Mining and Exploiting (Mobile) Payment Credential Leaks in the Wild

Shangcheng Shi, Xianbo Wang, Wing Cheong Lau

The Chinese University of Hong Kong

Abstract

Over the past decade, an increasing number of mobile apps have integrated the third-party payment function from service providers, or so-called Cashiers. Thus, end-users can perform the payment within the smartphone through these Cashiers readily. To secure their services, the Cashiers define various payment credentials, e.g., PKCS#12 certificates, and share them with mobile apps for authentication and authorization operations, such as refund. Despite the security-critical nature of these payment credentials, the existing works focus on the specific credential leaks from known sources, e.g., Android APKs or GitHub. In contrast, little effort has been spent to study the prevalence of payment credential leaks in the wild and their security impacts. In this white paper, we begin by giving the background of the mobile payment service from four first-tier Cashiers that serve over 1 billion users globally. After that, we introduce the potential leaking sources of the payment credentials, including the new ones that have not been investigated on a large scale before. For example, we find that the backend servers of mobile apps can expose payment credentials to the public inadvertently, which is caused by the insecure SDKs from the Cashiers. Then, we describe four exploits enabled by the payment credential leaks when combining other implementation flaws. These exploits all bring about serious consequences, ranging from direct financial loss to the mobile apps and privacy violations to end-users. Specifically, with the leaked payment credentials, the attacker may steal money from the account of the mobile apps directly and obtain all the user payment records. Further, we design and implement an automatic tool to conduct large-scale mining for payment credential leaks. Consequently, we manage to discover around 20,000 leaked payment credentials, which affect thousands of mobile apps and millions of end-users. Finally, we give some suggestions to Cashiers as the mitigations.

1. Background

A typical online payment service involves three parties, *i.e.*, the User (or User-Agent), the Cashier, and the Merchant. As to the mobile payment service, 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 parentheses in the rest of this paper.

The target of mobile payment is to convince the *MS* that the User has paid the order with the balance in his Cashier account. Despite its widespread usage, there is no unified standard for mobile payment, and the services from the Cashiers are diverse. Specifically, their workflows and definitions of payment credentials differ, so we will briefly discuss them here.

1.1. Workflow of Mobile Payment Service

After reviewing the documents from four first-tier Cashiers[1], we summarize a general workflow for mobile payment and present it in Figure 1. The step-to-step illustration is given as follows.

1) The user needs to log into the *MA*, which may be conducted through the SSO service from a third-party Cashier.
2) After choosing products in an *MA*, the user selects a Cashier to check out. Then, the app sends a request containing the ordering details to *MS*.
3) The *MS* generates a payment order and feeds it back to the *MA*.
4) The *MA* passes the payment order to the *CA*, which then displays the details, *i.e.*, trade_amount and payee.
5) Once the user confirms the payment, the *CA* sends a payment request to its server (*CS*).
6) The *CS* processes the request. Afterward, the Cashier notifies the *MS* (through the backURL specified in the payment order in Step 3), *i.e.*, asynchronous notification, and the *CA*, *i.e.*, synchronous notification, about the payment by the user.
7) The *CA* returns synchronous notification to the *MA*.

[1] We use *Cashier1* to *Cashier4* to denote these four studied Cashiers because they require us to anonymize their names in their response to our responsible disclosure.

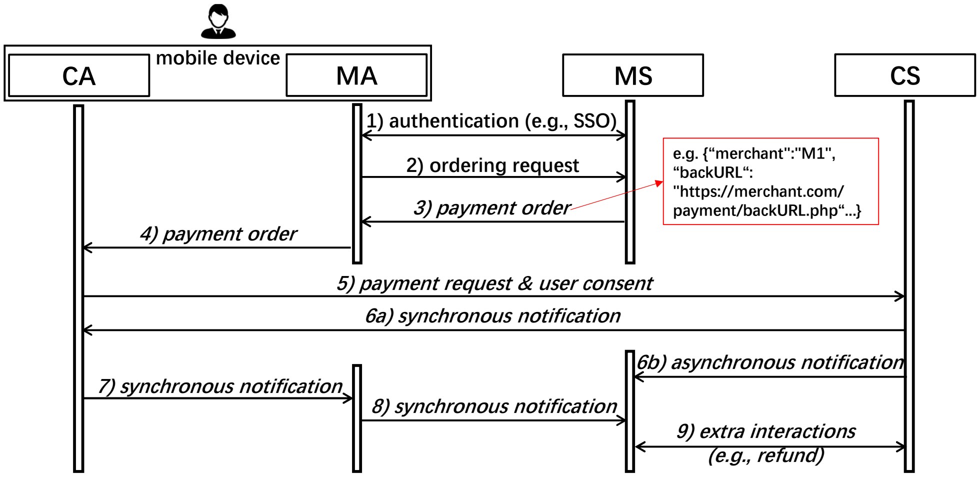
mobile device
CA
MA
MS
CS
1) authentication (e.g., SSO)
2) ordering request
3) payment order
e.g. {“merchant”:"M1", “backURL“: "https://merchant.com/payment/backURL.php“...}
4) payment order
5) payment request & user consent
6a) synchronous notification
6b) asynchronous notification
7) synchronous notification
8) synchronous notification
9) extra interactions (e.g., refund)

Figure 1. Workflow of Mobile Payment Service

8) The *MA* feeds the received notification back to its server and queries the payment status of the order.
9) The *MS* may request the Cashier, *i.e.*, *CS*, directly for privileged operations, *e.g.*, refund.

1.2. Payment Credential Overview

1.2.1. Payment Key. Most steps in Figure 1, in italic, are secured by either a Hash-based Message Authentication Code (HMAC) or digital signature. Thus, Cashiers and apps need to share some payment keys beforehand. Table 1 summarizes the various payment keys defined by the Cashiers, which can be categorized into the following two types.

Secret Key. All the Cashiers under study support HMAC to secure payment-related messages. Once adopted, both the *MS* and *CS* will use the secret key as the salt of a hash function to generate the HMAC. Apart from *Cashier2*, this type of key is generated by Cashiers.

Signing Key. Other credentials in Table 1 belong to this type and are used to generate the digital signatures of network messages during mobile payment. For the actual production, both the *MS* and *CS* need to maintain a pair of RSA keys (or the equivalent) and share their public keys. In runtime, either party will sign the request with its own private key and verify the response with the other's public key.

As *Cashier1* supports two methods to generate the digital signature, it defines two types of payment keys, *i.e.*, RSA key and RSA' key. Another crucial difference between the two is that *Cashier1 shares its public key across all the apps in RSA keys, which is instead app-specific in RSA' keys.*

In comparison, *Cashier3* assigns the (Merchant) apps PFX certificates to secure outgoing requests. These certificates are PKCS#12 files issued by a specific Certificate Authority and protected by user-defined passwords. As such, the *MS* needs to unlock the file to extract the private key for generating the signature. Similar to the RSA key in *Cashier1*, *the public key of Cashier3 is shared among all its apps and is included in the official backend SDK.*

TABLE 1. SUMMARY OF PAYMENT KEYS OR THE EQUIVALENT

Cashier	Payment Key	Usage	Assigned by the Cashier?	Shared Cashier's Public Key?
Cashier1	Secret Key	HMAC	✓	N/A
	RSA Key	Digital Signature	×	✓
	RSA' Key	Digital Signature	×	×
Cashier2	Secret Key	HMAC	×	N/A
Cashier3	Secret Key	HMAC	✓	N/A
	PFX Cert	Digital Signature	✓	✓
Cashier4	Secret Key	HMAC	✓	N/A

1.2.2. Other Payment Credentials. Some Cashiers define other credentials to enhance the security of their payment services.

Android Signing Key. The mobile apps of *Cashier2* and *Cashier4* will validate the integrity of the *MA* when receiving the payment order, *i.e.*, Step 4 in Figure 1, by checking its package signature. Thus, the app developers should also keep the associated Android signing keys, in the form of .jks or .keystore files, private.

Client Certificate [1]. *Cashier2* issues a per-app-based certificate for each app to support its service. This file is in the format of PKCS#12 and password-protected. Then, the *MS* needs to unlock and present it in each critical API call, *e.g.*, refunding requests (Step 9 in Figure 1), to the *CS* for the authentication purpose.

2. Payment Credential Leaks

Here, we talk about the potential leaking sources of payment credentials, among which public GitLab repositories and Merchant Servers, *i.e.*, *MS*s, have never been discovered before.

2.1. Public Git Repositories

As discovered by [2], some developers push the production code to GitHub without removing sensitive credentials. Even if the developers have noticed the leaks, they may not take the correct fix, namely, updating the credentials to hide the leaked credentials. Thus, the attackers can steal them from the public repositories.

On the other hand, we find that some companies establish GitLab services on public IPs and make their repositories public, which may also contain payment credentials. The companies assume that others cannot find and access these public repositories. However, the powerful web crawlers from search engines like Google break the assumption. Consequently, the attacker can easily identify and download these GitLab repositories and further extract payment credentials from them.

2.2. Mobile Apps

Many developers deploy some server-side operations in their mobile apps, which thus embed credentials [3]. Then, the attacker can decompile the *MA* to get the desired credentials. Since the iOS ecosystem is proprietary, most of the app packages are encrypted and available from the official app store only so that they are not suitable for large-scale testing. Therefore, our work mainly focuses on Android apps. Nevertheless, we still consider the leaks within the iOS projects from public git repositories, which contain the whole frontend source codes.

2.3. Merchant Servers

We surprisingly discover that the servers of (Merchant) apps, *i.e.*, *MS*s, can be another source of payment credential leaks. This is caused by the insecure design of Cashiers' backend SDKs and the lack of protection on the associated credential files by Merchant Servers.

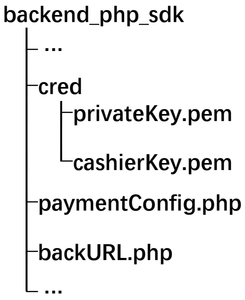

Figure 2. Structure of an Official Backend SDK

Cashiers provide SDKs to facilitate the deployment of payment service. Figure 2 shows the structure of an official SDK from a Cashier with over 300 million daily active users. For this SDK, the "backURL.php" file processes the asynchronous payment notifications from the *CS* (Step 6b in Figure 1). Besides, the "paymentConfig.php" file specifies the location of the Merchant's signing key (Section 1.2.1) to a static file, namely "cred/privateKey.pem", where the document also requires the developer to store his private key. Consequently, the insecure design enables the attacker to retrieve the key file from Merchant Servers by applying the following trick.

- The location of "backURL.php", *i.e.*, the value of backURL in the payment order (Step 3 of Figure 1), is visible to the attacker-controlled handset. Based on the SDK structure in Figure 2, the attacker can then infer the location of the Merchant's private key, namely "privateKey.pem".
- If the Merchant Server (*i.e.*, *MS*) does not block the access to this private key file, the attacker can then directly retrieve this file. Notably, this Cashier does not give such a warning in its document to protect this private key file.

On the other hand, some official SDKs include payment credentials in scripts or other inaccessible files such that the attacker cannot steal them in most cases, while the other SDKs can still suffer the leakage issue. We automate the detection of the kind of payment credential leaks in Merchant Servers (by the URL Enumerator in Figure 4). The result turns out that 7.11% of the tested servers fail to protect their payment credentials.

3. Exploiting Payment Crednedential Leaks

This section discusses four types of exploits enabled by payment credential leaks. These exploits can cause direct financial loss to the leaking app and its end-users. Since some of the studied Cashiers provide SSO service, it is also possible to crack the authentication mechanism of the app.

3.1. Merchant Impersonation Attack

With the leaked payment credentials, the attacker can forge requests to the Cashiers as benign Merchants for conducting privileged operations, *i.e.*, refunding and money transferring.

3.1.1. Refunding. The operation enables the attacker to shop for free. The attacker may first purchase merchandise in the target app as a normal user. Then, he needs to obtain the related trade_num to forge the refunding request, which is always available from the transaction record from the *CS*. Besides, the attacker may even refund the other orders related to normal users to hide his identity.

The exploit is feasible for all the leaking apps in *Cashier1*, *Cashier3*, and *Cashier4*, where we have conducted the proof of concept experiments with official demo accounts or under sandbox environments. Although *Cashier2* requires the client certificate in the function call (Section 1.2.2), our mining tool detects over 3,000 certificates in the wild, which can also be uncovered by the attacker (in Table 2).

3.1.2. Money Transferring. Except for *Cashier3*, all the Cashiers under our study provide the service such that the (Merchant) app can transfer the money from its own Cashier account to an arbitrary user account. Once the leaking app activates the service, the attacker can forge a request to steal the money directly. With the services like [4], he may exchange the stolen money into bitcoin and move it abroad.

On the other hand, although *Cashier2* asks for the client certificate in the interface, these certificates can also be leaked (in Table 2). Besides, *Cashier2* requires the SSO user_id of the payee in the request, which is app-specific and unknown to end-users. However, its value appears in the transaction record such that the attacker can pay for a forged order to get it.

3.2. Android Package Signature Forgery

As stated in Section 1.2.2, *Cashier2* and *Cashier4* authenticate the *MA* in each payment. However, many Android signing keys are leaked in public git repositories (in Table 2) as the developers push the whole frontend project online. Thus, the attacker can modify several lines of code and forge malicious apps with valid signatures. Then, the malicious apps can trick the users into paying for the attacker's order unintentionally by replacing the payment order (Step 4 in Figure 1).

Using the leaked Android signing keys, we have managed to package apps that can pretend to be the benign one to the *CA*s in *Cashier2* and *Cashier4* and bypass their verification. Overall, our mining tool detects 493 such leaks (in Table 2). The statistics from third-party app markets show that *10 of the related apps have more than one million downloads*.

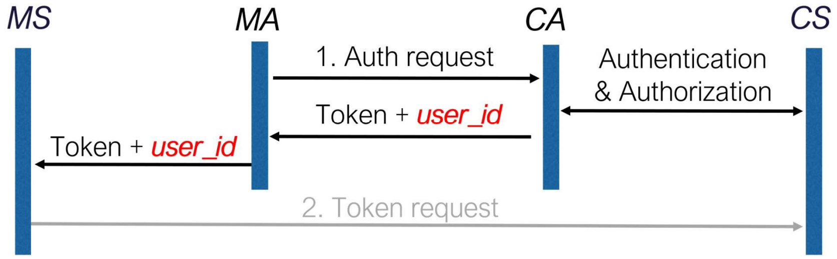

Figure 3. Workflow of Mobile SSO Services from the Cashiers

3.3. Backward SSO Attack

Some of the Cashiers, *e.g.*, *Cashier2*, provide authentication service via SSO. We find some flaws in their systems such that the credential leaks in the payment service can reversely affect the authentication mechanism of the apps.

Specifically, these Cashiers use the same set of user_ids for both payment and SSO services, which are available to the attacker once their payment credentials are leaked. Although the protocol has been proven secure [5] by design, some apps do not follow the standard workflow of OAuth [6] and rely on the user_id from the client-side (in Figure 3) to authenticate the user instead of querying the *CS* in Step 2. Consequently, these apps become vulnerable to Profile Attack [7], meaning that the attacker can replace the user_id that goes through his smartphone to hijack the accounts of victim users.

Besides, we find that some developers reuse the values of their payment keys to be the SSO secret, which amplifies the impact of payment credential leaks to SSO. According to our test, 2% of the payment keys also work as the SSO secret.

3.4. Cross-App Notification Forgery

Leveraging some poor implementations by both the Cashiers and apps, the attacker can use the leaked payment credentials to compromise the payment service of other apps.

As stated in Section 1.2.1, the public key of the Cashier in *Cashier1* and *Cashier3* can be shared among apps (in Table 1). Thus, the attacker can elaborately forge some payment notifications from the *CS* with the leaked payment credentials and inject them into the session of another app, which is cryptographically-consistent but logically-incorrect. However, the *MS* may fail to check the app identifier inside, *i.e.*, whether the message is sent to itself or not. Then, the app will be cheated and enable the attacker to shop for free.

Due to ethical considerations, we cannot fully quantify the exact impact of the exploit without attacking real Merchants. Nevertheless, we have identified an open-source framework for *MS*s to be vulnerable, which integrates the payment services from the studied Cashiers and has over 100,000 downloads. Towards this end, we set up our own Merchant with this framework and complete a Proof-of-Concept (PoC) experiment on it.

4. Automatic Mining Tool for Leaked Payment Credentials

To support large-scale mining for payment credential leaks, we develop an automatic tool. The framework of our tool is given in Figure 4, which consists of three modules. The Crawler first identifies the public git repositories and Android APKs that are likely to support the payment function from the public data sources. Then, the Scanner analyzes the filtered input and recognizes all the potential credentials. Notably, relying on the backURLs embedded in Android APKs, the URL Enumerator inside the Scanner probes the *MS*s to detect the exposed credential files (mentioned in Section 2.3). Finally, the Detector processes the suspected credentials and verifies their validity.

5. Empirical Testing

5.1. Dataset

Our mining tool takes public GitHub/ GitLab repositories and Android APKs as the input dataset. Overall, we clone 139,206 GitHub repositories that integrate the payment services from the Cashiers. Meanwhile, we collect 943 GitLab

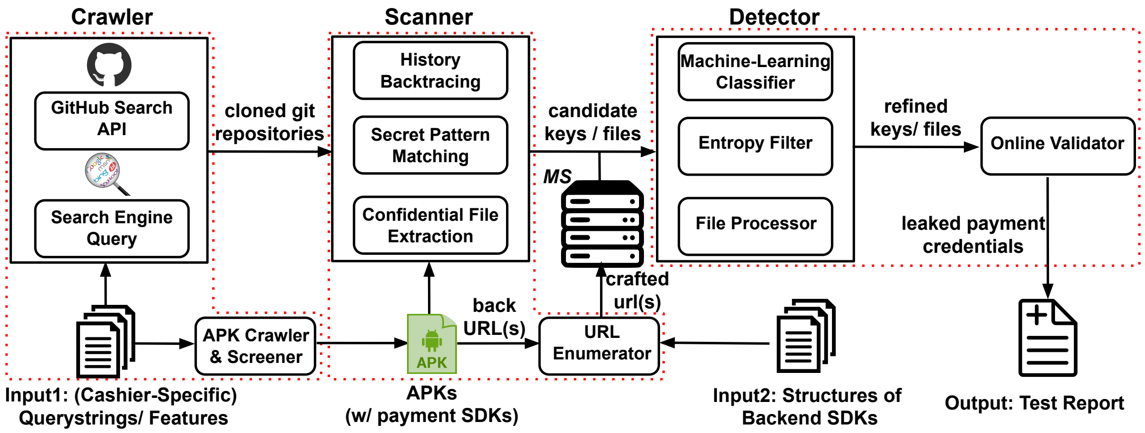

Figure 4. System Architecture of Our Automatic Mining Tool

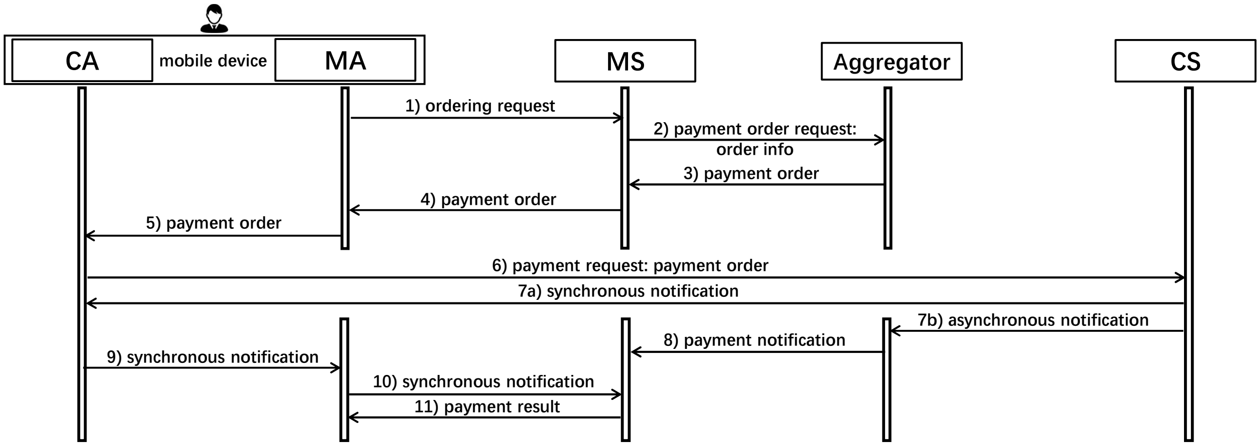
CA
mobile device
MA
MS
Aggregator
CS
1) ordering request
2) payment order request:
order info
3) payment order
4) payment order
5) payment order
6) payment request: payment order
7a) synchronous notification
7b) asynchronous notification
8) payment notification
9) synchronous notification
10) synchronous notification
11) payment result

Figure 5. Workflow of Payment Aggregator

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), from three first-tier app markets in September 2019.

5.2. Test Results

Table 2 shows the test results, where our tool has detected around 20,000 unique payment credentials. Specifically, 10.34% of public git repositories, 3.21% of Android APKs, and 7.11% of the *MS*s leak payment credentials. The average running time for each input is around 300 seconds.

TABLE 2. SUMMARY OF THE LEAKED PAYMENT CREDENTIALS

Cashier	*Cashier1*			*Cashier2*			*Cashier3*		*Cashier4*	
Source\Credential	Secret Key	RSA Key	RSA' Key	Secret Key	Client Cert	Android Key	Secret Key	PFX Cert	Secret Key	Android Key
GitHub Repo	900	1518	1737	6651	3131	491	0	188	25	1
GitLab Repo	9	20	20	57	31	1	0	1	0	0
Android APK	75	1950	354	2567	3	0	2	0	10	10
Merchant Server	0	44	0	0	11	0	0	2	0	0
Overall	975	3332	2085	9093	3170	492	2	189	34	1

* The table has removed the duplicate credentials in each row.
†The overlapping among different data sources is removed.

5.2.1. Git Repositories. Overall, 14,493 public git repositories leak 23,011 payment credentials before removing duplicate credentials. Among the results, 1,792 credentials (7.79%) only appear in the history, indicating that *some developers have noticed the leak issue but used the wrong method, i.e., pushing new commits, to fix it.* On average, it takes developers about 51 days to make the wrong fix. In contrast, the others exist in the latest code and are available from the search result [8].

Notably, as mentioned in Section 2.2, we also search for payment credential leaks in iOS-related git repositories. From the collected input, our mining tool has identified 365 and 347 unique payment keys in *Cashier1* and *Cashier2* separately.

Meanwhile, the four search engines perform differently in detecting public GitLab repositories, where Google contributes most of the results. Besides, we find that most of these repositories are owned by some *outsourcing companies*.

5.2.2. Android APKs. Our mining tool detects 9,008 payment keys (including 4,958 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.

There are two other interesting findings when we study the result from Android APKs. First, we study the leaking locations of these payment credentials and find that around 2,000 keys are from the same location or Android activity. This activity actually belongs to the official frontend demo project provided by one Cashier. Although this Cashier claims that the project is only for demo use and gives a strong warning about the leak issue in this demo. Many Merchant Apps still reuse it for their own payment service.

Meanwhile, we find one certain payment key appearing in hundreds of Merchant Apps, which belongs to a payment aggregator. Figure 5 presents the workflow of the payment aggregator, which works as the proxy between the real Merchant Server and Cashier Server. However, this aggregator leaks its key in the frontend SDK and affects all the related apps.

5.2.3. Merchant Servers. URL Enumerator (in Figure 4) has identified 802 backURLs and found 57 credentials. Notably, some of the results are only available from this leaking source. For example, 38 RSA keys (in *Cashier1*) do not appear elsewhere. As many apps do not embed backURLs, our tool cannot obtain the values from APKs directly. Nevertheless, the attacker may shop normally in the app and intercept payment order, *i.e.*, Step 4 in Figure 1, to extract it. Moreover, iOS apps can suffer the same issue, and we have manually found an example from the collected GitHub repositories.

6. App Identity Resolution

To exploit the leaked credential, we need to get the identity of the associated Merchant App. Notably, some payment credentials are extracted from the backend code in git repositories so that we cannot know the related apps directly. Therefore, we resort to the Cashiers for help, which host all the app information. In short, we use the following three methods to get app identities.

6.1. Crafting Payment Request

In most cases, the Cashiers do not force the verification of the *MA*. Consequently, right before the payment (between Step 4 and 5 in Figure 1), the *CA* will extract app information, *e.g.*, name and logo, from *CS* and present it to the user for consent. Thus, we can craft payment requests by ourselves with the detected keys, send them to the *CA*, and rely on the returned information to resolve the app identities. We automate the approach in Web App Payment and recognize 1,590 apps. The remaining credentials only support in-app payment and need manual efforts to identify their owners.

6.2. Parsing Client Certificates

Cashier2 performs strict verifications on the *MA*, such that the first method is not applicable. Fortunately, over 40% of the payment keys in *Cashier2* are leaked along with the associated client certificates (Section 1.2.2) in git repositories. Meanwhile, it is *Cashier2* that issues the certificate, which includes the desired app information. We have identified 2,812 apps by parsing their client certificates.

6.3. Hooking the Cashier App

In *Cashier2*, the *CA* verifies the integrity of *MA* by checking its signature before processing the incoming payment order, *i.e.*, Step 4 in Figure 1. This process is called app registration and is initiated by *MA* through invoking *Cashier2*'s frontend SDK, with its app identifier as the input. The process inside the *CA* involves two steps: 1) *getAppInfo*: the *CA* exchanges the app identifier (from the payment order) for the app information from its server. 2) *verifyAppInfo*: the *CA* compares the package name and signature locally and refuses to proceed upon a check failure.

Based on the observation, the attacker can fetch the app information from the *CS* with solely the app identifier. Although the network protocol of *Cashier2*, *i.e.*, the interactions between the *CA* and *CS*, is proprietary, the attacker can take the hooking approach as a bypass. Firstly, he can reverse engineer the *CA* to locate the registration function (*i.e.*, *getAppInfo*). Then, he can dynamically hook the function, inject the suspected app identifier, and intercept the response message for resolving the app identity. We have developed the proof of concept code for the approach above.

7. Responsible Disclosure and Mitigations

We have reported the detected credentials to all the four studied Cashiers and got their confirmations. According to the response, the Cashiers will send a warning to the leaking apps. Unfortunately, 12 months after our first report (submitted in September 2019), we find that less than 20% of these leaking apps have revoked their credentials, while the other leaking apps take various wrong fixes without updating the leaked payment credentials. For example, some of them choose to remove the GitHub repositories with leaks, which can actually be archived by some online service, *e.g.*, Google BigQuery [9]. Consequently, it is still feasible for the attacker to steal the leaked payment credentials.

Considering the severe consequence of payment credential leaks discussed in Section. 3, we give the following suggestions for the mitigations.

- The Cashiers should explicitly alarm developers about the serious consequences of payment credential leaks.

- The Cashiers should review their services and timely fix the insecure implementations, *e.g.*, flawed backend SDKs (in Section 2.3) and shared user_ids across services (in Section 3.3).
- The Cashiers should proactively monitor and revoke the leaked payment credentials to mitigate the potential exploits (mentioned in Section 3).
- The Merchants should periodically change their payment credentials.

8. Conclusion

In this white paper, we perform an empirical study on mobile payment credential leaks. Specifically, we study the service from four first-tier Cashiers and identify new leaking sources of their payment credentials. To present the impacts of the leak issue, we construct 4 practical attacks with severe security consequences, which allow the attacker to cause direct financial loss or privacy violation to either mobile apps or their end-users. We also develop an automatic tool to conduct large-scale testing, leading to the discovery of around 20,000 leaked payment credentials that affect thousands of apps and millions of users.

References

[1] Wikipedia, “Client certificate,” https://en.wikipedia.org/wiki/Client_certificate, 2020.

[2] M. Meli, M. R. McNiece, and B. Reaves, “How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories,” *Proceedings 2019 Network and Distributed System Security Symposium*, no. February, 2019.

[3] H. Wen, J. Li, Y. Zhang, and D. Gu, “An Empirical Study of SDK Credential Misuse in iOS Apps,” *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2018-Decem, pp. 258–267, 2019.

[4] LocalBitcoins, “Localbitcoins,” https://localbitcoins.com, 2019.

[5] M. Backes and D. Hofheinz, “How to break and repair a universally composable signature functionality,” in *Information Security, 7th International Conference, ISC 2004, Palo Alto, CA, USA, September 27-29, 2004, Proceedings*, 2004, pp. 61–72.

[6] D. Hardt, “The OAuth 2.0 authorization framework,” 2012.

[7] R. Yang, W. C. Lau, and S. Shi, “Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols,” in *ACNS*, 2017.

[8] GitHub. (2019) Github search api. [Online]. Available: https://developer.github.com/v3/search

[9] Google, “Google BigQuery,” https://cloud.google.com/bigquery/, 2019.