Messy States of Wiring: Vulnerabilities in Emerging Personal Payment Systems

Jiadong Lou, Xu Yuan *
University of Louisiana at Lafayette

Ning Zhang
Washington University in St. Louis

Abstract

This paper presents our study on an emerging paradigm of payment service that allows individual merchants to leverage the personal transfer service in third-party platforms to support commercial transactions. This is made possible by leveraging an additional order management system, collectively named *Personal Payment System (PPS)*. To gain a better understanding of these emerging systems, we conducted a systematic study on 35 *PPS*s covering over 11740 merchant clients supporting more than 20 million customers. By examining the documentation, available source codes, and demos, we extracted a common abstracted model for *PPS* and discovered seven categories of vulnerabilities in the existing personal payment protocol design and system implementation. It is alarming that all *PPS*s under study have at least one vulnerability. To further dissect these potential weaknesses, we present the corresponding attack methods to exploit the discovered vulnerabilities. To validate our proposed attacks, we conducted four successful real attacks to illustrate the severe consequences. We have responsibly disclosed the newly discovered vulnerabilities, with some patched after our reporting.

1 Introduction

Pervasive network connections in modern computing devices are enabling the adoption of the online payment service as a more convenient and safer method for monetary transactions. Reports from eMarketer [20, 21] state that 1.06 billion people are using a proximity mobile payment. Asia's 577.4 million proximity mobile payment users make up about half of that total, largely due to rapid adoption in China. In recognition of the growing market, there have been an increasing number of mobile payment platforms designed to enable payments among users without having to go through traditional methods such as credit cards and checks, often reducing the risk to the user in the case of information disclosure to the vendor. The well-known third-party platforms include but are not limited to Alipay [2], Wexpay [10], Apple pay [4], Paypal [6], and Venmo [8].

However, existing payment platforms have several limitations. First, individual payment accounts on these platforms are not designed to handle large volumes of transactions, thereby there is a lack of scalable methods to automatically associate orders with payment transactions. Second, while there exist merchant accounts that can be registered on the payment platforms to provide the aforementioned functionalities, the barrier to entry is quite high for many small businesses. For example, it requires a government-issued license (considerable delay in application) in China to get a merchant account with Alipay. Lastly, there is also a non-trivial upfront cost commitment to get started. Realizing these drawbacks, a new form of payment management service has emerged to serve as a broker between buyers and sellers, providing a minimalist payment management system with significantly lower transaction fees and initial financial commitment. They, however, are not individual financial institutions and do not offer wallet functions. Instead, these payment management systems rely on existing personal transfer services from third-party platforms for actual money transfers. We refer to this new paradigm of payment system as the personal payment system (*PPS*).

Since this new payment system builds on top of the existing personal money transfer interface and has to involve multiple rounds of complex interactions between different entities in the ecosystem, security remains a challenging issue for both the *PPS* and its users. Recognizing the role of payment systems in the digital economy, there have been several studies analyzing and demonstrating security issues in both web-based payment services [18, 34, 45, 46, 48, 52] and in-app payment services [25, 37, 54]. However, they have mainly focused on commercial payment services, and little attention has been given to the emerging *PPS*.

In this paper, we present our systematic analysis of the emerging *PPS* where we dissect its design elements, ecosystem, and potential vulnerabilities. The 35 *PPS*s under study

*Corresponding author: Dr. Xu Yuan (xu.yuan@louisiana.edu)

offer both web and mobile app applications covering over 11740 merchant clients supporting more than 20 million customers. By analyzing the technical documents and applications of these 35 *PPS*s, we abstract the common design pattern of their payment systems and the corresponding business processes. We found that all *PPS*s have five key components to satisfy the essential business needs of their clients, namely *PPS* enrollment and key distribution, order generation, order payment, payment notification, and order inquiry. We found seven unique patterns of vulnerabilities that are common in the majority of the studied *PPS*s. Of the five key components, the majority of the vulnerabilities are within the order generation process. Based on the discovered vulnerabilities, we designed and implemented five proof-of-concept attacks that chain together multiple vulnerable patterns to demonstrate the real-world threat. To minimize the impact on the real-world systems, all experiments were designed to attack our own test accounts, and we also reported the processes and results to all vendors so that they could mitigate the impact of the attacks. The financial ramifications of the attacks are typically mitigated by closure of the test accounts.

To mitigate the threat of these vulnerabilities, we conducted a systematic analysis of the root causes and classified them from different perspectives, such as protocol vs. implementation. With the analysis, we list 10 suggestions for *PPS* providers, merchants, and the buyers who make payments.

Lastly, to ensure that the vendors had enough time to fix the vulnerabilities, we contacted them individually about the vulnerabilities several months before the submission of this manuscript. This allowed several vendors to finish patching the reported vulnerabilities at the time of writing. We have also disclosed the vulnerabilities to various security response platforms, including Tencent security response center [7] and Alibaba security response center [1].

In summary, we have made the following contributions in this paper:

- We dissected the internals of a newly emerging paradigm of payment system, the Personal Payment System (*PPS*), and presented a common abstracted model of these new payment systems.
- Based on our analysis of the *PPS*, we have discovered 7 vulnerable patterns and presented 5 new attacks methods that exploit these vulnerable patterns.
- We conducted an empirical study to analyze the payment services from the 35 most widely used *PPS*s and exposed the security issues corresponding to our discovered 7 vulnerable patterns. Four real-world attacks on the websites adopting the *PPS*s were also conducted to demonstrate our discovered vulnerabilities.
- Following the practice of responsible disclosure, we have reported all the discovered design flaws and worked with some vendors in fixing these vulnerabilities.
- To mitigate these vulnerabilities, we have conducted a root cause analysis and provided a discussion on how to secure the ecosystem.

2 Personal Payment System

2.1 PPS Definition

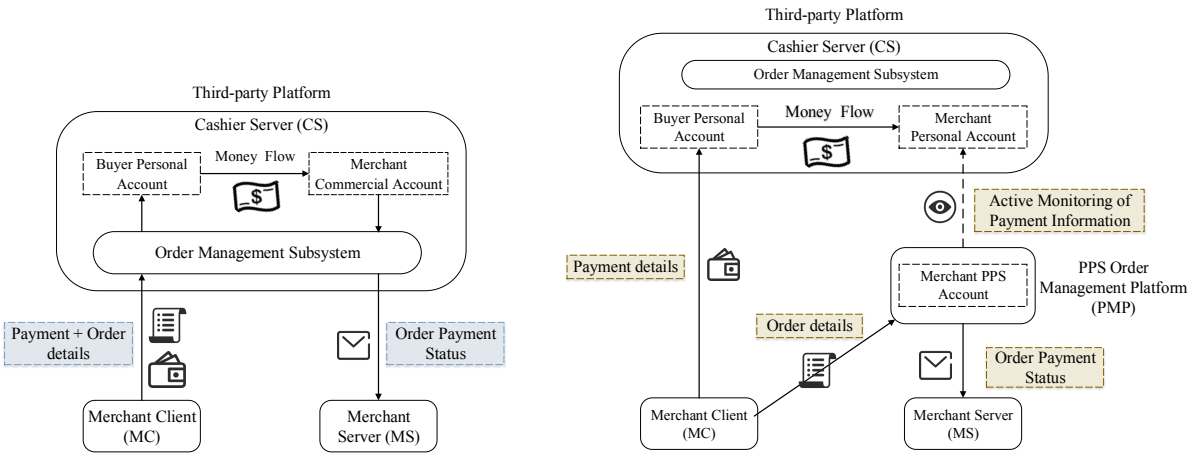
Personal payment system is an emerging paradigm that couples personal money transfer functions provided by existing third-party platforms with an order management platform developed by a *PPS* provider. It allows individual small businesses to leverage personal financial accounts registered in third-party platforms to meet the demands of commercial transactions without incurring significant upfront costs. Due to its low barrier of entry, these new systems have successfully attracted a large number of merchants, especially startups and small businesses.

Third-party payment platforms provide free personal money transfer services, and each user can register a personal account for sending money to or receiving money from others. However, the personal transfer interface does not include payment and order related management functionalities (e.g., recording the payment information and order status, monitoring the money flow, informing merchants about payment status). Therefore, it is often impossible to use these accounts directly for commercial transactions. Recognizing this opportunity, *PPS* providers develop a payment management platform to complement existing payment platforms. To use the *PPS* platform, a merchant has to register an account on the *PPS* system and link this account to an existing personal account on a third-party payment system such as AliPay. The *PPS* platform will work in the background to present to its users a unified commercial-capable interface that supports both payment and order tracking.

2.2 Personal Payment System v.s. Commercial Payment System

While both the existing mainstream third-party commercial payment systems (*TP-CPS*) and the *PPS* offer the ability to manage transactions and payments at the commercial scale, there are major differences between the ecosystems of the two. The high level workflows of *TP-CPS* and *PPS* are summarized in Figures 1(a) and 1(b), respectively.

In the *TP-CPS*, as shown in Figure 1(a), there are three entities, i.e., merchant client (*MC*), merchant server (*MS*), and cashier server (*CS*). *MC* is the merchant client, where users can browse merchandise and make orders. *MS* is the merchant server that hosts the client content. It is also responsible for processing orders and confirming payment statuses of orders. The *CS* in a third-party platform manages money transactions between different accounts, and offers the ability to track payment status. With a commercial account on a third-party



(a) The transaction process in the TP-CPS.

(b) The transaction process in the PPS.

Figure 1: The transaction process in two payment paradigms

payment platform, order management is part of the service and is tightly integrated. As a result, *MC* and *MS* only need to interact with third-party systems to upload and obtain both the payment and order status for an agreed upon fee.

On the other hand, for *PPS*, shown in Figure 1(b), a merchant makes use of an independent order management system. He/she also needs to register an account with the *CS*; however, the account is a personal account without the ability to access the integrated order management system, and therefore there is no commission fee imposed on the transactions. To leverage the personal payment system for commercial transactions, *PPS* provides a *PPS Order Management Platform* (*PMP*), an additional platform offering the commercial payment functionalities traditionally provided by *CS*, such as recording the transaction information, monitoring cash flow in the transaction, and verifying the order status.

The transaction flow in *PPS* can be summarized as follows: A merchant needs to register two accounts, one personal account at the third-party payment platform (i.e., *CS*) and a merchant commercial account at the *PMP*. After the buyer places an order, the payment information will be sent to the *CS* while the order information is transmitted to the *PMP*. When the contracted payment is made in the *CS*, the *PMP* will then notify the *MS* to continue the order process. The addition of an independent *PMP* in the order and payment management process is the unique design element in this ecosystem that avoids the transaction fee. To obtain payment status, the merchant account at *PMP* has to keep monitoring for the money transfer event at the *CS*.

2.3 PPS Abstracted Model

The abstracted model common to all *PPS*s is described in this subsection, from the initial enrollment of the service to order processing as well as order query.

2.3.1 Enrollment and Key Distribution/Update

Enrollment and initial key distribution - In order to use *PPS*, a merchant needs to register for a personal account on the third-party cashier server as well as a commercial account on the payment management system in the *PPS*. After completion of the registration process of the merchant commercial account on the *PPS* website, the enrollment process starts with the initial key distribution. First, a unique key (henceforth referred to as **KEY**) is generated for each merchant account. The **KEY** is distributed via the merchant account web page on the *PPS* platform and can only be viewed via this web portal. The initial **KEY** distribution process is protected by common web-based security techniques using merchant account authentication (i.e., logging in) at the *PPS* website.

Key update - While the initial key is only available via the *PPS* website, it is also possible to request key renewals subsequently and receive the updated keys via web APIs. As a result, there are two ways *PPS* merchants can request and receive key updates, via the *PPS* website or via a REST API call. All the *PPS*s we studied support key update via API. Furthermore, 88% of them do not require authentication. More details can be found in section 3.1.

Common interface and order format - Almost all the *PPS*s offer API-based payment service in some form. Specifics about the order and the transacting parties are usually encapsulated in a JSON object sent through the order API (*O-API*). Required fields often include order identifier, payment URL, price, merchant ID as well as the signature that aims to protect the integrity of the object. This signature is also widely referred to as the **Token** in many of the user documents. Surprisingly, we found all *PPS*s use the MD5 mechanism to generate the **Token** (i.e., single MD5 or multi-layer MD5).

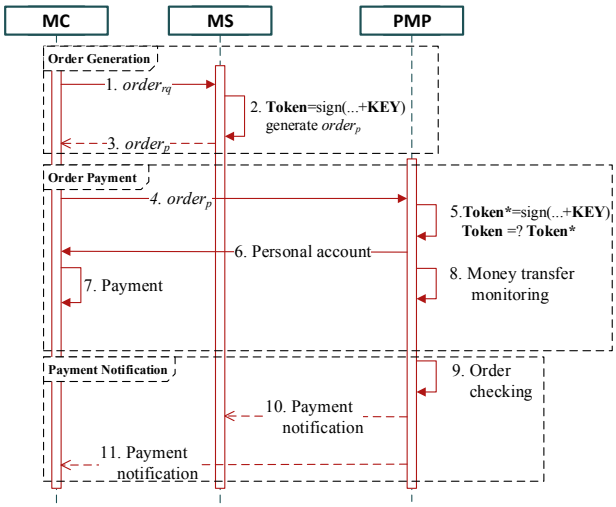


Figure 2: The transaction flow of *PPS*.

The input of MD5 is a string that concatenates all the related order information (i.e., values in all fields) in an order packet as well as the **KEY**. The ordering of all fields varies in different *PPS*s. The most common approach is to order the fields alphabetically.

2.3.2 Order Generation

When the user makes a purchase attempt, an order is generated at *MS* and delivered to *MC*, as shown in Figure 2.

1. In the first step, *MC* generates an order request, denoted as $order_{rq}$, and then sends it to *MS*. The $order_{rq}$ often contains only the minimal amount of information that is necessary to associate the order with the merchandise, such as the unique item identifier. Since the merchant may offer multiple payment options, the buyer's choice is also included in this request.

2. After receiving the $order_{rq}$, *MS* generates the order packet $order_p$, which contains detailed order parameters for the requested merchandise, and a **Token** is also generated using **KEY** to protect the integrity.

3. *MS* delivers $order_p$ back to *MC*.

2.3.3 Order Payment

As shown in Figure 2, this stage includes five steps as follows.

4. *MC* sends the order packet $order_p$ to *PMP*.

5. Upon receiving the order packet $order_p$, the *PMP* will look up the stored **KEY** for the merchant specified in the $order_p$, then use the received $order_p$ and **KEY** to verify the integrity of the order packet. Note that when *PMP* receives multiple orders from one merchant with the same price simultaneously, it might change the payment amount by a tiny deviation on these orders so that *PMP* can identify these orders by monitoring the paid amount.

6. The payment account information (i.e., the QR code of the merchant's personal account registered at the chosen third-party platform) is sent to the *MC*.

7. The buyer pays the required amount to the merchant's personal account.

8. *PMP* monitors the merchant's personal account for the expected money flow. The actual implementation of how the *PMP* can monitor the personal account of the merchant differs among the *PPS*s under our study. For example, some of the *PPS*s monitor the financial transactions on the personal account of the merchant by installing a client app on the merchant's smartphone that will hook into the notification interfaces of the third-party payment platform apps. Even though merchants need to specifically provide consent to such monitoring in order to use the *PPS*, there are significant privacy and security concerns with such designs. However, we will leave the investigation of these apps for a another time.

2.3.4 Payment Notification

In this stage, *PMP* sends notification to both *MS* and *MC* after confirming the payment. Three steps are included as follows:

9. Once the *PMP* detects the money paid to the merchant's personal account, *PMP* compares it to the expected paid value of the pending orders.

10. A notification is then sent to the *MS* (via $notify\_url$ in $order_p$) which includes the payment status code, order ID, and/or monitored payment value, indicating whether an order is successfully paid or not.

11. The same notification is also sent to the *MC* via the $return\_url$ in $order_p$.

Notably, not all *PPS*s include the paid value in the notification to *MS* in step 10. In fact, few *PPS*s include the price in the notification. As mentioned in step 5, there can be small deviations between the actual paid value and the item price. This small deviation is a mechanism by which *PMP* distinguishes different transactions from the same merchant by making small adjustments to the price such that price is unique in each transaction. Furthermore, *MS* is not informed of this deviation in advance, therefore, they would be not able to associate a money transaction with the order number. As a result, the value of the price in the notification is often not used, and only the order number and the payment status code in the notification are used by *MS* to verify the success of the

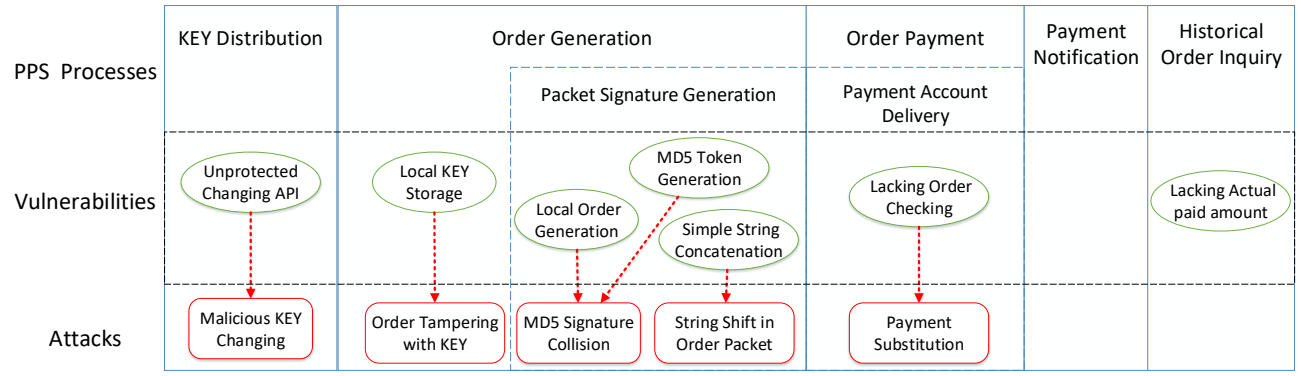


Figure 3: The Vulnerabilities and attacks in the *PPS* flows.

payment on a particular order in *PPS*. This unique design, which ignores consistency in the order price of different order processing stages, turns out to be problematic from a security perspective, which will be discussed later in section 3.

After all the steps above, the payment transaction of one order is completed as shown in Fig. 2, and the merchandise can be shipped.

2.3.5 Order Inquiry in PPS

In *PPS*, the order management system is provided by *PMP* to support the historical order inquiry service.

1. *MS* generates the query request (denoted as $query_o$), which contains the merchant identification code (*merchantID*), and the order ID (*orderid*). This request is protected by the $\mathbf{Token}_q$.
2. *MS* sends this query packet $query_o$ to *PMP*.
3. Upon receiving the $query_o$, *PMP* looks up the merchant **KEY** based on the merchant ID in the $query_o$. Using the **KEY**, *PMP* verifies the $\mathbf{Token}_q^*$ in $query_o$.
4 Once the request is verified, *PMP* sends the inquiry result $query_{res}$ back to *MS*.

We also found that only the payment status code is included in the $query_{res}$ rather than the actual paid value in most of the *PPS*s. In this case, the merchant has no way to look up the actual payment value in historical orders. This creates an opportunity for the attacker, since merchants would not be able to go back and verify the payment value via the *PMP*.

3 Security Analysis

In this paper, we focus our analysis on the unique design of *PPS*, involving three key parties *MC*, *MS*, and *PMP*. As shown in Figure 3, there are five main stages as previously described, from key distribution to order generation, order payment, payment notification, and historical order inquiry. We found seven unique vulnerable patterns in multiple stages of the order processing pipeline. Based on these vulnerabilities, we have created five proof-of-concept attacks.

3.1 Vulnerable KEY Distribution/Update

As discussed in Section 2.3.1, the unique **KEY** is assigned when a merchant subscribes to the services and is often displayed on the *PMP* management web page. While accessing the key via web management interface is well protected with communication security mechanisms such as HTTPS, *PPS* also provides REST APIs to allow merchants to manipulate keys programmatically.

3.1.1 Unprotected Key Changing API

Through our study, we found that the web management interface is well protected, however, the API allows pre-authenticated requests to change **KEY**. In most of the *PPS*s we studied, the API only requires merchant ID to change **KEY**. The merchant ID is not a secret by design and can be directly obtained by examining the order packet. As a result, an adversary, who has the merchant ID, can easily forge a **KEY** change request from *MS*. This can lead to loss of merchant **KEY** or disruption of merchant's e-commerce.

3.1.2 Attack: Malicious KEY Changing

Taking advantage of this vulnerability, an attacker can first obtain the merchant ID from various places including any order packet sent from *MC* to *PMP*, then forge a key change request.

In some *PPS*s, the new **KEY** is included directly in the reply. Using the new **KEY**, the attacker can make arbitrary modifications to the order requests from this *MS*. Unfortunately, communication security mechanisms such as HTTPS with TLS do not mitigate this attack since attacker is the party

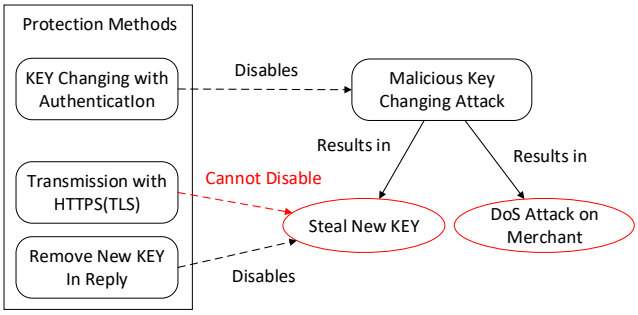


Figure 4: The attack, consequences, and protection in **KEY** distribution.

making the requests, instead of eavesdropping or launching man-in-the-middle attack over the network.

In other *PPS*s, the new **KEY** is not included in the reply, and can only be accessed via the standard web portal. The best an attacker can do is to leverage the interface to make frequent key changes to disrupt merchant operations and achieve DoS.

The relationship between different attack results and protections against malicious use of **KEY** change APIs can be found in Figure 4. It is possible to fix the root cause of this vulnerability by mandating authentication on key change request. It is also possible to limit the damage of a pre-authenticated key change API by removing the new **KEY** from the reply. However, communication security mechanisms, such as HTTPS, unfortunately cannot defend against key stealing attacks or DoS since the attacker is the one making the API calls.

3.2 Vulnerable Order Generation

Theoretically, orders should be generated and signed at the merchant server end before being delivered to the client. However, in practice, many implementations generate partial or entire orders at the client. In other words, the order request includes information fields that go directly into the final order packet.

3.2.1 Local Order Generation

When the orders are generated locally, there are two security implications. First, it implies that attackers can tamper with some fields of a locally stored order. In many cases, *MS* does not conduct additional cross validation, and then the attacker can successfully manipulate an order. For example, the attacker may reduce the price of an item. The second implication is related to information leakage. When the server performs additional validation, the attacker cannot simply reduce the price by modifying the order request, but he/she can still leverage the leaked information to help stage more advanced attacks. For example, in the case of a hash collision attack, which is described later in the section, it is important that the attacker can manipulate fields in the order to accommodate the spaces needed for near-collision blocks.

3.2.2 Local KEY Storage

The security of *PPS* mainly relies on the signature mechanism to verify order packets, where **KEY** is essential as discussed previously. However, to enable the local order generation, some *MC* implementations store **KEY** in *MC* for convenience. This allows an attacker to easily obtain the merchant's **KEY** by reverse engineering the *MC* program. With the **KEY**, an attacker can make arbitrary modifications to the order packet $order_p$.

3.2.3 Attack: Order Tampering with KEY

When the key is stored locally in *MC*, the attacker can easily extract it by reverse engineering, and is able to make arbitrary modifications to the order packet. After initiating the order process with *MS*, the compromised *MC* can modify the order price and generate a new order packet $order'_p$ with the modified price using the stored **KEY**. Furthermore, almost all *MS*s check the payment status code instead of the actual paid value due to the unique arrangement of price adjustment in *PPS*. Thus, they cannot notice such a price change. To make it worse, there is actually no way for the *MS* to find the historical payment value from *PMP* either, which we will discuss at the end of this section.

3.3 Vulnerable Packet Signature Generation

Packet signing with the **KEY** to generate a **Token** is the most important design aspect of the payment protocol in regards to preventing order tampering. Our analysis revealed two vulnerabilities.

3.3.1 String Concatenation in Token Generation

The first vulnerability falls in the process of string concatenation when generating a **Token**. As we have described in Section 2.3.1, **Token** is generated by concatenating all items (i.e., parameters and their values) in the order packet along with the **KEY** into a string and inputting that string to the MD5 algorithm. While there are *PPS*s that separate the fields in the order with a special character, some *PPS*s simply concatenate all fields without any delimiter. As a result, when the suffix of one field is shifted to the prefix of the next field in the order packet, the generated **Token** does not change. This allows an attacker to tamper with the order packet by shifting some characters from one field to the next field.

3.3.2 Attack: Order Tampering using String Shift

To exploit this vulnerability, the attacker needs to take advantage of several unique designs that are uniform to *PPS*. Since

PMP does not have the capability to store all the detailed information (including pricing) for individual merchandise from all the merchants, *PMP* has to rely on the received order packet to obtain the merchandise price and certify whether this price has been modified via the token. Additionally, since *MS* lacks visibility in how *PMP* manipulates prices to multiplex orders, it needs to rely on the payment status flag in the payment notification packet rather than the paid amount. As a result, since neither *MS* nor *PMP* know what the correct payment amount is, if the adversary can forge an order that passes verification on the token, then he/she can purchase any item at a much lower price. While all the fields are well defined using a JSON object, the signature verification is only over the string concatenation of values from the consecutive fields. This implies that the trailing bits of a field can be maliciously shifted to the heading bits of the next field without impacting the signature.

To give an illustrative example, let's consider a price modification attack. To launch the attack, the attacker can shift some suffix of a *Price* field into the neighboring *return_url* field or optional fields. Figure 5 shows one example of this operation, where we assume the *Price* field is 100 and the neighboring *return_url* is "www.xxx.com". In this attack, the attacker modifies the order packet $order_p$ at *Step 3* by moving the last *0* in *Price* to *return_url* before sending it to *PMP*. That is, one 0 in the *Price* field is shifted to the front of the *URL* field. Then the modified packet will include the new price of 10, and the *URL* of 0www.xxx.com. *return_url* works as the function of notifying *MC* of the payment status.

Figure 5: An example of the string shift in the order packet.

Since the string concatenation of the request has not changed, the **Token** remains the same and can pass the verification at *PMP*. As a result, the attacker only has to pay a tenth of the price to purchase the item. However, because of the modification on the *return_url*, his merchant client app will not receive the notification, but the loss of this functionality is not important for *MC*.

3.3.3 MD5-based Token Generation

The second vulnerability is from the use of weak hash cryptographic primitives. From our empirical study, most of the *PPS*s are using weak cryptographic primitives such as MD5 message-digest algorithms to generate the **Token**. There have been extensive studies demonstrating the weakness of MD5 [14, 19, 24, 49]. MD5 collision attack has been verified and implemented in [49], in which two different inputs that have the same prefix can generate the same output string. Later, the chosen-prefix collision attack proposed in [41–44] allows an attacker to change the prefix part of one input, but still generate the same MD5 output. Consequently, an attacker can maliciously modify some of the fields in the order packet (e.g., price), but still generate the same **Token** for fooling the *PMP*, even without the **KEY**. However, exploiting this requires chaining several vulnerabilities together.

3.3.4 Attack: Order Modification based on MD5 Collision

Attackers have to take advantage of the vulnerabilities from local order generation, in which some parameters are generated at *MC* instead of *MS*, so that before the **Token** is generated at the *MS*, they can create the collision based on the parameters leaked at the *MC*. Different from the string shifting attack, by leveraging collision attacks in the cryptographic hash function attackers can make significant changes to the order. With this attack it is possible to modify the values of the fields instead of shifting bits from one field to its neighbor. Implementing the MD5 signature collision in *PPS* should follow these three steps:

Parameter Acquisition. To create the MD5 collision, the attacker needs to obtain enough parameters from certain fields in the order packet. Before the order packet has been signed, the attacker can generate two packs of orders that differ in the expected field, usually the price field, but result in the same calculated MD5 value. The existing algorithm creates the MD5 collision by generating the two different data blocks after the prefix, and as a result, an optional field behind the price is necessary for placing the collision data blocks. Furthermore, for MD5, if *String1* and *String2* collide, then appending the same string before or after *String1* and *String2* would also cause a collision. The attacker only has to obtain the parameters between the price and an optional field for creating a collision.

Orders have a time window in which the payment can be accepted. Since the time for calculating the MD5 collision is usually longer than this window, capturing the packet after the payment has been set up and then generating the collision is impractical. However, we can predict the necessary parameters to practically implement the MD5 attack. Based on our empirical studies, some *PPS*s put the price near the optional field, usually named *return_URL* or *orderuid*, in addition to some other possibilities. Then the attackers can create the collision based only on two different prices. Moreover, some parameters between the price and the optional field can be obtained from other orders, such as $notify\_url$, merchant ID, and the merchandise name. These parameters can be easily obtained by applying for the payment and capturing the network packet. The most difficult problem comes from the condition where the order ID lies in the area between the price and the optional field, since it varies in different orders. We

collected order IDs from some merchants and found that a timestamp with a random sequence is the most common form. For example, an order from *xddpay* platform contains the order ID "20201009053425901798", where "20201009053425" is the timestamp and "901798" is the random number. Based on the demo provided for the *PPS*, the random sequence is often created by the function "math.round(seed)". If the seed does not specially assign but only adopts the default value, which is the common situation in the sample code, the current time will be used as the seed when calling the function, and knowing that we can pre-calculate the random sequences to predict the order ID.

MD5 Collision Generation. The Chosen-Prefix attack method proposed in [43] can be employed by us to achieve such a goal. Given two different prefixes (denoted as *Prefix1* and *Prefix2*), two corresponding suffixes (denoted as *suffix1* and *suffix2*) can be constructed so that the concatenated values of *Prefix1*||*suffix1* and *Prefix2*||*suffix2* collide under MD5, with the time complexity of 2^{39}, where || denotes the concatenation of two strings.

The original price and the modified price are referred to as P_1 and P_2 respectively. Since the attacker is most likely trying to reduce the price he has to pay for an item, we assume P_2, the modified price, is a smaller value than P_1. Here we assume the optional field is a $return_{URL}$ (this field is usually designed as an optional field in our collected *PPS* protocols). One key property of URL is that any content after # will be ignored, therefore we can add # at the end of the URL and place the near collision blocks after the # so that we can even eliminate the influence of collision blocks in the order packet. Let S_1 and S_2 be the generated suffixes, which are the collision blocks. By using the Chosen-Prefix attack, the attacker can generate two strings, $P_1||URL\#||S_1$ and $P_2||URL\#||S_2$, that collide in MD5 algorithm, as shown in Figure 6. Furthermore, the parameters are delivered through JSON form, and some characters (i.e, " : {} []") are keywords. Any S_1 or S_2 that contains any of those characters can cause parsing problems. In the practical experiment, we will calculate multiple collision cases to avoid this situation.

Parameters Replacing. Leveraging the vulnerability that some order parameters are generated locally at the *MC*, the attacker first prepares two sets of parameters with the same MD5 value. The attacker then manipulates the *MC* to include $P_1||URL\#||S_1$ in the order request packet $order_{rq}$ to purchase the merchandise. Once the order packet is sent back to *MC* from *MS* after generating the MD5 **Token**, the attacker replaces the field of $P_1||URL\#||S_1$ with $P_2||URL\#||S_2$ while using the same token to generate a new packet $order_p^{'}$. The attacker then sends it to *PMP* to continue the transaction process. Figure 7 summarizes the procedure of this attack.

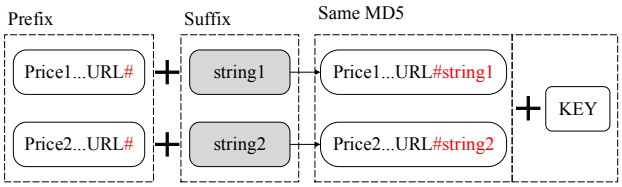


Figure 6: MD5 collision for two pairs of order parameters.

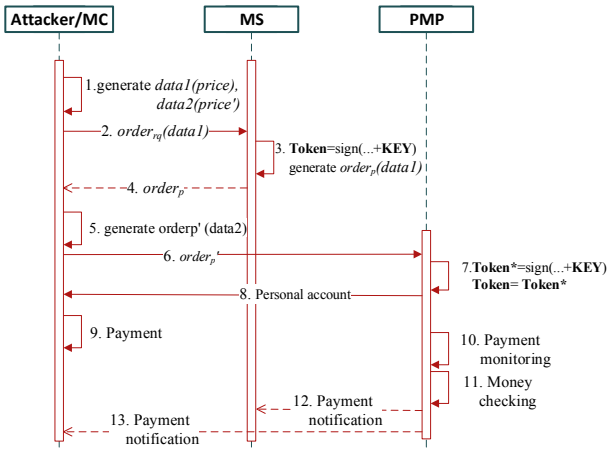


Figure 7: The flow of MD5 collision attack on *PPS*.

3.4 Vulnerable Payment Account Delivery

As shown in Figure 2, the merchant's personal account will be sent to the *MC* in QR code form after *PMP* has checked the packet signature as part of the payment process. The price will also be sent along with the QR code and displayed on the payment webpage to prompt the user. In general, if the displayed money amount is the same as the commodity price, the user at *MC* will trust this information and make the payment.

3.4.1 Lacking Order Checking Mechanisms

Most of the payment interfaces designed by the *PPS*s only display the price, order ID, and QR code to the *MC* so that a user can confirm and pay the bill. However, there is neither information on the merchandise for the order nor the shipping address. As a result, the order ID is the only clue a buyer can use to associate the payment with the item he/she is trying to purchase. However, since a buyer has no way to obtain his order ID, he will trust the one displayed by the payment interface. Even if the order has been substituted, the victim buyer doesn't notice the displayed order ID is not the one for his order. It leaves the opportunity for an attacker to swap a buyer's order payment information with his/her own order without the victim being aware .

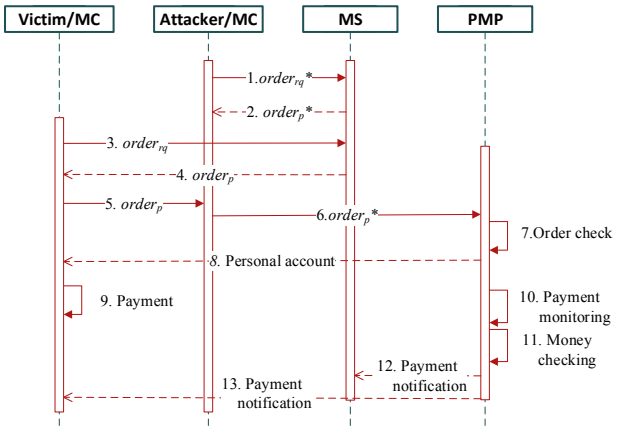


Figure 8: The flow of attack with payment substituting.

3.4.2 Attack: Payment Substitution

A payment substitution attack is shown in Figure 8. In this attack, the attacker first performs the man-in-the-middle attack to block the order, $order_p$, sent from the victim's *MC* to the *PMP*. Meanwhile, he obtains an order packet for his own order, $order_p^*$. Then the attacker substitutes the order information in the JSON field of $order_p$ with that in the $order_p^*$ and sends this tampered order packet $order_p$ to the *PMP*. Since this order request was a legitimate one, *PMP* will gladly accept the request and return the corresponding payment information to the victim's client based on the unchanged header information in $order_p$. The victim user, not knowing that his order has been swapped, will pay for the order but cannot get his purchased item, while the attacker gains his merchandise without paying.

3.5 Vulnerable Historical Order Inquiry

Generally, it is desirable to have an order inquiry system that can support queries on the details of previous transactions, such as *order id, payer id, goods' name,* and *payment amounts*. However, we found that most of the inquiry APIs in *PPS*s often return only a flag indicating whether an inquired order is paid or not. This limitation is also a key facilitator for our other attack in Sections 3.2.3, 3.3.2, and 3.3.4, since by the inquiry API a merchant can never recognize that an order's price has been tampered with.

4 Empirical Study

In this section, we discuss our empirical study to analyze the payment service from *PPS*s. Our goal is twofold. First, we investigate the usage of *PPS* and expose security issues existing in *PPS* by detecting the potential vulnerabilities as we have discussed in Section 3. Second, we use case studies to exhibit our attacks on some real-world payment applications supported by *PPS*s, demonstrating that the revealed vulnerabilities can cause serious consequences in real-world transactions.

Table 1: The list of collected 35 *PPS*s

PPS Names	Website
Paysapi	https://www.paysapi.com/
Xddpay	https://www.xddpay.com
Sdpay	https://www.sdpay.cc/doc/pay.html
020zf	https://www.020zf.com
Weimifu	http://weimifu.net/index.php
Pay10086	http://www.pay10086.com/docpay
Yktapi	http://weimifu.net/yktApi/index.php
Xunhupay	https://www.xunhupay.com
Paypayzhu	https://www.paypayzhu.com
Caiwumao	http://www.jiakeshuma.com
Userspay	http://pay.userspay.com
Greenyep	https://www.greenyep.com
Qianmapay	http://qianma.app/
Bearpay	http://www.bearpay.net
Xinyipay	http://www.bosee.cn/index.html
Zhifu	https://zf-api.com
BufPay	https://bufpay.com
ARYA	http://www.moont.cn
L pays	http://lp.edlm.cn/
Paycats	https://www.paycats.cn/
188PC	http://188pc.cn
PayJS	https://payjs.cn
Heimipay	https://www.heimipay.com/
Fastpay	http://www.weixin.mobi
Huanxipay	https://www.zhapay.com
Yijinka	http://www.yijinka.com/
PersonalPay	http://www.personalpay.cn
Shouxiaoqian	https://shouxiaoqian.com
XorPay	https://xorpay.com
7CPay	https://www.7cpo.com
Yuandianpay	https://www.suyoupay.cn
Dunpay	https://www.dunpay.net
Jupay	http://pay.jam00.comk
XPay	http://xpay.exrick.cn/
Sihupay	http://jia.bendilaosiji.com/

4.1 PPS Ecosystem

We collect *PPS* systems primarily via internet search and forum topics that match common keywords for *PPS*, such as "personal money collection", "security payment interface", "visa-free", among others. To this end, we found 35 *PPS*s, listed in Table 1. Their vulnerabilities are shown in Fig. 9, where the orange block represents that the *PPS* has the corresponding vulnerability and the red block indicates that the *PPS* has fixed the vulnerability based on our feedback. *PPS* names marked in red have temporarily stopped providing their payment services after our investigation.

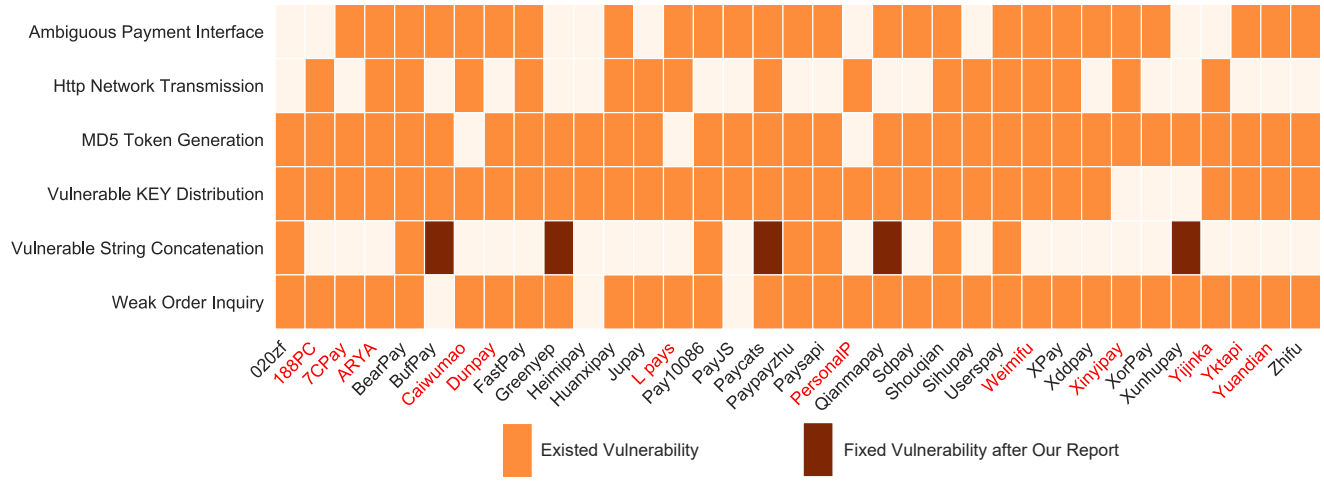


Figure 9: The vulnerabilities distributions in the collected 35 *PPS*s, where the horizontal axis lists the names of *PPS*s and the vertical axis lists the vulnerabilities that we have discussed in Section 3.

4.1.1 Discovering the Use of PPS in Merchant

The use of *O-API*, which is designed for the order interaction between *MC* and *PMP*, is a strong indicator for the use of *PPS* in the merchant system. Almost all of the *O-API*s are in the form of REST APIs at a URL. These URLs can be obtained by automatically parsing the user documents, which are then used to match against the source code of websites or applications. In the case of websites, they are directly visible, while parsing the mobile apps requires a basic reverse engineering tool such as Androguard [3].

4.1.2 Usage Statistics of PPSs

*PPS*s are currently used in both websites and mobile apps, with the web as the recommended method of deployment. In web-based deployment, *PPS*s are incorporated as plugins or website templates by the providers. Our study shows that *PPS*'s plugins have been downloaded more than 11,611 times according to the statistics of two popular repositories (i.e., Packagist and WordPress) and some data from *PPS* websites [11, 28–30, 50, 51], covering at least 10 thousand merchants and 20 million customers. Specifically, for one popular *PPS*, FastPay [11], 1,292 merchants and 129 corporations are using its service to implement the payment function in their products. The number of customers relying on this service could be in the millions. For example, we have found more than 10,000 customers that recharge the accounts and conduct purchase services on the website of an SEO-related merchant that adopts the Paysapi *PPS*. In terms of use of *PPS* in mobile apps, a total of 26,956 apps were crawled from SnapPea and Android Market app stores belonging to different categories. We found 564 apps that contain the string of unique *PPS* interface URLs. Through manual inspection, we found that 67 of them have employed *PPS* payment services, with average downloads of 2,000. These statistics evidence that *PPS* service has become an emerging payment paradigm, attracting a large number of merchants and individuals, and the adoption of *PPS* is still growing rapidly. They also justify the necessity and importance of our investigation.

Table 2: Signature Mechanisms of 35 *PPS*s

	Order packet	Order Inquiry Packet
One-layer MD5	30	31
Two-layer MD5	2	2
No signature	3	2

4.2 PPS Vulnerability Analysis

4.2.1 Vulnerable Signature Mechanism

To our surprise, we found three *PPS*s that do not adopt any signature mechanism to protect the order packets in transmission, i.e., no **Token** for integrity protection. The remaining 32 *PPS*s leverage a weak cryptographic primitive, MD5 specifically, for the token generation. 30 of them leverage the one-layer MD5 algorithm while 2 of them adopt the two-layer MD5 algorithm, all of which can be broken by MD5 collision attack (Section 3.3.4). For one-layer and two-layer MD5, the **Token** is generated by **Token**=*MD5(order parameters*+**KEY***)* and **Token**=*MD5(MD5(order parameters)*+**KEY***)*, respectively. In the order inquiry packet, 31 *PPS*s leverage the single layer MD5 algorithm, and 2 *PPS*s adopt the two-layer MD5 algorithm in the signature mechanism. The remaining two *PPS*s do not require the packet signature for order inquiry. Table 2 summaries the token generation mechanisms of 35 *PPS*s.

Table 3: Price field position in string concatenation strategies

	Separated Field	Direct connection
Near optional field	3	0
Near Notify URL	4	12
Near Required Fields	13	1

4.2.2 Vulnerable String Concatenation

To discover this vulnerability, our analysis is based on two criteria: 1) whether the values among different fields are separated with delimiters or the field names, and 2) whether the price field is nearby some optional field. Table 3 shows the concatenation methods of all 32 *PPS*s that adopt signature mechanisms. 13 *PPS*s have string concatenation without delimiters or the field name among neighboring fields, among which 12 *PPS*s have the optional field of *Return URL* near the price field. Here, the *Return URL* is the website address *MC* jumps to after the payment behavior. When the value in *Return URL* is modified, *MC* stagnates at the payment interface page, but the transaction is still successfully placed at the *MS*. As a result, these 12 *PPS*s are vulnerable to our proposed string shift attack (Section 3.3.2). The remaining 20 *PPS*s require the merchants to add a delimiter (i.e., &) between any two fields or include the corresponding field name before the values (e.g., "price=xxx") in the concatenated string. Our string shift attack does not work on these *PPS*s.

4.2.3 Vulnerable Key Changing API

We found that 31 out of 35 *PPS*s do not adopt signature mechanisms in the request packet and may be vulnerable to the malicious key changing attack described in Section 3.1.2. Additionally, among these 31 *PPS*s, 14 send new **KEY**s as cleartext, allowing an attacker to intercept the packet and obtain this new **KEY**.

4.2.4 Vulnerable Order Inquiry

All *PPS*s employed a cryptographic checksum, such as MD5, in the inquiry request packet to prevent malicious order inquiry. In the response packet, we found that all 35 *PPS*s attach the payment status, i.e., pending, error, and success, in the response packet, represented by a status code, while only 3 *PPS*s include both the expected amount and actual value paid. Without the expected and actual payment values, it becomes impossible for the merchant to perform audits afterward, significantly impacting financial operations should there be an attack.

4.2.5 Vulnerable Payment Interface

As discussed previously, the payment interface can be crucial in assisting the buyer to identify a payment swapping attack in which an attacker swaps in his order to mislead the victim buyer into paying for the attacker's order. We found that the payment interfaces of all 35 *PPS*s display the order ID, the expected amount, and the pay-to account (i.e., QR code), but not buyer-oriented information, such as shipping address and merchandise recipient. One representative interface is shown in Figure 10. The order ID is the only information designed for identifying the attribution of this payment account, but according to our analysis it actually does not help buyers recognize the order attribution. Generally, when the displayed price in the interface of these 24 vulnerable *PPS*s matches a buyer's expectation, he is prone to pay for it without checking the order details. As a result, the attacker can easily perform the payment substitution attack. There are 9 *PPS*s that provide additional order details such as product names. Such additional information does help the user to recognize the order substitution, which limits the attacker's swap.

Figure 10: A representative payment interface shown at the *MC* in the *PPS*.

4.2.6 Missing Security Guideline in User Documents

One of the key questions we aim to answer is whether there is clear and concise guidance on the best security practices for developers. Since the majority of deployment vulnerabilities we found are related to local key storage, and the historical order inquiry is the important step for identifying tampered orders, we are focusing on the provided guidance around these issues. Our analysis shows that more than half of *PPS*s do not provide important security guidelines for either **KEY** storage or *order verification*. According to our analysis on 35 *PPS*s' documents, we found only 15 of them give the tip that the **KEY** must be kept in the *MS*. The lack of this guideline has contributed to many developers using the remaining 20 *PPS*s mistakenly placing the **KEY** at *MC*. For historical order inquiry, we only found 3 *PPS*s that suggest merchants adding additional verification steps to record the order payment history.

4.2.7 Insecure Network Transmission

Additionally, network transmission security is a common requirement to prevent man-in-the-middle attacks. Insecure network communication is one of the key enablers of theft of important parameters transmitted between *MS* and *PMP*. Although we do not mention this trivial attack in the security analysis since it has a strict requirement to sniff the channel between *MS* and *PMP*, we still provide related statistics here. We find that among 35 *PPS* platforms, 17 *PPS*s use the *HTTP* protocol to deliver the packet for the **KEY** changing response. These 17 *PPS*s all place the new **KEY** in cleartext form in the response packet. As a result, merchants adopting their payment services are theoretically at risk of **KEY** sniffing attacks.

4.3 Cases for Real-world Attacks

To validate the vulnerable patterns we analyzed, we conduct several real-world attacks on our own merchant account to understand the feasibility and limitations of these attacks. The video recordings and related MD5 attack materials for all attack experiments are provided in [9]. Note that all the attacks demonstrated in our paper are launched against our own merchant and user accounts, even though they apply generally.

4.3.1 String Shift Attack

We choose the *Paysapi* website and perform the string shift attack, aiming to recharge a certain amount to our registered account but pay less than the amount. From our analysis, we know that *Paysapi* has a vulnerable token generation method which simply concatenates all the fields together before hashing, i.e., it is vulnerable to the string shift attack discussed in Section 3.3.2. We use the *Fiddler 4* tool to intercept, counterfeit, and re-send the transmission packet between our personal computer and the *Paysapi* server.

We type 30 Chinese Yuan in the input field of *Paysapi* user interface and send this order request to the merchant. After the merchant sends the order packet back to the user interface, we use the *Fiddler 4* tool [5] to intercept the packet while preventing the user from delivering it to the *Paysapi* server. We manually shift the character "0" from the price field of the order packet to the beginning of the *return_url* field and then forward the new packet to the *Paysapi* server. The payment interface will display only requiring 3 Chinese Yuan. After the user pays 3 Chinese Yuan, the payment interface displays the payment successfully; however, the website fails to show the notification page. We check the balance in our account and find it is 30 Chinese Yuan, but we only pay 3 Chinese Yuan. This indicates the success of our attack without being noticed by the *Paysapi* server. We left the balance as it and notified merchant of this test without using it to purchase any commodity for ethical consideration. The video recording of our attack is shown in the file named "string_shift_attack.mp4" in [9].

4.3.2 Key Changing Attack

We perform two real merchant-oriented attacks where the first one targets disabling the merchant's service and the second one targets stealing **KEY**. Merchant accounts are registered on two different *PPS*s, the *Paysapi* and the *Xunhupay*, *Paysapi* includes the new key in the reply, while *Xunhupay* doesn't. Our goal is to demonstrate the vulnerabilities in the **KEY** changing interface.

For the test on *Xunhupay*, we log in through the merchant portal and see that the current **KEY** is "r7Ep7kymuyQQE6taVQNF". We operate the registered account and click the key changing button. Meanwhile, we use the *Fiddler 4* tool to monitor the request packet for the key changing transmission to the *PPS*. *PPS* changes the **KEY** to "hiTcAvcYicv24XjdcwRY". However, *PPS* does not send a response packet carrying the **KEY**. Instead, the registered account needs to refresh his profile page to see this new *KEY*. We extract the merchant ID information from the monitored packet, forge a new request packet, and send it to the *PPS*. *PPS* proceeds to change **KEY** to "Tk7pK5BneK2373NuU76E". As there is no response packet sent from *PPS* to the registered account for *KEY* change, the merchant's service will be disabled until he refreshes his profile page to notice the key is changed.

For the test on *Paysapi*, we log into the system and follow the same steps as in the experiment above. However, since *Paysapi* includes **KEY** in the response. It is possible to steal the new key just by examining the key change response. The video recordings of these two attacks are shown in [9], with the files named as "disabled_attack.mp4" and "key_stealing_attack.mp4".

4.3.3 Payment Substitution Attack

The goal of this attack is to verify the ability of an attacker to swap out the content of an order such that the victim pays for the order of the attacker. Two user accounts are registered, one for the victim and the other for the attacker. The victim user and the attacker are placed in the same local area network under the same router.

First, the victim opens his interface and clicks to recharge the account with 10 Chinese Yuan. When the attacker monitors that the victim is recharging the account with 10 Chinese Yuan, he intercepts the packet sent from the merchant service to the victim and blocks its transmission to the *PMP*. Meanwhile, the attacker clicks the recharging function in his account. We intercept the packet transported from the merchant server to him without forwarding it to the *PMP*. We record the important order parameters in this packet and replace them with those in the intercepted packet from the victim. The new

substituted packet is then sent to the victim's *PMP*. The payment interface is successfully shown on the victim's website, which does not show any order identity information. The victim makes the payment for the attacker's order. We found the balance in the attacker's account is shown as 10 Chinese Yuan now, but the balance in the victim's account remains unchanged. This indicates the success of our payment substitution attack. The video recording of this attack is exhibited in [9] with the file name of "substitution_attack.mp4"

4.3.4 MD5 Collision Attack

The goal of performing this attack is to verify that the MD5 collision attack is practical in real-world *PPS* systems. We carefully choose the donation payment on a blog website as the attack target since donating a lower amount will not cause serious effects. We notified the owner of the blog for this experiment. This blog employs the payment services provided by Paysapi, whose **Token** generation mechanism arranges the price directly in front of an optional field "$Return_{URL}$". The donation price is set to be 0.02 Yuan, and the $Return_{URL}$ can be obtained by capturing a normal order packet using the Fiddler 4 tool so that the parameters for performing the MD5 collision attack can be collected. We use an open resource on github [40] to calculate the chosen prefix MD5 collision blocks, where the two prefixes differ in the price values, "0.01" and "0.02". The calculation was processed on a computer with CPU: Intel i7-8700k, GPU: NVIDIA GeForce GTX1080 Ti, and RAM: 64G, where the CUDA was employed. It takes 7 days to find a collision with the two prefixes where the collision block is free from JSON keywords and the MD5 value is the same, "9f1ec604dce1bc1c0b1dd368dda3dd44". We start to make the payment (i.e., donation) on the blog and block the network packet sent from *MC* to the *MS*. Since the price and $Return_{URL}$ are generated in the website client, we modified the price and $Return_{URL}$ fields with the collision block of 0.02 Yuan. After the order packet is signed at *MS* and sent back to the *MC*, we again block and capture its transmission from *MC* to the *PMP* server. The MD5 collision results are stored in two ".bin" files; therefore, we use a hex editor to open the blocked order packet and replace the price and the $notify_{URL}$ with the collision block of "0.01" Yuan. This order passes the *PPS* certification, and the payment interface is successfully sent back to the website, then we pay for it at a lower price. The two collision files in this attack are exhibited in [9] with the file names of "prefix1.txt.coll" and "prefix2.txt.coll".

4.4 Ethical Consideration and Responsible Disclosure

4.4.1 Ethical Consideration on Real-world Attacks

We carefully designed and conducted our case study to avoid impact on real-world entities. We have conducted the attack on credit recharging transactions such that no real product would be shipped as a result of the attack. In all experiments, we made use of our test accounts created solely for demonstrating the attacks. We also did not receive any services and goods using the hijacked payment systems. At the end of our experiments, we always let the authority know the detailed procedures and results so that they can correct at the back end.

4.4.2 Responsible Disclosure

We first reported all our findings to the *PPS* providers in January 2020. Unfortunately, no formal email responses were received in the first round before March 2020, while 9 *PPS*s providers, *Bufpay, Xunhupay, 020zf, Paycats, Heimipay, Qianmapay, paysapi, Greenyep,* and *Xddpay*, gave us feedback through other online chat tools. During the second *PPS* inspection on August 2020, we found that 5 *PPS*s had updated the payment protocols with safer string concatenation mechanisms. The list of these *PPS*s is shown in Fig. 9. 12 of 35 *PPS*s which possess multiple vulnerabilities stopped providing payment services after our report, including *Yktapi, Caiwumao, Weimifu, ARYA, L Pays, Xinyipay, 188pc, PersonalP, Yijinka, 7cPay, Yuandian,* and *Dunpay*. We have also reported the vulnerabilities to the CVE on August 8th, 2020, and received their vulnerability confirmation on August 10th, 2020. However, our reported issues do not match the requirements for applying for a CVE ID.

As we will discuss in Section 5.1, some critical vulnerabilities are design flaws in the *PPS* protocols. Therefore, all existing merchants supported by *PPS* are at risk. Since all the *PPS*s adopt the personal money transfer service from the Alipay and WeChat Pay third-party payment platforms, on August 14th, 2020, we reported the vulnerable *PPS* list and the security issues to the Security Response Center of Tencent (WeChat Pay) and the Alibaba Security Response Center (AliPay), which are responsible for the security of their payment ecosystem. The Alibaba Security Response Center has confirmed our reported *PPS* issues on September 20, 2020, and will continue to monitor technical reports in this area to improve payment security in *PPS*.

5 PPS Vulnerability Summary

Through the lens of our study, *PPS*, an emerging payment platform that aims to bring together the personal payment account on third-party platforms and an independent order management platform, still faces challenges in security and usability despite its popularity. In this section, different attacks are grouped and analyzed from different perspectives, with the goal of shedding light on this new system's security in the future.

5.1 Flaws Classification

The vulnerable patterns described in Section 3 fall into issues in either protocol or implementation.

- *PPS Protocol Vulnerability* - Protocol vulnerabilities are due to insecure design in the payment protocol, and it is very hard for the merchants to correct them by themselves. Among seven vulnerabilities, *unprotected Key changing API, simple String Concatenation in Token Generation, MD5-based Token Generation, Lacking Order Examining Mechanisms,* and *No Paid Value in Order Inquiry* fall into this category.

- *Implementation Vulnerability* - The vulnerabilities in this category are often business and implementation specific, i.e., caused by a lack of understanding of either the *PPS* API or secure design. *Local Order Generation* and *Local KEY Storage* are the two majority vulnerabilities in this category. Correcting these problems is easier, requiring changes only at the client side.

5.2 Attack Classification

Based on the attack motivation, five attacks proposed in Section 3 can be grouped into three categories as follows:

- *Malicious User Attack.* In this category, an attacker manipulates his own client in an attempt to benefit himself, such as by reducing the price of the item under purchase. The attacks of *Order Tampering with KEY, String Shift in Order Packet,* and *MD5 Signature Collision* fall into this category.

- *Victim-oriented Attack.* In this attack, the adversary can intercept the communication packets between other users (victim *MC*) and *MS*. His goal is to achieve financial gain by manipulating network packets. *Payment Substitution attack* belongs to this category.

- *Merchant-oriented Attack.* An attacker in this category primarily aims to attack the merchant or other user from the Internet, without access to the victim's traffic or devices. *Malicious KEY Changing* belongs to this type.

5.3 Defending the Attacks and Improvement

From the perspective of *PPS* providers, it is possible to make the following changes: 1) **KEY** changing API should require authentication; 2) The newly issued **KEY** should be transmitted as ciphertext or via a secure connection such as *HTTPs*; 3) Adopt strong cryptographic primitives instead of MD5; 4) Display clear payment and order information to users in a unified manner; 5) Provide actual paid amount in both the order payment status notification and the order inquiry results.

From the perspective of merchants, the following changes might be beneficial: 6) **KEY** should never be stored at the client side; 7) The order parameters should be generated and signed at the server, and then transmitted to the user client; 8) Check the actual paid value for each order when receiving the payment notification.

Finally, users need to carefully check order information: 9) Always check the payment attribution information, such as order ID, before making payment; 10) Avoid making payments over insecure links.

6 Related Work

6.1 Security in Branchless Banking

The branchless banking system is the foundation of modern e-commerce, and its security has been studied in several previous works. In particular, the analysis started with the classic paper where Anderson first raised questions on the security of banking systems [13]. Since then, there has been extensive works done studying the enhancement of password-based authentication in mobile banking [33, 39]. At the same time, SMS-based mobile bank application is a common design which is vulnerable to attacks on messages [27], so there have been plenty of studies aiming to improving such mechanisms [16, 17, 26]. Even though there have been significant amount of efforts on securing the mobile banking systems, recent studies are still raising concerns on the security of existing systems [12, 22, 31, 32]. A comprehensive study of the existing branchless banking applications in different counties was conducted in [35, 36]. By analyzing the application communication flow, the authors found critical vulnerabilities that can lead to compromised transaction integrity in six of the seven applications. While these efforts are related to our work, we are analyzing a new paradigm that couples online banking and third-party management systems, which faces unique problems in security.

6.2 Security in Online Payment System

The existing work on online and mobile payment systems with third-party platforms mainly fall along two lines, i.e., web-based and in-app payment systems.

6.2.1 Web-based Payment System Analysis

For web-based payment systems, existing security analysis has primarily focused on merchant websites that integrate third-party platforms. In [48], Wang *et al.* focused on vulnerabilities in several popular online stores that adopt third-party payments, like *PayPal* and *Amazon Pay*. Dynamic protection strategies were proposed for automatically protecting the third-party web services in [52]. In [46], the static detection methods based on the symbolic execution framework were

proposed to detect the vulnerabilities in merchant websites. Furthermore, [18, 34] generalized black-box detection techniques across multiple web applications, with the analysis based on network traces or user behaviors. [45] proposed pattern-based attack methods to automatically generate test cases for checking security issues of multi-party web applications.

6.2.2 In-app based Payment System Analysis

The other line of research focused on exploring the vulnerabilities for in-app based payment systems. In [37], it was found that attackers can bypass server-side validation in *Google-developed in-App Billing* to make a purchase for free. Following this line, a tool named *VirtualSwindle* was later proposed in [25] to automatically target the in-app billing service for shopping free in Android applications. Closely related to our work is a systematical security analysis of third-party in-app payment services in the Chinese market [53, 54]. They outlined seven secure design patterns for constructing a secure transaction process and discussed the potential impact of not following them.

6.2.3 Other Payment Methods Analysis

In [15], Chen *et al.* explored syndication payment services by analyzing the user documentation via NLP-based techniques on syndication services to detect logic vulnerabilities. Moreover, credit, debit, and gift card usage in online payment suffers from card counterfeiting. Using keyloggers and cameras, attackers can steal card data and forge a copy [23]. By monitoring network transmission, a counterfeit card can be created to shop in the real world store [47]. To protect gift card security, in [38], a new method to detect counterfeit gift cards without needing to scan the original was proposed.

However, none of the existing work focuses on the *Personal Payment System (PPS)*, which is a newly emerged payment service that has different system mechanisms and customers from those explored in the existing work. To the best of our knowledge, we are the first to offer a systematic study to reveal security issues in *PPS*.

6.3 MD5 Collision Techniques

The payment system relies heavily on a secure signature mechanism to prevent packet transmission tampering. Since the *PPS* system generally adopts the MD5 as the Token generation hash function, we reviewed works on the MD5 collision attack to help examine payment security. The MD5 collision was first noticed by Den Boer and Bosselaers in [19], which demonstrated that two different vectors can produce an identical digest. In [49], a full MD5 collision was generated by Xiaoyun Wang's group, indicating that MD5 has gradually become an insecure digest method. In addition, a practical collision case of two X.509 certificates with different public keys resulting in the same MD5 hash value was provided in [24]. After that, plenty of works improved the MD5 collision approaches, from the identical prefix collision to the chosen prefix while shortening the calculation time [14, 41–44], demonstrating the weakness of MD5. In this paper, we leverage open resources to conduct the MD5 collision experiments and design the attack scheme targeting the vulnerabilities in the *PPS* payment process.

7 Conclusion

Personal Payment System (*PPS*) represents an emerging paradigm where small business owners leverage an independent management platform in combination with a personal financial account on a third-party payment system to conduct e-commerce. However, the added complexity in composing the two independent services for payment transaction and order management significantly increases the risk of security vulnerabilities. In this paper, we studied the 35 most widely used *PPS*s supporting more than 20 million users and presented an abstracted model that captures the common design elements within these systems. In our security analysis of these systems, we found 7 vulnerable patterns in these designs. By chaining these vulnerabilities together, we presented 5 proof of concept exploits of these vulnerabilities. Moreover, we also conducted four real-world attacks to allow an attacker to purchase items at a lower price without a trace. We have designed and conducted all the experiments on our own accounts to minimize the impact on real customers. Following the practice of responsible disclosure, we have also reported to and worked with vendors to fix some of the vulnerabilities. Lastly, we put forth a set of suggestions for future deployments of *PPS*.

Acknowledgement

This work was supported in part by Louisiana Board of Regents under Contract Numbers LEQSF(2018-21)-RD-A-24 and in part by US National Science Foundation under grants CNS-1837519, CNS-1916926, and CNS-1948374.

References

[1] Alibaba security response center. `https://security.alipay.com/`.

[2] Alipay. `https://www.alipay.com`.

[3] Androguard. `https://github.com/androguard/androguard`.

[4] Apple pay. `https://www.apple.com/apple-pay/`.

[5] Fiddler 4. https://www.telerik.com/fiddler.

[6] Paypal. https://www.paypal.com/us/home.

[7] Tencent security response center. https://en.security.tencent.com/.

[8] Venmo. https://venmo.com.

[9] Video records and md5 attack files for attacks in the case stuty. https://www.dropbox.com/sh/kbo321oaw03qils/AAAJSUmncKo3heKY0BOZnSi4a?dl=0.

[10] Wexpay. https://pay.weixin.qq.com.

[11] Fastpay PPS. http://www.weixin.mobi/, 2020.

[12] Gilberto Marins de Almeida. M-payments in brazil: Notes on how a country's background may determine timing and design of a regulatory model. *Wash. JL Tech. & Arts*, 8:347, 2012.

[13] Ross Anderson. Why cryptosystems fail. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 215–227, 1993.

[14] John Black, Martin Cochran, and Trevor Highland. A study of the md5 attacks: Insights and improvements. In *Proceedings of International Workshop on Fast Software Encryption*, pages 262–277. Springer, 2006.

[15] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *USENIX Security Symposium*, pages 747–764, 2019.

[16] Ming Ki Chong. *Usable authentication for mobile banking*. PhD thesis, University of Cape Town, 2009.

[17] Sheila Cobourne, Keith Mayes, and Konstantinos Markantonakis. Using the smart card web server in secure branchless banking. In *Proceedings of International Conference on Network and System Security*, pages 250–263, 2013.

[18] G Deepa, P Santhi Thilagam, Amit Praseed, and Alwyn R Pais. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109, 2018.

[19] Bert Den Boer and Antoon Bosselaers. Collisions for the compression function of md5. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 293–304. Springer, 1993.

[20] eMarketer. Global mobile payment users 2019. https://www.emarketer.com/content/global-mobile-payment-users-2019, 2019.

[21] eMarketer. Global mobile payment users 2019. https://www.emarketer.com/content/china-mobile-payment-users-2019, 2019.

[22] Andrew Harris, Seymour Goodman, and Patrick Traynor. Privacy and security concerns associated with mobile money applications in africa. *Wash. JL Tech. & Arts*, 8:245, 2012.

[23] B Krebs. All about fraud: How crooks get the cvv. http://krebsonsecurity.com/2016/04/all-about-fraud-how-crooks-get-the-cvv/, 2016.

[24] Arjen K Lenstra, Xiaoyun Wang, and BMM de Weger. Colliding x. 509 certificates. https://eprint.iacr.org/2005/067, 2005.

[25] Collin Mulliner, William Robertson, and Engin Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 2014 ACM symposium on Information, computer and communications security*, pages 459–470, 22014.

[26] Baraka W Nyamtiga, Anael Sam, and Loserian S Laizer. Enhanced security model for mobile banking systems in tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research*, 1(4):4–20, 2013.

[27] Baraka W Nyamtiga, Anael Sam, and Loserian S Laizer. Security perspectives for USSD versus SMS in conducting mobile transactions: A case study of tanzania. *international journal of technology enhancements and emerging engineering research*, 1(3):38–43, 2013.

[28] Packagist. Payjs PPS. https://packagist.org/?query=PayJs, 2020.

[29] Packagist. Paysapi PPS. https://packagist.org/?query=paysapi, 2020.

[30] Packagist. Xunhupay PPS. https://packagist.org/?query=Xunhupay, 2020.

[31] Michael Paik. Stragglers of the herd get eaten: security concerns for gsm mobile banking applications. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, pages 54–59, 2010.

[32] Saurabh Panjwani. Towards end-to-end security in branchless banking. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 28–33, 2011.

[33] Saurabh Panjwani and Edward Cutrell. Usably secure, low-cost authentication for mobile banking. In *Proceedings of Symposium on Usable Privacy and Security*, pages 1–12, 2010.

[34] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS*, 2014.

[35] Bradley Reaves, Jasmine Bowers, Nolen Scaife, Adam Bates, Arnav Bhartiya, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–31, 2017.

[36] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications in the developing world. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 17–32, 2015.

[37] Daniel Reynaud, Dawn Xiaodong Song, Thomas R Magrino, Edward XueJun Wu, and Eui Chul Richard Shin. Freemarket: Shopping for free in android applications. In *NDSS*, 2012.

[38] Nolen Scaife, Christian Peeters, Camilo Velez, Hanqing Zhao, Patrick Traynor, and David Arnold. The cards aren't alright: Detecting counterfeit gift cards using encoding jitter. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 1063–1076, 2018.

[39] Ashlesh Sharma, Lakshmi Subramanian, and Dennis Shasha. Secure branchless banking. In *Proceedings of ACM SOSP Workshop on Networked Systems for Developing Regions (NSDR)*, 2009.

[40] Marc Stevens. Md5 and sha-1 cryptanalytic toolbox. `https://github.com/cr-marcstevens/hashclash`.

[41] Marc Stevens. On collisions for md5. `https://www.win.tue.nl/hashclash/`, 2007.

[42] Marc Stevens, Arjen Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and colliding x. 509 certificates for different identities. In *Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–22. Springer, 2007.

[43] Marc Stevens, Arjen K Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and applications. *International Journal of Applied Cryptography*, 2:322–359, 2012.

[44] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne De Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In *Proceedings of Annual International Cryptology Conference*, pages 55–69. Springer, 2009.

[45] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, Luca Compagna, et al. Attack patterns for black-box security testing of multi-party web applications. In *NDSS*, 2016.

[46] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *NDSS*, 2014.

[47] American Underworld. Report on carding, skimming. `https://www.youtube.com/watch?v=kbrU9Jwhww`, 2012.

[48] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *IEEE Symposium on Security and Privacy*, pages 465–480, 2011.

[49] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35, 2005.

[50] WordPress. Xunhupay PPS. `https://wordpress.org/plugins/xunhu-wechat-payment-for-woocommerce/`, 2020.

[51] WordPress. Xunhupay PPS. `https://wordpress.org/plugins/xunhu-alipay-payment-for-woocommerce/`, 2020.

[52] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. Integuard: Toward automatic protection of third-party web service integrations. In *NDSS*, 2013.

[53] Wenbo Yang, Juanru Li, Yuanyuan Zhang, and Dawu Gu. Security analysis of third-party in-app payment in mobile applications. *Journal of Information Security and Applications*, 48:102358, 2019.

[54] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In *NDSS*, 2017.