows.

- To tamper the total amount in the order or user's authorization.
- To abuse obtained token, e.g., invoke particular APIs out of user's intention.
- To inject malicious code in the merchant client and the embedded TPC SDK.

6.2 Integrity of Payment

A payment consists of the following four elements. (1) the ***User*** who initiates a payment, (2) the ***Order*** placed by the user who initiates that payment, (3) the ***Payment*** made by the user, and (4) the ***Merchant*** which the order is placed

[6] The order contains the items the user has ordered and the prices of the items.

Algorithm 1. Test case generation algorithm

```
procedure Test Case Generation
   FOR $M \in$ attack models
      $Bool\ ENDNORMAL == TRUE$
      FOR $P \in$ protocols
         FOR step $S \in P$
            $A = M$.ChooseActions()
            $R = S$.GetActiveRole ()
            IF $R = M$.GetRole()
               $A = R$.GetProtocolAction()
               $A$.Perform()
            ELSE $A = M$.GetRole().GetAction()
               $A$.Perform()
               IF $P$.CanProceed()$!= TRUE$
                  $ENDNORMAL == FALSE$
                  BREAK
               ENDIF
            ENDIF
         ENDFOR
         IF $ENDNORMAL == TRUE$
            CheckIntegrityOfPayment()
         ENDIF
      ENDFOR
   END
```

in and the user should pay. We represent the association of ***Payment***, ***Order***, ***User*** and ***Merchant*** as **POUM**. This association specifies the fact that a user makes a payment for the order to the merchant. Essentially, each transaction can be abstracted as such an association.

To ensure that a payment is conducted in a correct and secure way, the integrity of the **POUM** must be guaranteed. In other words, the integrity of the **POUM** implies that the user has made a payment with correct amount for the intended order to the right merchant. Therefore, after executing the system on each test case, we check the payment's **POUM** from perspective of different parties to ensure that the **POUM** has not been changed by any participant.

6.3 Testing and Evaluation

The algorithm for generating test cases under the above attacker models is shown in Algorithm 1. As shown in the algorithm, during the protocol execution, an honest participant always follows the protocol, while a malicious attacker enumerates the actions it is able to conduct under the capabilities we define in Sect. 6.1. For the malicious user attacker model, we consider user and merchant App as the same role in the protocol execution, given that the mobile device is under the malicious user's control. Therefore, the merchant App in this case should be considered as part of the malicious user. In the malicious merchant attacker model, both merchant server and the merchant App are considered malicious. During the action conducting, the algorithm checks if the protocol can proceed to next step, because the participants may reject the unexpected messages and terminate the protocol. At the end of each protocol execution, we check the integrity of **POUM** to decide whether the test case has revealed a problem of the protocol implementation.

7 Problems Identified and Correct Usages

In this section, we report the identified bugs during system testing and then discuss the correct usages that are summarized from the bugs.

7.1 Identified Bugs

PayPal Android SDK. We find three bugs (shown in Fig. 9) in PayPal payment when we test the system with test cases incorporating the attacker which has compromised the communication channel between the PayPal SDK and PayPal server. This attacker represents several practical system and network attacks. For example, it can be a malicious merchant who incorporates a modified version of SDK to change the parameters; it can be a malicious App which embeds a network proxy (e.g., [8]) and has been installed on the same device as the merchant App; it can be a privileged App which is assigned *root* or *ADB* priviledge [7]; it can be a public WiFi hotspot under attacker's control.

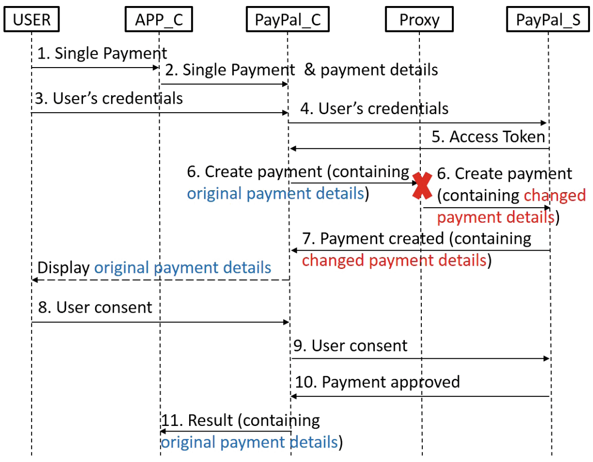
We have reported all the bugs to PayPal who confirmed our findings and stated that the bugs will be fixed in the later version of PayPal Android SDK.

Payment details being Changed. We find that PayPal SDK accesses an API endpoint to create payment and delivers the payment details to the PayPal server. In this step, the attacker modifies the message with different amount and currency. Later, the PayPal SDK displays the payment details to the user and waits for the user to authorize the payment. We observe that when applying the above test case, even after the attacker has changed the payment details in the transmitted messages, the amount displayed to the user remains the one before the attacker changes it. In addition, even the merchant client actively retrieves the payment details by invoking the APIs of the SDK, the returned payment details remain the same as the unchanged one. This implies that even after PayPal's server replies with the changed payment details, PayPal's SDK does NOT update the payment information. This flaw is shown by steps labelled in red in Fig. 9a.

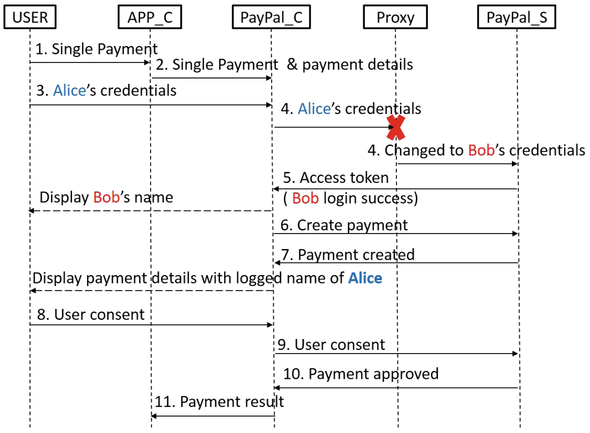
This bug can lead to an attack where a malicious merchant can overcharge an incautious user. For example, a malicious merchant can change the payment amount to a higher number. Since the SDK shows the original payment even after the payment being changed by the merchant App, there is no information for the user to immediately find he/she has been overcharged.

User Credentials being changed. As shown in Fig. 9b, when a user, e.g., Alice, enters her credentials to log into PayPal, the credentials can be changed to Bob's username and Bob's password. In addition, we observe that the Activity in PayPal SDK still shows the username of Alice. This implies that PayPal server never verifies whether the payer in the payment details is the same as the user under authentication.

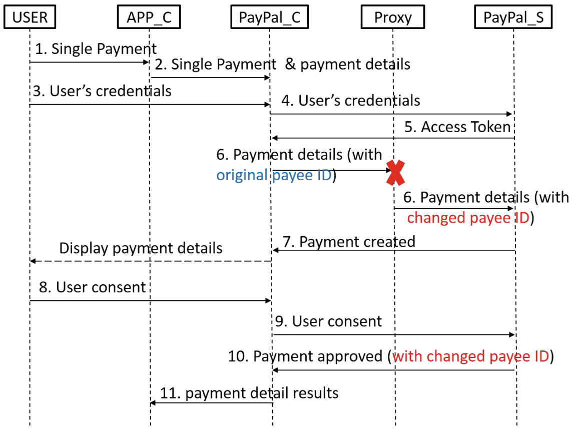
Although this issue may be less harmful to end users than the previous issue, we highlight that the PayPal server should be responsible to verify the consistency of payer and the authenticated user, and the SDK should in all


USER
APP_C
PayPal_C
Proxy
PayPal_S
1. Single Payment
2. Single Payment & payment details
3. User's credentials
4. User's credentials
5. Access Token
6. Create payment (containing original payment details)
6. Create payment (containing changed payment details)
7. Payment created (containing changed payment details)
Display original payment details
8. User consent
9. User consent
10. Payment approved
11. Result (containing original payment details)

(a) Payment details transmitted being changed.



(b) Account credential transmitted being changed.


USER
APP_C
PayPal_C
Proxy
PayPal_S
1. Single Payment
2. Single Payment & payment details
3. User's credentials
4. User's credentials
5. Access Token
6. Payment details (with original payee ID)
6. Payment details (with changed payee ID)
7. Payment created
Display payment details
8. User consent
9. User consent
10. Payment approved (with changed payee ID)
11. payment detail results

(c) Payee (Merchant) ID transmitted being changed.

Fig. 9. Identified bugs in PayPal SDK.

cases check and verify the payment details returned by the server and displays correct information to the end users.

Payee (Merchant) ID being changed. This bug is shown in Fig. 9c. When a user initiates the login, PayPal SDK accesses to an endpoint with a basic access authentication header using the merchant ID in base64 encoding [11]. The attacker substitutes the merchant ID with another merchant ID under his control. Once the user logs in, the PayPal server returns an OAuth bearer token [10]. This token binds the user to the changed merchant ID, such that the payment is also associated with the changed merchant. Later, once PayPal SDK accesses REST API endpoint to create payment with the token, the payment is paid to the attacker's merchant ID. We remark that unlike the first issue, the payee information is not displayed to the user in this case. Our investigation finds that the payment details returned from PayPal's server to the SDK does not include who the payee (merchant) is. The payee information only appears on the last message from PayPal server, i.e., after the payment is completed.

The above bug can lead to the following scenario. A network attacker might change the parameters when user is making a transaction with merchant. If the user does not check who she is paying to, she might pay to a wrong merchant.

Visa Checkout SDK. We also have done the same testbed building and security analysis on Visa Checkout SDK. We have found that the Visa Checkout SDK follows a very strict step-by-step process. It also does not incorporate as rich functionalities as PayPal, such as the future payment and profile sharing. Therefore, no problem is found from the Visa checkout SDK, and the security problems we have found in PayPal do not exists in Visa Checkout SDK.

7.2 Correct Usage Summarization

In addition to the rule shown in the motivating example, we summarize rule #2 and rule #3 from the three bugs in Fig. 9 introduced in Sect. 7.1.

#2. *A merchant App* ***should not*** *assume the payment information returned from PayPal SDK is correct and complete.*

As stated in the first bug (Fig. 9a) and the second bug (Fig. 9b), after receiving payment details which may have been changed by the malicious merchant, PayPal SDK does not accordingly update the information displayed to the user.

#3. *For every payment, the merchant server* ***must*** *verify the payment information (****including payer ID, payee ID, amount, currency and freshness of payment****) and the status of the transaction to ensure the correctness of the payment.*

This rule applies for both PayPal and Visa. The messages sent to the merchant server from the merchant App may have been tampered by the malicious users. Therefore, the merchant server should not trust these messages. Instead, it should use the Payment ID received from the merchant client to make a direct API call to the PayPal (or Visa) server to retrieve the detailed payment information.

In particular, the merchant should verify the correctness of payer ID, payee ID, amount, currency and freshness of the payment. In addition, the merchant should not deliver any service or items to the users before the payment is verified. Moreover, since the messages out of the device can be changed by the malicious users, the verification of payment must be performed from merchant server as specified in rule #1.

8 Ambiguity in Documents

In this section, we report the ambiguities between the interpretation of the document and the facts we get from the system implementation.

i. *Future Payment allows the merchant to capture 15% more than the amount in the payment authorized by the user. This* ***should*** *be explicitly displayed to users when the users check out with PayPal.*

This ambiguity is observed from a test case with malicious merchant attacker model where the malicious merchant successfully changes the amount to a larger number. Using manual testing, we identify this upper bound of the extra amount (15%) which the merchant can capture. This is not a bug, since we later find this policy in one of PayPal's documents named *Authorization* & *Capture* [4] which is burred deeply among other documents. It states that the merchant can charge user 15% more with an upper bound of $75. However, since there is no such statement in the document of PayPal Android SDK, this may cause confusion in the responsibility between the merchant App developers and PayPal regarding who should be the party warning user of this policy. An App developer often only focuses on the functionality implementation of the App, but tends to overlook the policy issues. Thus, it is likely that the developers only read the SDK documents, such that they may never notice the policy and let alone to inform the users.

ii. *When a user has authorized the merchant for profile sharing, the merchant becomes able to charge the user through the future payment, even though the user has never authorized the future payment before.*

This is observed in a test case under malicious user attacker model when the malicious user replaces the future payment authorization code with another authorization code he has obtained for profile sharing. To examine the cause, we surprisingly find that the scope of profile sharing includes the permission of future payment. This implies that the merchant can wrap the request for permission of future payment into a request of profile sharing, such that an incautious user who intends to authorize the profile sharing may actually authorize the future payment. In addition, this security-sensitive information on relation of the profile sharing and future payment is not stated clearly in the document of PayPal Android SDK.

iii. *When a user has previously authorized the merchant with future payment, the authorization code of profile sharing to the same merchant from the same user automatically enables the merchant to make future payment without user's authorization, even if the merchant App does not request the future payment access in the profile sharing.*

This is observed in the test case under malicious merchant where he changes the future payment code with profile sharing code that a same user has previously authorized. Contrary to ambiguity **ii.**, if the user has previously authorized the merchant to make future payment, and later the user also consents the merchant to do profile sharing without future payment access in the scope. The authorization code for profile sharing can be used to cerate and capture a payment.

iv. *Although multiple steps are stated necessary by the documents, order payment in single payment can be captured directly without the following steps.*
   – *executing the order,*
   – *and authorizing the order.*

This ambiguity is discovered in a test case under malicious merchant attacker model where the malicious merchant skips the above mentioned steps and captures the payment directly, The PayPal SDK document does not detail the order payment, but only provides a link to a REST API document [1]. In that document, an order payment has to take five steps to complete, starting from the initial step "*Create the order*", then "*Get customer (user) approval*", "*Execute the order*", "*Authorize and order*" and lastly to "*Capture an order*". However, based on our testing results, the order payment created from the PayPal SDK can be captured directly without the "*Execute*" and "*Authorize*" steps.

v. *Client Metadata ID is not a necessary information for mobile payment protocol of future payment.*

This ambiguity is discovered during the trace refinement. After the client metadata ID is removed from the request sending by SDK to the TPC server, the response from the TPC server does not change. This implies that the client metadata ID is not a necessary information at all. This is contradictory to PayPal SDK's document [2] which clearly states that Client Metadata ID is necessary.

9 Related Work

Our work is related to the following two areas – third party library analysis and flaws detection from integrated applications. In this section, we brief related work in these two areas.

Third party library. In [19], the authors conduct security analysis on the China's mobile payment market. They find security vulnerabilities in different payment libraries and suggest security rules for developers. Different from it, this work aims to use a systematic approach to identify hidden assumptions and ambiguities. In [18], the authors aim to uncover the hidden assumptions for using the SDKs in secure authentication and authorization. In [15], the authors leverage black-box testing with known attack patterns to test the security of multi-party web applications.

Flaws Detection. The other type of related research is detecting flaws in application implementations. In [6], the authors develop a tool to automatically extract and translate the protocol into a formal model. Then vulnerabilities of the protocol can be identified by formally analyzing the extracted model. Similar to this work, a Single Sign-on (SSO) protocol is extracted from network traffic and formally modeled. Through this, security vulnerabilities are identified through formally verifying the formal models [20]. While in Pellegrino et al.'s work [14], the authors use black-box testing to test web applications, aiming at finding logic flaws. [16] uses a static analysis to identify the vulnerabilities in e-commerce web applications. Prior to this, [17] studies Cashier-as-a-Service based web stores and finds that integration of the third-party services might introduce vulnerabilities into the web applications.

10 Conclusion

We propose a systematical approach to identify correct usage and hidden assumptions in mobile payment protocols that developers should be aware of. These identified usage and assumptions urge both the protocol designers and the TPC SDK developers to provide clearer and well-formed documents. More techniques [5] should be used to check, and if possible, to formally verify the security of the payment protocol implementation.

Acknowledgement. We thank all the anonymous reviewers and our shepherd Dr. Xiao Zhang for their invaluable comments and guidance in revising this paper. This research is supported (in part) by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

References

1. Create and process orders (2016). https://developer.paypal.com/webapps/developer/docs/integration/direct/create-process-order/. Accessed Aug 2016
2. Future payments mobile integration (2016). https://github.com/paypal/PayPal-Android-SDK/blob/master/docs/future_payments_mobile.md. Accessed Aug 2016
3. Paypal sandbox testing guide (2016). https://developer.paypal.com/docs/classic/lifecycle/ug_sandbox/. Accessed Aug 2016

4. Authorization and Capture (2016). https://developer.paypal.com/docs/classic/admin/auth-capture/. Accessed Aug 2016
5. Bai, G., Ye, Q., Wu, Y., Merwe, H., Sun, J., Liu, Y., Dong, J.S., Visser, W.: Towards model checking android applications. IEEE Trans. Software Eng. **PP**, 1 (2017)
6. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: Authscan: automatic extraction of web authentication protocols from implementations. In: 20th Annual Network and Distributed System Security Symposium (NDSS) (2013)
7. Bai, G., Sun, J., Wu, J., Ye, Q., Li, L., Dong, J.S., Guo, S.: All your sessions are belong to us: investigating authenticator leakage through backup channels on android. In: 20th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 60–69. IEEE (2015)
8. ML Communication: Proxydroid (2017). https://play.google.com/store/apps/details?id=org.proxydroid&hl=en. Accessed 7 Aug 2017
9. Denale, R.: U.S. census bureau news-quarterly retail e-commerce sales, 17 May 2016. https://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf. Accessed Aug 2016
10. Jones, M., Hardt, D.: The OAuth 2.0 authorization framework: Bearer token usage. Technical report (2012)
11. Josefsson, S.: The base16, base32, and base64 data encodings (2006)
12. Meola, A.: The rise of m-commerce: mobile shopping stats and trends, December 2016
13. Oberheide, J., Jahanian, F.: When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In: Proceedings of the Eleventh Workshop on Mobile Computing Systems and Applications, pp. 43–48. ACM (2010)
14. Pellegrino, G., Balzarotti, D.: Toward black-box detection of logic flaws in web applications. In: 21st Annual Network and Distributed System Security Symposium (NDSS) (2014)
15. Sudhodanan, A., Armando, A., Carbone, R., Compagna, L.: Attack patterns for black-box security testing of multi-party web applications. In: 23rd Annual Network and Distributed System Security Symposium (NDSS) (2016)
16. Sun, F., Xu, L., Su, Z.: Detecting logic vulnerabilities in e-commerce applications. In: 21st Annual Network and Distributed System Security Symposium (NDSS) (2014)
17. Wang, R., Chen, S., Wang, X., Qadeer, S.: How to shop for free online-security analysis of cashier-as-a-service based web stores. In: IEEE Symposium on Security and Privacy, pp. 465–480. IEEE (2011)
18. Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., Gurevich, Y.: Explicating SDKs: uncovering assumptions underlying secure authentication and authorization. In: Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 13), pp. 399–314 (2013)
19. Yang, W., Zhang, Y., Li, J., Liu, H., Wang, Q., Zhang, Y., Gu, D.: Show me the money! Finding flawed implementations of third-party in-app payment in android apps (2017)
20. Ye, Q., Bai, G., Wang, K., Dong, J.S.: Formal analysis of a single sign-on protocol implementation for android. In: 20th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 90–99. IEEE (2015)