An Empirical Study on Mobile Payment Credential Leaks and Their Exploits

Shangcheng Shi[1(✉)], Xianbo Wang[1], Kyle Zeng[1,2], Ronghai Yang[1,3], and Wing Cheong Lau[1]

[1] The Chinese University of Hong Kong, Hong Kong, China
{ss016,xianbo,wclau}@ie.cuhk.edu.hk
[2] Arizona State University, Tempe, USA
zengyhkyle@asu.edu
[3] Sangfor Technology Inc, Shenzhen, China
yangronghai@sangfor.com.cn

Abstract. Recently, mobile apps increasingly integrate with payment services, enabling the user to pay orders with a third-party payment service provider, namely Cashier. During the payment process, both the app and Cashier rely on some credentials to secure the service. Despite the importance, many developers tend to overlook the protection of payment credentials and inadvertently expose them to the wild. Such leaks severely affect the security of end-users and the merchants associated with the apps, resulting in privacy violations and actual financial loss. In this paper, we study the payment credential leaks for four top-tiered Cashiers that serve over one billion users and tens of millions of merchants globally. Through studying practical mobile payment systems, we identify new leaking sources of payment credentials and find 4 types of exploits with severe consequences, which are caused by the credential leaks and additional implementation flaws. Besides, we design an automatic tool, PayKeyMiner, and use it to discover around 20,000 leaked payment credentials, affecting thousands of apps. We have reported our findings to the Cashiers. All of them have confirmed the issue and pledged to notify the affected merchant apps, while some of these apps have updated the leaked payment credentials afterward.

Keywords: Mobile payment · Payment credentials · Security testing

1 Introduction

In the past decade, mobile payment service has become worldwide popular with total transaction value exceeding $1.15 trillion in 2019 [16]. Through the mobile apps of third-party Cashiers, end-users can perform the payment within smartphones readily instead of using cash or another physical token, *e.g.*, credit card.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 399, pp. 79–98, 2021.
https://doi.org/10.1007/978-3-030-90022-9_5

Given its prevalence, a wide range of mobile apps have integrated the service from the Cashiers. The Cashiers also release their SDKs and documents online to ease the deployment of mobile payment.

The mobile payment process involves sophisticated multi-party authentication and authorization. For security purposes, the Cashiers and the apps need to set up various payment credentials beforehand and make use of them in each session. The payment credentials include a collection of unique data (*e.g.*, RSA keys) and confidential files (*e.g.*, PKCS#12 files). With the credentials, either the Cashier or the app can authenticate itself to the other party and conduct privileged operations such as money transferring or refunding.

Given the significance of payment credentials, app developers *must* keep them private. Otherwise, an attacker can exploit the leaked credential to perform privileged transactions by impersonating the benign app. Despite the critical nature, prior research mainly focuses on the discovery of *general* credentials leaked from specific sources, *e.g.*, mobile apps [22] or public GitHub repositories [11]. In contrast, relatively few efforts have been spent to understand to what extent the *payment credentials* are being disclosed in the wild.

Towards this end, in this paper, we perform the first in-depth empirical study of payment credential leaks for four top-tiered Cashiers. These Cashiers serve over 1 billion end-users and tens of thousands of mobile apps. In particular, we aim to address the following three research questions: (1) *Where can the payment credentials be leaked?* (2) *What damage can the leaked credentials cause?* (3) *How to investigate the prevalence of payment credential leaks on a large scale?*

To solve the first two questions, we study the mobile payment services from the four Cashiers and their actual implementations by various merchant apps. Consequently, we find new leaking sources (to be discussed in Sect. 3), besides the known ones studied by previous works. For example, we find that even the backend servers of (merchant) apps, due to the insecure designs of SDKs by the Cashiers, can leak payment credentials unintentionally (Sect. 3.3). Moreover, we find four types of exploits based on leaked credentials that have severe consequences, ranging from privacy violations to financial loss. Notably, two of these exploits leverage extra implementation flaws, enabling the attacker to harm *other innocent apps or another third-party service, i.e., Single Sign-On (SSO).*

As for the third question, we design and implement PayKeyMiner to automatically identify the leaked payment credentials from all potential sources. Using this tool, we have conducted large-scale testing and discovered around 20,000 valid payment credentials, affecting thousands of apps. We summarize our contributions as follows:

- We have found new leaking sources of payment credentials.
- We have discovered 4 types of exploits with the leaked payment credentials.
- We have proposed an automatic tool, PayKeyMiner, to detect payment credential leaks in the wild.
- Using PayKeyMiner, we have uncovered around 20,000 payment credentials.

The rest of the paper is organized as follows. Section 2 introduces the mobile payment services and related credentials of four mainstream Cashiers. Section 3

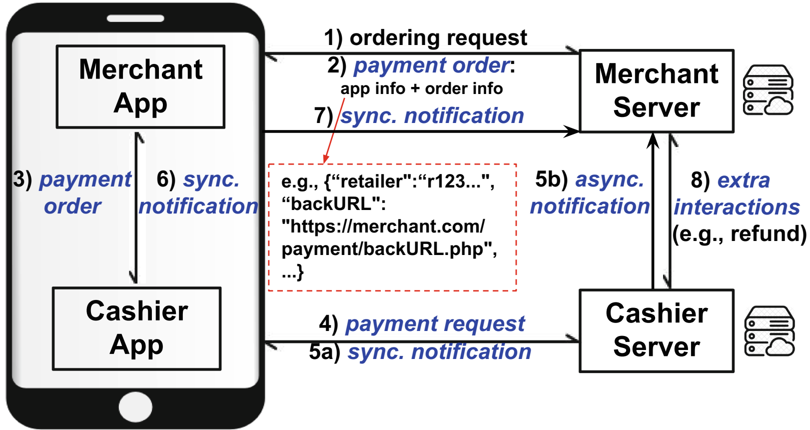

Fig. 1. General workflow of mobile payment service

discusses the leaking sources of payment credentials. Section 4 elaborates on how to exploit payment credential leaks. Section 5 describes the design and implementation of PayKeyMiner, while Sect. 6 presents the empirical testing. We review related work in Sect. 7 and conclude the paper in Sect. 8.

2 Background

A traditional e-payment service involves three parties, namely, the User (User-Agent), the Cashier, and the Merchant. In the context of mobile payment, the Cashier and Merchant map to their backend servers, *i.e.*, Cashier Server (*CS*) and Merchant Server (*MS*), while the User-Agent becomes their frontend mobile apps, *i.e.*, Cashier App (*CA*) and Merchant App (*MA*). For ease of presentation, we will use the notations in the parentheses in this section if not specified.

2.1 Workflow of Mobile Payment Service

The mobile payment service enables the Cashier to acknowledge to the Merchant that the user has paid the order in its app, *i.e.*, *MA*. Since there exists no standard for mobile payment, we review the technical documents from the Cashiers and get its general protocol flow (in Fig. 1), which works as follows:

1. After shopping in an *MA*, a user selects a third-party Cashier to check out. Then, the app sends a request containing the ordering details to its server.
2. The *MS* generates a payment order and feeds it back to its app.
3. The *MA* passes the payment order to the *CA*, which then displays the details, *e.g.*, *trade_amount* and *payee*, to the user for payment authorization.
4. Once the user confirms the payment, the *CA* sends a request to its server.

5. The *CS* processes the payment request and responds to the *CA*, *i.e.*, synchronous notification. It will also notify the *MS* through the *backURL* (in Step 2), *i.e.*, asynchronous notification, about the transaction completion.
6. The *CA* forwards the synchronous notification to the *MA*.
7. The *MA* sends the received notification back to its server.
8. Besides payment, Merchants may request the Cashier to refund paid orders, transfer money into a user account, or download transaction records, *etc.*.

Table 1. Summary of payment keys or the equivalent

Cashier	Payment credential	Usage	Assigned by the cashier?	Shared Cashier's public key among Merchants?
Cashier1 (a)[a]	Secret Key	HMAC	✓	N/A
	RSA (Private) Key	Digital signature	×	✓
Cashier1 (b)[a]	RSA (Private) Key	Digital signature	×	✓
	RSA′ (Private) Key	Digital signature	×	×
Cashier2	Secret Key	HMAC	×	N/A
Cashier3	PFX Cert	Digital signature	✓	✓
	Secret Key	HMAC	✓	N/A
Cashier4	Secret Key	HMAC	✓	N/A

[a] *Cashier1* provides two sets of mobile payment services.

2.2 Payment Credentials

Payment Key. Most of the messages in Fig. 1, in italic, are secured by either the digital signature or hash-based message authentication code (HMAC). Table 1 summarizes the related two types of payment keys and their security settings.

HMAC Secret Key. The Secret Keys in Table 1 belong to the category. When adopted, both the Merchant and Cashier will use the same Secret Key as the salt of a hash function, *e.g.*, MD5, to get the HMAC of payment-related messages. Apart from *Cashier2*, this type of key is generated by the Cashiers.

Signing Key. The other items in Table 1 are used to generate the digital signature. Then, both the Cashier and Merchant need to hold a pair of asymmetric keys and share their public keys. During mobile payment, either party will sign the request with its own private key and verify the response with the other party's public key. In this paper, we focus on the Merchant's private key.

As *Cashier1* supports two digital signing methods, it defines two types of payment keys, *i.e.*, RSA key and RSA′ key. Another crucial difference between these two keys is that *Cashier1 uses the same public key across the Merchants in RSA key, which is instead app-specific in RSA′ key.*

Besides, the Merchant's private key in *Cashier3* is included in a PFX certificate, which is a PKCS#12 file and password-protected. As such, the Merchant

needs to unlock its certificate to extract the private key inside to sign the messages in each payment session. Similar to the RSA key in *Cashier1*, *the public key of Cashier3 is shared among its Merchants*, enabling the exploit in Sect. 4.4.

Other Credentials. Some Cashiers define other credentials for better security.

Android Signing Key. When receiving the payment order, *i.e.*, Step 3 in Fig. 1, *Cashier2* and *Cashier4* will validate the *MA* by checking its package signature. Thus, the Merchants should keep their Android signing keys private.

Client Certificate [23]**.** *Cashier2* issues per-app based certificates to its Merchants. The file is in the format of PKCS#12 and password-protected, which is required in the requests to the Cashier, *i.e.*, Step 8 in Fig. 1, for authentication.

2.3 Threat Model

In our threat model, the attacker aims to steal the payment credentials from publicly available sources (in Sect. 3). Then, he can exploit these credentials to cause financial loss and privacy violations to victim Merchants and their users.

To be specific, the attacker may use the leaked payment credentials to forge messages, *e.g.*, a refunding request, to cheat either the Cashier or Merchant. Meanwhile, the attacker can behave like a normal user in the Merchant App and modify the messages that visible to his smartphone. The attacker can also package a malicious Merchant App and trick the victim users into using it.

3 Leaking Sources of Payment Credentials

We introduce the leaking sources of payment credentials here, among which public GitLab repositories and Merchant Servers have not been studied before.

3.1 Public Git Repositories

Meli *et al.* [11] find some developers push the production code to GitHub without removing their credentials. We further notice that such leaks are still prevalent nowadays, even for payment credentials. Moreover, previous works overlook the leaks that only appear in old commits, while we will bridge this gap (in Table 6).

Meanwhile, we find some companies establish GitLab services on public IPs and make their repositories public, which may contain payment credentials. As such, the attacker can download these repositories and get the credentials inside.

3.2 Mobile Apps

Many developers embed credentials in their mobile apps. The existing works, *e.g.*, [25], study the leaks in the latest version of apps, while such leaks may happen in obsolete app versions only. To address this limitation, we set up a full-scale

Android APK dataset (in Table 4) for our testing. As we will see in the empirical testing result (in Table 6), 31.9% of the leaked payment credentials only appear in old versions of mobile apps, which is beyond the scope of previous works. Since the iOS ecosystem is proprietary, its app packages are encrypted and unsuitable for large-scale testing, so we mainly focus on Android apps. Nevertheless, we still consider the leaks within the iOS projects from public git repositories.

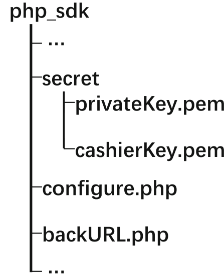

Fig. 2. Structure of an official SDK for Merchant Servers

3.3 Merchant Servers

We discover that the servers of Merchants can be another leaking source of payment credentials. The root causes are the insecure backend SDKs from Cashiers and the lack of access control on the related credential files by Merchant Servers.

The Cashiers provide SDKs to facilitate the deployment of mobile payment. Figure 2 shows the structure of an official SDK for illustration. For this SDK, "backURL.php" processes the asynchronous notifications, *i.e.*, Step 5b in Fig. 1. Besides, the "configure.php" file specifies the location of the Merchant's private key (Sect. 2.2) to a static file, namely "secret/privateKey.pem", while the document also requires the developer to store his private key in the same file. Consequently, this insecure practice enables the attacker to steal the private key file from Merchant Servers.

Specifically, the attacker can get the location of "backURL.php" from the payment order, namely, Step 2 in Fig. 1, *e.g.*, "https://x.com/pay/backURL.php". Based on the SDK structure in Fig. 2, he can then infer the location of the Merchant's private key, *i.e.*, "https://x.com/pay/secret/privateKey.pem". Once the Merchant Server does not block access to this file, the attacker can steal it.

As indicated in Table 2, some SDKs include payment credentials in scripts or other inaccessible files such that the attacker cannot steal them in most cases, while the other SDKs fail to follow this practice. According to our testing, 7.11% of the servers use these vulnerable SDKs and leak their credentials (in Table 6).

Table 2. Summary of backend SDKs from Cashiers

Inaccessible Credentials?	*Cashier1* (a)	*Cashier1* (b)	*Cashier2*	*Cashier3*	*Cashier4*
PHP	×	✓	×	×	×
Java	✓	✓	✓	✓	✓
C#	✓	✓	×	✓	✓

4 Exploiting Leaked Payment Credentials

The section presents four types of exploits caused by the leaked credentials. Leveraging extra implementation flaws, two of them enable the attacker to harm the other Merchant Apps without leaks and even another third-party service, *i.e.*, SSO. Due to ethical considerations, we cannot fully quantify the impacts of all these exploits, but we have tried to reproduce them in real payment systems.

4.1 Merchant Impersonation Exploit

Previous work [25] finds the attacker can use the leaked payment keys from mobile apps (Sect. 3.2) to get the transaction records of leaking apps illegally. Despite this known exploit, the attacker can also use leaked payment credentials to impersonate benign Merchants for more critical operations, *i.e.*, refunding and money transfer, **causing actual financial loss**. Although *Cashier2* requires the client certificate in these requests (Sect. 2.2), PayKeyMiner detects over 3,000 leaked certificates, which can be uncovered by the attacker (in Table 5).

Remark: The exploit of refund applies to all the leaking apps. As not all the Merchants activate the money transfer function, we cannot quantify the affected apps ethically. However, we have performed Proof of Concept (PoC) tests with demo Merchant accounts (from the Cashiers) under the production environment.

4.2 Android Package Signature Forgery

Although *Cashier2* and *Cashier4* validate the Merchant App (Sect. 2.2), many Android signing keys are leaked in public git repositories as the developers push the whole frontend project online. Thus, the attacker can modify several lines of code and package malicious Merchant Apps with valid signatures. Then, these malicious apps can **trick victim users into paying for the attacker's order** by replacing a payment order with the same amount, *i.e.*, Step 3 in Fig. 1.

Remark: We have performed PoC experiments with the leaked Android signing keys. Overall, PayKeyMiner detects 493 such leaks (in Table 5), where *10 of the related apps have more than one million downloads.*

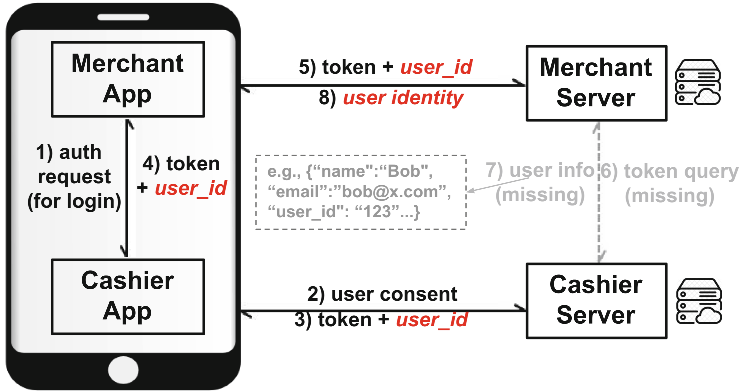

Fig. 3. Workflow of profile exploit

4.3 Backward SSO Attack

Some Cashiers, *i.e.*, *Cashier1* and *Cashier2*, provide SSO service, meaning the user can both login and pay the Merchant App through the Cashier. However, these Cashiers fail to isolate these two services, enabling the attacker to exploit leaked payment credentials to compromise the SSO service of Merchant Apps.

As depicted in Fig. 3, some Merchant Servers trust the *user_ids* from their frontend apps to authenticate the user without querying the Cashier in Step 6. Such flawed implementations enable the so-called Profile Exploit [17,24], where **the attacker may inject the *user_ids* of victim users into his sessions to hijack their accounts in the Merchant Apps**. Although both *Cashier1* and *Cashier2* set the *user_ids* to *private*, their payment and SSO services share the same set of *user_ids*, which appear in transaction records. Thus, the attacker may use leaked credentials to get the *user_ids* of all paying users and launch the exploit above. *We have performed the PoC for this exploit on real Merchant Apps with our own testing accounts.*

Meanwhile, the payment keys in *Cashier2* are generated by the Merchants (in Table 1), while its credentials for SSO, *i.e.*, *SSO_Secret*, are assigned by the Cashier instead. However, some Merchants reuse the values of *SSO_Secret* to be their payment keys, extending the impact of the leak issue in payment to SSO.

Remark: The *user_ids* in *Cashier1* are shared across Merchant Apps. Among the collected APKs (in Sect. 6.1), 26,413 apps (11.3%) include its SSO service and may be affected. Nevertheless, the *user_ids* in *Cashier2* are app-specific, so the exploit above only affects 497 leaking apps. Besides, we test 3,000 randomly-chosen leaked keys in *Cashier2*, and 43 of them (1.4%) act as the *SSO_Secret*.

4.4 Cross-App Payment Notification Forgery

Yang *et al.* [25] show the leaked HMAC secret key enables the attacker to forge notifications and cheat the leaking Merchant App. Even if the digital signature

is used, we find the attacker can still deceive the other innocent Merchant Apps with fake notifications, as Cashier's public key tends to be shared (in Table 1).

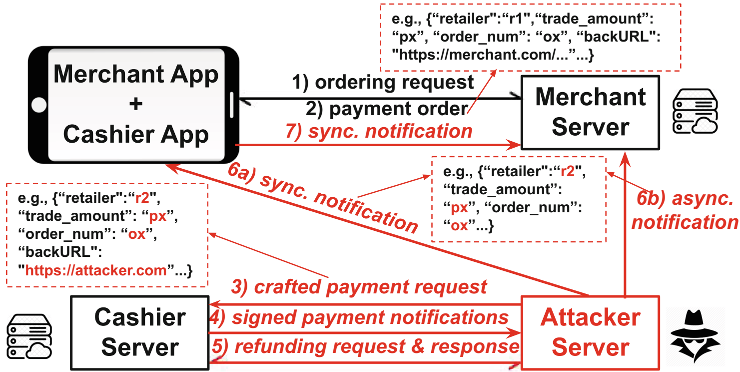

Fig. 4. Workflow of cross-app payment notification forgery

Figure 4 gives the workflow of this exploit. Specifically, the attacker may use a leaked signing key to create an order with the same *trade_amount* and *order_num* (in Step 3) as the target order in the victim Merchant App (from Step 2). After paying for this crafted order, the attacker can get signed notifications by specifying *backURL*, while he may refund the order later. Then, the attacker can send these notifications to the victim Merchant Server, which are cryptographically correct due to the Cashier's shared public key among the Merchants. Once overlooking the app identifier, *e.g.*, *retailer* in Fig. 4, in the notifications, **this victim Merchant will be cheated and let the attacker shop for free**.

Remark: As we cannot examine Merchant Servers without real attack, it is hard to quantify the impact of this exploit ethically. Nevertheless, we have set up a Merchant Server using the server code in a leaking GitHub repository, which is likely for production. Besides, we configure our own credentials (under Cashiers' sandbox environment) in the server and complete the PoC for this exploit.

5 System Architecture of PayKeyMiner

To investigate the prevalence of payment credential leaks, we build an automatic tool called PayKeyMiner. We give its system design and implementation here.

5.1 System Overview

Figure 5 presents the system architecture of PayKeyMiner, which consists of three modules. Crawler (Sect. 5.2) first identifies the git repositories and Android APKs that are likely to support mobile payment. Then, Scanner (Sect. 5.3) analyzes the filtered input and recognizes all potential credentials. Finally, Detector (Sect. 5.4) processes and validates these suspected credentials.

5.2 Crawler

PayKeyMiner crawls public git repositories and Android APKs and detects the payment-related ones. Although there are three leaking sources of payment credentials (in Sect. 3), we cannot directly locate the Merchant Servers. Thus, we use the URL Enumerator in Scanner to detect such leaks later (in Sect. 5.3).

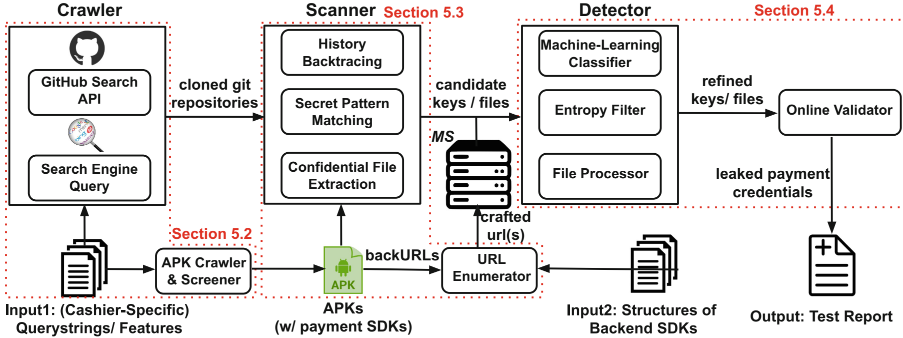

Fig. 5. System architecture of PayKeyMiner

Table 3. Examples of the constructed query strings

Type	Cashier	Sample	Illustration
Data	*Cashier1*	CUXwL**	Segment of Cashier's public key
Code	*Cashier2*	**PayConfigure	Classes in backend SDK
File	*Cashier3*	**PayPlugin.a	Files in iOS SDK

Git Repository Crawling. We count on GitHub Search API [8] and search engines like Google to find public git repositories. To get payment-related input, we study practical payment systems and summarize some *invariant* patterns that app developers cannot circumvent in integrating the mobile payment service. Table 3 shows some examples for better illustration, which can be categorized into three types, namely, data, code, and file. For instance, the RSA public key of *Cashier1* is shared among the Merchants (in Table 1), so the developers have to include it in their servers to validate the incoming messages from the Cashier. Thus, we construct query strings based on these invariants to filter out the git repositories irrelevant to the payment service.

On the other hand, GitHub only returns 1,000 search results, which cannot be tackled by previous works, *e.g.*, [11]. Fortunately, we find the *file_size* metric to

partition the search results into continuous ranges and bypass the query quota, enabling PayKeyMiner to cover almost all the related public GitHub repositories. However, similar functionality does not exist in search engines like Google. As a workaround, our tool merges the search results from several sources and identifies related GitLab repositories from them.

APK Database Setup and Screening. As many Merchant Apps work in certain countries only, where Google service is unavailable, most of them do not appear in Google Play. Thus, we take three third-party app markets as data sources, namely, Apkpure [2], Anzhi [1], and Wandoujia [20], while PayKeyMiner crawls all the available APKs from them, including older versions.

Moreover, we identify the APKs with payment services based on some fingerprints of the frontend SDKs from Cashiers that can survive the obfuscations. Specifically, these Cashiers explicitly require the developers not to obfuscate some classes within their frontend SDKs. These classes serve as entry points for inter-app communications, *e.g.*, Step 3 in Fig. 1, and renaming them will make these SDKs out of work. For the same reason, most frontend SDKs require the Merchant App to declare certain Activity entry points in AndroidManifest.xml, which is another fingerprint that can even survive bytecode packing. Although our tool cannot handle stronger packers that encrypt the original AndroidManifest.xml, less than 10% of Android apps adopt this kind of packer [6].

5.3 Scanner

This module scans the filtered input from Crawler to get suspected credentials. Given the target difference, it works in either a whitebox or blackbox manner.

Whitebox Scanning. This type of scanning applies to git repositories or APKs, whose code is available. The Scanner first traces back the history of the input, as the developers may remove the leaked credential in its latest version. Specifically, our tool uses *git-log* to get the previous modifications in the given git repository. As to APKs, the Scanner groups all its old versions and decompiles them for subsequent processing.

To identify candidate payment keys, we manually learn their patterns from the official technical documents from the Cashiers and formulate the corresponding regular expressions. For example, the Secret Key in *Cashier2* is an alphanumeric string with a length of 32. Then, we apply text-based pattern matching algorithms to the input. We take this language-agnostic method because the input git repositories can be written in various programming languages.

The Scanner also searches for credential files, *i.e.*, PKC#12 files and Android signing keys (in Sect. 2.2). Since these files are for specific purposes, they use constant file extensions, *e.g.*, .jks, so that our tool can easily identify them.

Blackbox Scanning. URL Enumerator (in Fig. 5) scans the Merchant Servers to get the exposed credential files caused by flawed backend SDKs (Sect. 3.3).

To examine the Merchant Servers, we need to identify their *backURL*s, which only appear in runtime, *i.e.*, Step 2 in Fig. 1. Besides, it is hard to extract these *backURL*s by simulating user behaviors in each Merchant App, as it involves the complex processes of authentication and shopping. Fortunately, some developers embed the *backURL*s in APKs, making them available after the decompilation.

Meanwhile, we study the structures of backend SDKs, *e.g.*, Fig. 2, for the relative paths from their *backURL* handler scripts to credential files, *i.e.*, Input2 in Fig. 5. With collected *backURL*s, the URL Enumerator then crafts and probes the URLs that may point to the exposed credential files in Merchant Servers. For ethical considerations, we use the HTTP HEAD method to get the metadata of these credential files to check their existence without downloading them locally.

5.4 Detector

For better efficiency, the Detector pre-processes the input from Scanner first with three sub-modules, namely Machine-Learning Classifier, Entropy Filter, and File Processor, to remove the false positives or reduce their impact. Finally, Online Validator validates suspected payment credentials.

Machine-Learning Classifier. The output from the Scanner contains the false positives caused by data files, *e.g.*, system logs, as many developers back up their servers in public git repositories. To distinguish configuration files from data files, we develop a machine-learning-based classifier. Specifically, we use file content, file name, and file path as the features, which are encoded and fed to a model based on XGBoost [4]. Then, we train this model with a dataset of 18,663 files from the initial test output, which have been manually tagged. The resultant model gives 97.47% accuracy with a 2.45% false-positive rate and a 5.63% false-negative rate. In runtime, the sub-module prioritizes the candidates from configuration files to reduce the overhead by false positives.

Entropy Filter. Some payment keys are generated by the Cashier (in Table 1), which may have similar Shannon Entropy [11]. Thus, we run the prototype of our tool to collect enough valid keys as ground truth. Consequently, we deploy Entropy Filter based on their entropy distribution. In runtime, this filter screens out the outliers that are 2 standard deviations away from the mean, as they have little chance to be the true positives ($< 5\%$). We randomly select and retest 500 input screened out by the filter, and none of them turns to be the false negative.

File Processor. As credential files, namely, PKCS#12 files and Android signing keys (in Sect. 2.2), are password-protected, the tool needs to crack them. Meanwhile, it validates Android signing keys by identifying associated apps here.

Notably, it is the Cashiers that assign these PKCS#12 files and guide their settings. For example, *Cashier3* suggests the password be a 6-digit. With the domain knowledge, our tool cracks over 90% of files using John the Ripper [13].

However, the passwords of Android signing keys are user-defined without guidelines. Thus, our tool tries with all available strings from the original git repositories and succeeds in unlocking 61.3% of the files. Based on the Android package names related to these cracked files, the File Processor then identifies the associated Merchant Apps and downloads their APKs. By comparing the hash values of an Android signing key and the corresponding APK, the tool manages to check the activeness of the former. Overall, PayKeyMiner has detected 493 valid Android signing keys (in Table 5).

Online Validator. This sub-module uses the suspected credentials to prepare an order query request with invalid parameters to the Cashier, *i.e.*, Step 8 in Fig. 1. This request is expected to be cryptographically correct but logically incorrect, so the Cashier will return an error message. For example, “Trade_NOT_EXIST” in Fig. 6 means that the given payment credential is valid, while “ILLEGAL_SIGN” indicates the false positive. Thus, we use this heuristic to validate the payment credentials without violating privacy. Note that the tool accesses the Cashier Servers in production mode but not sandbox mode, so the detected payment credentials are owned by the real Merchant Apps. Besides, to avoid affecting the normal production of the Cashiers, we set a timeout of 30 min for each test and control the interval between requests to 1 s. On average, it takes PayKeyMiner around 250 s to test each input.

```
<?xml version="1.0" encoding="utf-8"?>
<Cashier><is_success>F</is_success><error>ILLEGAL_SIGN</error></Cashier>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Cashier><is_success>F</is_success><error>TRADE_NOT_EXIST</error></Cashier>
```

Fig. 6. Sample of error messages from the Cashier

Table 4. Statistics of payment SDK integration in APKs

	#Total	*Cashier1*	*Cashier2*	*Cashier3*	*Cashier4*
#Android App	233550	29269 (12.5%)	46408 (19.9%)	7234 (3.1%)	466 (0.2%)
#App Version	1240961	182124 (14.7%)	233262 (18.8%)	53516 (4.3%)	3596 (0.3%)

Table 5. Summary of the leaked payment credentials

Cashier	*Cashier1* (a)		*Cashier1* (b)		*Cashier2*			*Cashier3*		*Cashier4*	
Source\Credential	Secret Key	RSA Key	RSA Key	RSA′ Key	Secret Key	Client Cert	Android Key	PFX Cert	Secret Key	Secret Key	Android Key
GitHub Repo	900	944	574	1737	6651	3131	491	188	0	25	1
GitLab Repo	9	12	8	20	57	31	1	1	0	0	0
Android APK	75	1766	184	354	2567	3	N/A	0	2	10	N/A
Merchant Server	0	44	N/A	N/A	0	11	N/A	2	0	0	N/A
Overall†	975	2578	754	2085	9093	3170	492	189	2	34	1

* Each row has deleted duplicate items. † The overlap among different sources is removed.

6 Empirical Testing

This section presents our empirical testing and the test results from PayKeyMiner.

6.1 Dataset

Our dataset includes 139,206 GitHub repositories related to the payment services. Meanwhile, we collect 943 GitLab repositories from four search engines, *i.e.*, Google, Yahoo, Bing, and Baidu. On the other side, we crawl 233,550 Android apps, with overall 1,240,961 versions (*i.e.*, APKs), in September 2019. As discussed in Sect. 5.2, we preprocess these APKs, and Table 4 gives the result.

6.2 Test Results

Table 5 shows the test results, where PayKeyMiner has detected around 20,000 unique and valid credentials. Specifically, 10.34% of public git repositories, 3.21% of APKs, and 7.11% of Merchant Servers leak payment credentials (in Table 6).

Git Repositories. Through the online validation (in Sect. 5.4), our tool detects 23,011 valid payment credentials from 14,493 public git repositories, including the duplicate credentials (in Table 6). Among these results, 1,792 credentials (7.79%) only appear in old git commits, indicating that *some developers have noticed the leak issue and fixed it wrongly by pushing new commits without revoking their credentials.* On average, it takes the developers around 51 days to make this wrong fix. On the other hand, the other leaked credentials exist in the latest version of code and are available from GitHub search results [8].

Meanwhile, the four search engines perform differently in detecting public GitLab repositories, where Google contributes most of the results (in Table 6). Besides, we find that most of these repositories are owned by some *outsourcing companies.* These companies are responsible for the deployment of payment service but not its maintenance, so that they may not care about security.

Table 6. Statistics of the empirical testing

Source	GitHub	GitLab				Android APK	Merchant Server
		Google	Baidu	Yahoo	Bing		
#Input	139206	410	288	223	51	233550	802
		Overall: 943					
#Detection	14419 (10.36%)	62	9	2	1	7492 (3.21%)	57 (7.11%)
		Overall: 74 (7.85%)					
#Credential Leak	22830	181				9011	67
#Leak in History	1765 (7.7%)	27 (15.0%)				2878 (31.9%)	N/A

* The table shows the result before removing duplicate credentials.

Table 7. Statistics of Android apps with credential leaks

#Download	$(0,10^4)$	$[10^4,10^5)$	$[10^5,10^6)$	$[10^6,10^7)$	$[10^7,10^8)$	$[10^8,\infty)$	#Total
#Leaking App	2606	4698	132	35	17	4	7492

As mentioned in Sect. 3.2, we also search for payment credential leaks in iOS-related git repositories. From the collected input, PayKeyMiner has identified 365 and 347 unique payment keys in *Cashier1* and *Cashier2* separately.

Android APKs. PayKeyMiner detects 9,011 payment credentials, including 4,961 unique ones, from 7,603 APKs that are associated with 7,492 apps. Surprisingly, we find 3 client certificates in *Cashier2* are leaked with payment keys, where the developers implement the server-side operations, *e.g.*, refunding, in their apps. Notably, 31.9% of these credentials only exist in old app versions.

The affected apps range from an official tax-payment app to a financial app with over 5 million downloads, and we will discuss them in Sect. 6.5. As indicated in Table 7, most of these leaking apps have less than 1 million downloads, while 4 of these Merchant Apps have over 100 million downloads.

We also measure the occurrence frequency of the leaked payment keys in APKs. The result indicates that two of these keys appear in 1,446 and 419 APKs separately, which are owned by two *payment syndicators* [5]. The *payment syndicator* helps Merchant Apps to integrate the payment services from multiple Cashiers. Then, Merchant Apps can use the *payment syndicator*'s Cashier account for their payment services. However, these two *payment syndicator*s embed payment keys in their frontend SDKs to Merchants, affecting all the related apps.

Besides, we analyze the leaking locations of payment keys in APKs. From the result, over 1,800 keys are leaked from three particular files, *which belong to the official demo project from* one of the Cashiers under our study. Although the code claims that it is for demo use and gives a serious warning on the credential leak issue, many developers still reuse it for ease of implementation.

Merchant Servers. Our tool detects 57 exposed credential files from 802 Merchant Servers. As many Merchant Apps do not embed *backURL*s, we cannot

obtain their values directly. However, the attacker may dynamically execute the app and get its *backURL* from the payment order, *i.e.*, Step 2 in Fig. 1. Notably, iOS apps can also leak payment credentials from their servers due to the insecure backend SDKs (Sect. 3.3). From the collected GitHub repositories, we have manually found such cases in the server code of some iOS projects.

6.3 Resolving the Merchant Apps

Some payment credentials are leaked from the server code in git repositories so that we cannot identify the associated Merchant Apps directly. To bridge the gap, we develop the following two approaches.

Crafting Payment Request. Many payment credentials work for both mobile apps and websites. Therefore, we can use the leaked credential to craft a payment request for the latter and send it to the Cashier, whose response contains the name of the related Merchant App used for user consent. Although *Cashier4* validates the Merchant App in mobile payment (Sect. 2.2), all its credentials work for websites, enabling the approach above. We automate this approach and recognize 1,590 apps, while the other credentials in *Cashier1* and *Cashier3* support mobile payment only and require manual efforts to find their owners.

Parsing Client Certificates. Since *Cashier2* checks the origin of payment requests, the first approach does not work for it. Fortunately, over 40% of the payment keys in *Cashier2* are leaked along with the associated client certificates (Sect. 2.2) in git repositories, which contain the information of related Merchant Apps. By parsing the client certificates, we have identified 2,812 apps.

6.4 Longitudinal Study

After our initial testing, we reported the result to the related Cashiers. Specifically, we submitted 624 and 2,728 unique payment keys to *Cashier1* and *Cashier2* separately (in Table 8), which were got from 4,380 payment keys within 3,662 GitHub repositories after removing the duplicate keys. According to the Cashiers, they would notify all the affected Merchants and urge them to update the keys. Meanwhile, we regularly monitor and validate the submitted keys with the online validator (Sect. 5.4) to study the reactions from the leaking Merchants. Surprisingly, less than 20% of these keys have been updated 12 months after our report. Table 8 summarizes the responses from these Merchants as follows.

– **Updating the Leaked Key.** These Merchants updated the leaked keys.
– **Hiding the GitHub Repository.** These Merchants set their GitHub repositories private without updating their keys.
– **Deleting Related Git Commits.** These Merchants removed the git commits that contain the leaked keys instead of revoking their keys.

Table 8. Responses from leaking Merchant Apps after our report

Cashier	*Cashier1*		*Cashier2*	
Fixing Methods\Retesting Time	3 months later	12 months later	3 months later	12 months later
#Updating the Leaked Key	2 (0.3%)	255 (35.5%)	337 (9.2%)	443 (12.1%)
#Hiding the GitHub Repo	127 (17.7%)	146 (20.3%)	377 (10.3%)	651 (17.8%)
#Deleting Related Git Commits	117 (16.3%)	65 (9.1%)	218 (6.0%)	198 (5.4%)
#Pushing New Git Commits	8 (1.1%)	3 (0.4%)	29 (0.8%)	24 (0.7%)
#No Response	464 (64.6%)	249 (34.7%)	2701 (73.8%)	2346 (64.1%)
#Detected Key (#Unique Key)	718 (624)		3662 (2728)	

- **Pushing New Git Commits.** These Merchants deleted their keys from the code and pushed new commits online. However, the leaked keys still existed in git history and were valid.
- **No Response.** These Merchants made no response.

As we can see, around 60% of the leaking apps took no reactions after being notified, while the others used four different fixing approaches. Among them, only updating the leaked key is correct because the data leak issue is by nature irreversible, and the keys are still valid and available in the other cases. For example, public GitHub repositories are archived by Google BigQuery [9] such that the attacker can still recover the desired payment credentials from them. Thus, neither deleting git commits nor hiding the repository will work.

This finding shows that *many leaking Merchants have not noticed the severity of the leak issue or take the wrong approaches to fix it.* Given the severe circumstance, we give the Cashiers the following suggestions.

- The Cashiers should explicitly alarm their Merchants about the serious consequences of payment credential leaks (mentioned in Sect. 4).
- The Cashiers should review their services and timely fix the insecure implementations, *e.g.*, flawed backend SDKs (in Sect. 3.3), shared *user_id*s across services (in Sect. 4.3), and misleading demo (in Sect. 6.2).
- The Cashiers should proactively detect and revoke the leaked credentials.

6.5 Case Study

We give two representative samples of leaking Merchant Apps here.

A Tax App. This app belongs to an official tax bureau of a certain region, *with over 80 million population*, which integrates both SSO and payment services from one of the Cashiers under our study. End-users can thus pay the tax through this app. Unfortunately, we find that the app maintains its frontend project in a public GitHub repository. Even worse, the app hard-codes its credentials in the SSO module, which can also be used for the payment service. Thus, the attacker can uncover the leaked key and steal all the tax payment records from the Cashier by impersonating the real Merchant (Sect. 4.1).

A Financial App. This app is from one studied Cashier and has over 5 million downloads, which also uses its own payment service to enable the purchase of financial products by end-users. Beyond our expectations, this app leaks its payment key in APK. As the app also enables the money transferring function, the attacker may launch a notification forgery attack [25] or even steal the money from the app directly by forging a request to the Cashier (Sect. 4.1).

7 Related Work

Online Payment Vulnerabilities. As an increasing number of third-party web and mobile applications have integrated the online payment service, the security of online third-party payment has become a popular research topic in recent years. Wang *et al.* [21] are the first to systematically study logic flaws within the third-party payment services adopted by web applications. Sun *et al.* [18] take a step forward by proposing an automatic approach to detect payment logic vulnerabilities in web applications based on symbolic execution. Mulliner *et al.* [12] study the vulnerability of local signature verification in Android apps and manage to crack the in-app billing service in 60% of apps. [10,14] assess the security of digital wallet services in developing countries and find several vulnerabilities in their protocols and mobile apps. Using some NLP techniques, Chen *et al.* [5] identify several logic vulnerabilities by analyzing the documents from payment syndicators. However, [5] assumes the implementations or designs from Cashiers to be secure, which has been proven incorrect.

[25] is a closely related work in which the authors perform the analysis on third-party payment services integrated by mobile applications and discover several types of common flaws, including the payment credential leak in Android packages. They claim the leak enables the attacker to query transaction information illegally. Compared to [25], we perform a deeper analysis of practical mobile payment systems and uncover more novel attack scenarios based on the leaked credentials and additional implementation flaws by Cashiers or Merchants. Two of these exploits enable the attacker to cheat the other Merchant Apps without credential leaks to shop for free or even compromise their SSO services.

Existing Works on Credential Leaks. Viennot *et al.* [19] perform a market scale measurement study on Google Play apps with their framework PlayDrone and find a significant number of apps leaking Amazon Web Service tokens and OAuth tokens with simple pattern matching. CredMinder [26] also targets credential leak detection in Android apps, but it employs code analysis rather than string matching and can effectively find credentials even when they are obfuscated. LeakScope [27] applies a similar static analysis method to detect misuse of keys for cloud services and uncovers 15,098 vulnerable apps. Wen *et al.* [22] build iCredFinder to fill the gap of credential leak detection for iOS apps. All the existing works, including [25], only analyze one particular version of app packages. In contrast, our framework can detect credential leaks in old app versions, even if the issue has been fixed in the latest version. PayKeyMiner can

also detect the leaked credentials from Merchant Servers (Sect. 3.3), which has not been studied before.

Meanwhile, Meli *et al.* [11], by collecting secrets with GitHub Search API and BigQuery snapshot, show that secret leak in GitHub repositories is also pervasive. Although we also study the leak issue from GitHub, PayKeyMiner performs a thorough analysis of the suspected repositories and considers the leaks in history (in Table 6), while [11] only assesses the files in GitHub search results. Besides, PayKeyMiner circumvents the query quota from GitHub [8] and thus covers almost all searchable repositories, which is beyond the scope of [11]. Moreover, this work manually validates the leaked credentials and fails to resolve related apps, while our tool automates these processes. Compared to [11], we also introduce new exploits with credential leaks and evaluate their impacts.

There are other open-source tools for leak detection in git repositories. For example, truffleHog [3] and Gitleaks [15] can detect credentials in a given git repository, while shhgit [7] monitors the real-time leaks based on GitHub public events API. These tools assume strong patterns or randomness in secret strings and have no validation phase, which will lead to an unacceptably high false-positive rate in our case as some payment credentials are generated by the developers and do not follow strict patterns (in Table 1). Thus, we combine coarse file searching, fine credential digging, and zero-impact online verification to ensure the broad coverage as well as high accuracy of our result.

8 Conclusion

In this paper, we perform an empirical study for mobile payment credential leaks. By studying the mobile payment services from four top-tiered Cashiers, we identify new leaking sources of payment credentials and discover four types of exploits caused by the credential leaks with severe consequences. Besides, we propose an automated tool, PayKeyMiner, to conduct large-scale testing for the leaked payment credentials in the wild. We implement the tool and use it to detect around 20,000 payment credentials that affect thousands of apps. Our study shows that the overall security quality of payment credentials seems worrisome. We hope that the Cashiers review their payment services, conduct pro-active scanning, and revoke the leaked payment credentials timely to protect their Merchants and the end-users behind.

Responsible Disclosure

We have reported all our findings to the Cashiers under our study and got their confirmation and acknowledgements. Many of the affected Merchants have changed their leaked payment credentials upon notifications by the Cashiers.

Acknowledgements. This research is supported in part by the CUHK Project Impact Enhancement Fund (Project# 3133292), the CUHK Direct Grant #4055155, and the CUHK MobiTeC R&D Fund.

References

1. Anzhi: Anzhi App Market (2021). http://www.anzhi.com
2. Apkpure: Apkpure App Market (2021). https://apkpure.com
3. Ayrey, D.: trufflehog (2021). https://github.com/dxa4481/truffleHog
4. Chen, T., Guestrin, C.: Xgboost: a scalable tree boosting system. In: ACM SIGKDD 2016 (2016)
5. Chen, Y., et al.: Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In: USENIX Security 2019 (2019)
6. Dong, S., et al.: Understanding android obfuscation techniques: a large-scale investigation in the wild. In: EAI SecureComm 2018 (2018)
7. eth0izzle: shhgit: find github secrets in real time (2021). https://github.com/eth0izzle/shhgit
8. GitHub: Github Search API (2021). https://developer.github.com/v3/search
9. Google: Google BigQuery (2021). https://cloud.google.com/bigquery
10. Kumar, R., Kishore, S., Lu, H., Prakash, A.: Security analysis of unified payments interface and payment apps in India. In: USENIX Security 2020 (2020)
11. Meli, M., McNiece, M.R., Reaves, B.: How bad can it git? characterizing secret leakage in public github repositories. In: NDSS 2019 (2019)
12. Mulliner, C., Robertson, W., Kirda, E.: Virtualswindle: an automated attack against in-app billing on android. In: ACM ASIACCS 2014 (2014)
13. Openwall: John the Ripper (2021). https://www.openwall.com/john
14. Reaves, B., Scaife, N., Bates, A., Traynor, P., Butler, K.R.: Mo(bile) money, mo(bile) problems: analysis of branchless banking applications in the developing world. In: USENIX Security 2015 (2015)
15. Rice, Z.: Gitleaks: Audit git repos for secrets (2021). https://github.com/zricethezav/gitleaks
16. Savvy, M.: Amazing stats demonstrating the unstoppable rise of mobile payments globally (2020). https://www.merchantsavvy.co.uk/mobile-payment-stats-trends
17. Shi, S., Wang, X., Lau, W.C.: MoSSOT: an automated blackbox tester for single sign-on vulnerabilities in mobile applications. In: ACM ASIACCS 2019 (2019)
18. Sun, F., Xu, L., Su, Z.: Detecting logic vulnerabilities in e-commerce applications. In: NDSS 2014 (2014)
19. Viennot, N., Garcia, E., Nieh, J.: A measurement study of google play categories and subject descriptors. In: ACM SIGMETRICS 2014 (2014)
20. Wandoujia: Wandoujia App Market (2021). https://www.wandoujia.com
21. Wang, R., Chen, S., Wang, X.F., Qadeer, S.: How to shop for free online security analysis of cashier-as-a-service based web stores. In: IEEE S&P 2011 (2011)
22. Wen, H., Li, J., Zhang, Y., Gu, D.: An empirical study of SDK credential misuse in iOS apps. In: APSEC 2018 (2018)
23. Wikipedia: Client Certificate (2021). https://en.wikipedia.org/wiki/client_certificate
24. Yang, R., Lau, W.C., Shi, S.: Breaking and fixing mobile app authentication with OAuth2.0-based protocols. In: ACNS 2017 (2017)
25. Yang, W., et al.: Show me the money! finding flawed implementations of third-party in-app payment in android apps. In: NDSS 2017 (2017)
26. Zhou, Y., Wu, L., Wang, Z., Jiang, X.: Harvesting developer credentials in android apps. In: ACM WiSec 2015 (2015)
27. Zuo, C., Lin, Z., Zhang, Y.: Why does your data leak? uncovering the data leakage in cloud from mobile apps. In: IEEE S&P 2018 (2018)