An Empirical Study of SDK Credential Misuse in iOS Apps

Haohuang Wen
School of Software Engineering
South China University of Technology
Guangzhou, China
onehouwong@gmail.com

Juanru Li
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
Shanghai, China
jarod@sjtu.edu.cn

Yuanyuan Zhang, Dawu Gu
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
Shanghai, China
{yyjess, dwgu}@sjtu.edu.cn

***Abstract*—During the development of web-based mobile apps, third-party SDKs (Software Development Kit) are frequently used to facilitate the integration of certain functionality such as push notification and mobile payment. Unfortunately, security issues are often considered as a second-tier problem and app developers are prone to implement apps with SDK misuses. Among those typical SDK misuses, the misuse of credentials is the one that introduces serious security threats. A credential is a set of unique information (e.g., APP ID, App Token, etc) allocated to a specific developer to help app authenticate the identity. However, if not properly used, the credential can be easily obtained by attackers and leads to not only the leak of confidential information of mobile developers but also direct threats to the privacy of end users.**

To investigate the SDK credential misuse issue on iOS platform, in this paper we conduct an empirical study against 100 popular iOS apps using two popular mobile SDKs (each SDK are widely used by at least 40 million users). We implemented iCredFinder , an automated analysis tool to search credential misuses in those apps and our experiment demonstrates 68 apps contain at least one misuse case. Our study demonstrates the severity of credential misuse on iOS platform: even for those well-developed SDKs and apps, credentials are not well protected and can be easily discovered. We expect that our study could help developers fix those flaws and promote better implementations.

***Index Terms*—iOS apps, Third-party SDKs, Binary code analysis, Credential exposure**

I. Introduction

Mobile SDKs are widely adopted by app developers so that they can easily build a variety of mobile apps for modern Android and iOS devices (e.g., smart phones and tablets). Among various kinds of features provided by different SDKs, one typical feature is to help apps interact with remote APIs related to back end services and information on servers. Since one same service is often provided to different apps, a remote API needs to authenticate the identity of guest apps. The solution for most mobile SDKs is to use credentials to distinguish different apps while keeping the same copy of the code. A credential is often a set of unique information (e.g., APP ID, App Token, etc) allocated to a specific developer when she registers herself to the web service provider. When the app uses certain web services, it is required to provide such credentials to express its identity. Otherwise, the request is blocked.

Since credentials are often the only authentication information for many web services, mobile developers need to properly manage them and should be extra vigilant about credential security. Unfortunately, the use of mobile SDKs often weakens this assumption. For one thing, SDK providers often publish vague instructions on how to use credentials, leading to mistakenly embedded and protected credentials. For another, even if a correct guide of credential management is published, it involves many aspects of protection and is often very complex. Developers still face various challenges in implementing a secure protection scheme. As a result, many flaws related to SDK credentials have been discovered in mobile apps.

Previous researches [1] proposed to uncover mobile SDK credential issues mainly focus on Android ecosystem because Android platform is open-source and the binary code analysis for bytecode of Android app is well-developed. The code of Android apps are mostly written in Java and then compiled into Dalvik bytecode. By leveraging typical program analysis techniques such as data flow analysis and program slicing, credentials used in programs can be located at the level of bytecode. Unfortunately, when the analysis targets turn into iOS apps, existing approaches are no longer available. First, iOS apps are compiled into native code executable to guarantee performance. Compared with bytecode binary executables of Android apps, native code executables on iOS are much more difficult to be decompiled and analyzed. The lack of relevant analysis tools and techniques hinders the automation of iOS app analysis and code audit is often conducted manually. Second, iOS apps are mainly developed in Objective-C with a complicated message dispatching mechanism. This makes it difficult to conduct an accurate control flow analysis for the program. Moreover, the executable of iOS app is often very large because it statically links all used third-party libraries. As a result, a fine-grained binary code analysis against iOS app is often time-consuming and error-prone. Finally, even for the same SDK, a provider often releases two versions for Android and iOS platform, respectively. The usage of such SDK on different platforms varies significantly and thus the experience of how to analyze credential misuse on Android cannot directly port to iOS platform. Due to those challenges, to the best of our knowledge, no systematic research on

credential misuse of iOS apps has been conducted.

In this paper, we seek to perform an empirical study of how iOS app developers misuse SDK credentials. We aim to answer two problems: 1) is it possible to automatically analyze SDK credential misuses in iOS apps; and 2) how to evaluate the found credentials in an app. To answer the first problem, we propose an automatic credential misuse detection solution for iOS apps and have implemented iCredFinder , a corresponding analysis system. To answer the second problem, we combine a static program analysis and a dynamic data validation to check the risks related to all discovered credentials. We not only statically detect credential in apps but also validate its property: is it a valid token or is it just an expired one. By leveraging the automated analysis of iCredFinder , we check a dataset of more than 20,000 iOS apps and choose 100 widely used apps integrated with two most frequently used mobile SDKs in Asia. These apps are in good maintenance state and have accumulated millions of users, which can well represent the coding practice of developers today. Among them, iCredFinder found 68 apps contain at least one credential misuse case. Individually, 62 and 29 apps have embedded valid credentials, respectively for the two SDKs. Considering that our studied SDKs are such popular ones (with over 40 million users [2]), the analysis results reveal that iOS apps are not as secure as previously imagined. Interestingly, our credential validation found 34 cases of invalid credentials besides the common credential exposure cases. Those apps work normally after we removed contained credentials, which indicates that the exposed credentials are mistakenly integrated by developers. Although those credentials are no longer used by the current version of apps, they are still kept in program code. We also consider this as a credential misuse because it may reveal some secret information of the past development.

The contributions of this paper are as follows:

- We conduct an empirical study on SDK credential misuse problem against iOS apps. To address the issue of iOS app binary code analysis, we present an automatic detection solution that is able to find SDK credential misuse in iOS apps. We also describe the prototype implementation of iCredFinder , our automated detection system.
- Focusing on two frequently used mobile SDKs, we checked 100 widely used apps with the help of iCredFinder . The results show the feasibility of our approach: we found 68 apps contains at least one credential misuse case.

II. Problem and Solution

A. SDK Credential Misuse in iOS Apps

1) Problems: Third-party SDKs are widely used to facilitate the development of iOS apps. Although these SDKs are designed as general components and are used by different apps to fulfill similar functions, the corresponding web services often require access control and authentication. Therefore, the web service often needs extra secret information (we define this as a **SDK credential**) to help authenticate the identity of the host app. A typical example of SDK credential is demonstrated in Figure 1. To authenticate itself to the web server, an app needs to provide SDK credential (Line 22 a constant string) and other necessary information (Line 20-27, including client id, redirect URL, etc).

Obviously, SDK credentials should be properly managed to avoid leakage and illegal usage. Since the SDK credential is used in a mobile device, which is generally considered as an untrustful environment, the protection of such SDK credential is essential for app developers. Unfortunately, developers tend to adopt insecure practice when integrating the SDK credentials. What's worse, documents and specifications of many SDK providers are likely to give vague examples of how the SDK credential should be used, and developers following these ambiguous instructions tend to misuse the SDK credential, leading to severe security risks. For instance, they may directly embed the credential information in the app. In the case in Figure 1, the SDK credential is hard-coded as a parameter in the construction of a dictionary and is then embedded into the argument field of an HTTP request for authentication. Since the credential is used without any protection, it can be directly extracted by any attacker using reverse engineering.

2) Attacks: A misused SDK credential may lead to serious security attacks including unauthorized use of web service, illegal access to user information and app data, etc. With a valid SDK credential of a specific mobile app, an attacker can act as a qualified developer to access certain third-party services. Take the credential exposure case in Figure 1 as an example, if the attacker first obtains the SDK credential, he can conduct a concrete attack demonstrated in Figure 2. In this attack, the attacker utilizes the obtained credential to construct an HTTP request. This request can be used to obtain an access token, which can then be used as the certificate to use third-party service (Step 1 and 2 in Figure 2). Therefore, the attacker successfully disguises as a qualified developer and is able to access any service provided for the SDK provider. With the illegally acquired identity, the attacker then requests for user information by invoking the related remote API. If he knows the uid of a target user, the attacker could construct another request to query private information of this user information (As shown in Step 3 and 4 of Figure 2). With the help of a leaked credential, confidential personal information of a specified user can be illegally accessed easily.

B. Challenges

Although similar researches on SDK credential of Android apps have been proposed, the study of such object against iOS apps is significantly different and non-trivial. An effective and accurate analysis faces the following three challenges:

- **Challenge I: Reverse engineering of iOS apps**. The ecosystem of iOS is proprietary and every iOS app in AppStore provides neither the source code nor the binary code. To analyze an iOS app an analyst has to make additional efforts to decrypt the binary executable. Note that it is infeasible to obtain decrypted binary executable

```
14  v3 = self;
15  v4 = -[SocialHandler deserializeURL:](self, "deserializeURL:", a3);
16  v5 = objc_msgSend(v4, "objectForKey:", CFSTR("code"));
17  v6 = objc_msgSend(
18         &OBJC_CLASS___NSDictionary,
19         "dictionaryWithObjectsAndKeys:",
20         CFSTR(                        ),
21         CFSTR("client_id"),
22         CFSTR("f45e                   "),
23         CFSTR("client_secret"),
24         CFSTR("authorization_code"),
25         CFSTR("grant_type"),
26         CFSTR(                        "),
27         CFSTR("redirect_uri"),
```

secret

Fig. 1: An Example of an Insecurely Embedded SDK Credentials

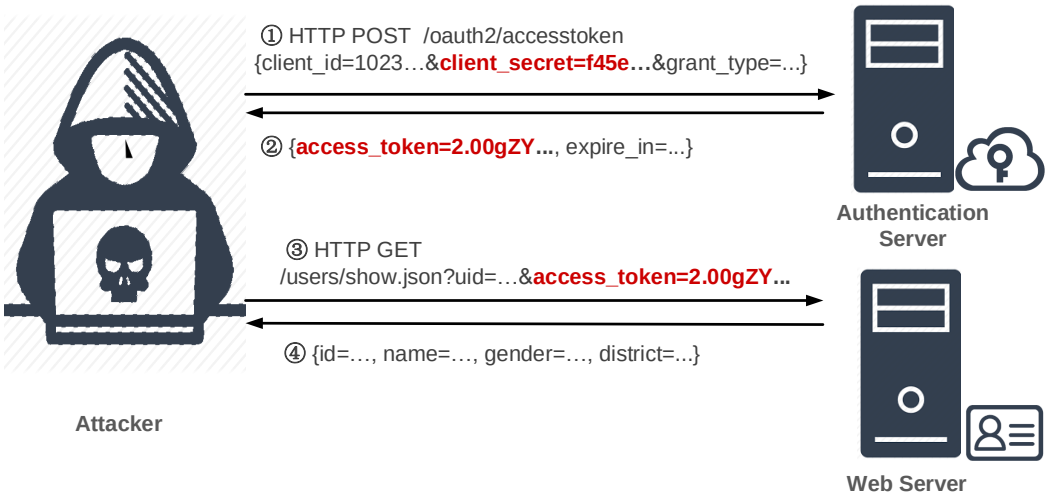


Fig. 2: How can an attacker harvest user information with a valid SDK credential

of an app without a jailbroken iOS device. In addition, binary code analysis of iOS app is much more difficult than that of Android app. Android app analysis is significantly facilitated by many well-developed bytecode decompilation utilities (e.g., JEB). In comparison, less binary code analysis tools are provided and the decompilation of Objective-C binary code in iOS apps is far from perfect, leading to the missing of much important semantics such as function parameter types and variable names.

- **Challenge II: Identifying credentials in iOS apps**: To study the credential misuse problem, the first necessary step is to identify the used credentials in the program code of an app. However, the identification of such credentials in an iOS app can learn little from the identification experiences on Android platform. The use of credential in an Android app and in an iOS app are significantly different even if these apps are developed by the same company. This is determined by different programming language styles and therefore program analysis techniques available for Android apps are not applicable when analyzing iOS apps. Besides, the authentication processes of one SDK on two platform are often not the same. Hence, new identification techniques are expected to be proposed.
- **Challenge III: Credential validation**: An interesting fact for many apps is that not all statically integrated credentials in an app can be utilized. Some of them may be expired beforehand and are left in the app by mistake. An accurate analysis should validate each credential to check whether a discovered credential is still a remaining threat. However, this often requires a dynamic verification and a solely static program analysis is not feasible to achieve such a goal.

C. Solution Overview

All these obstacles mentioned above make the existing analyzing techniques for SDK credential in Android apps fail to work against that in iOS apps. To address, we propose an automatic credential misuse detection solution for iOS apps combining with a static program analysis and a dynamic data validation. The solution adopts the following strategies:

- **Script-based iOS binary code analysis** To automate the binary code analysis of iOS apps, our solution utilizes

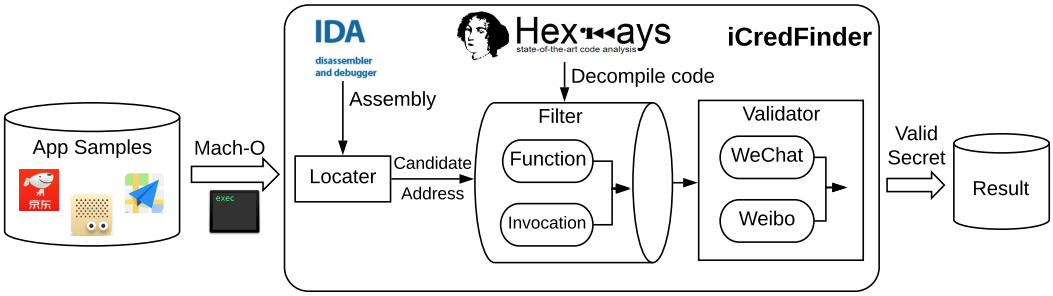


Fig. 3: Workflow of iCredFinder

IDAPython scripts to implement many functions such as automatically extracting and splitting function names, acquiring a function's address by name, etc. With the help of those analysis scripts, our analysis could deal with binary executables of commercial iOS apps with tens of thousands of functions stably and efficiently.

- **Heuristic-based credential identification**: Since a fine-grained program analysis against an iOS app is not only very time-consuming and error-prone since many iOS apps are very large (more than 100MB), our solution adopts a heuristic-based credential string searching strategy. This strategy speeds up the analysis and it also proves the potential candidates can be captured. We then supplement this identification with a dynamic validation to prove that the identification result is accurate.
- **Dynamic credential validation**: We observed that many web services released by SDK providers can be used as side channels to check the validity of a potential credential. Hence our solution can leverage these web services as remote oracles and construct certain queries to validate a credential candidate. Although such a remote oracle not directly responses the result of validity, we can deduce the needed information by observing its different replies.

We design and implement iCredFinder , a credential misuse analysis system to automate our analysis. The workflow of iCredFinder is displayed in Figure 3. To better illustrate the entire analysis process of iCredFinder , we make use of a simple case to present. Considering an iOS app that may contain misused credentials. iCredFinder first takes the Mach-O executable of this app as the initial input (the Mach-O executable is automatically extracted on a jailbroken iOS device with our developed shell script). Then, a locater of iCredFinder analyzes the assembly code and searches for credential candidates from the string sets of the executable. At the same time, the addresses of these candidates are also recorded. For instance, if a string (e.g., `b4adae3a2d91021ad33151f2ca707954`) matches the specific feature, it is chosen as a credential candidate and its corresponding address (e.g., `0x101781DE0`) is recorded. Next, the candidate is passed to a filter for further checking. The filter extracts two features of the credential from the decompiled code of the executable, including the function name (e.g., *-[AppDelegate registerShareSDK]*) and the invocation (e.g., *SSDKSetupWeChatByAppId:appSecret*). An invocation is a statement where the credential is taken as a parameter by a function call. To further filter the credentials with the features, we match the two features with some keywords. If at least one of them contains certain keywords (e.g., SDK, secret), the candidate is chosen as a potential one. Finally, those collected candidates are sent to the validator of iCredFinder . The validator adopts a dynamic analysis approach to validate them by leveraging some remote web service APIs (e.g., Access Token authentication API). If the remote API based dynamic query proves that the credential is a valid one, iCredFinder will report a risky credential use case.

III. System Design and Implementation

A. App Preprocessing

1) App collection: Before the binary code analysis, we have collected a large number of candidate iOS apps to build our sample set. The apps are selected from the top list of the iOS app market as they are likely to integrate various third-party SDKs and are frequently updated and maintained by the developers. Apps cannot be directly used for analysis if they are downloaded from the official Apple APP Store because they have been encrypted. To address this issue, we have to make use of the decryption tool like `dumpdecrypted` or `Clutch` to decrypt the apps.

2) Extraction of Mach-O Executable: Analyzing iOS apps with IDA will take a Mach-O executable as input. This executable can be found in the .ipa package of the app. To simplify the process, we implemented an automated tool for the extraction of executables. The hierarchical structure of a directory of an iOS app is displayed in Figure 4.

The .ipa file is essentially a compressed file containing resources and code of the app. Take Baidu Wallet app as an example, our target executable is *BWA*, which shares the same name with its parent directory *BWA.app*. As a result, we can easily write a program to automatically extract the Mach-O executables from a large number of apps.

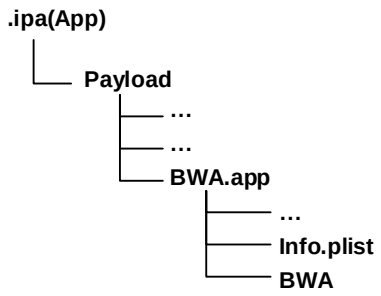


Fig. 4: The Hierarchical Structure of a Directory

B. Harvesting SDK Credentials

To find SDK credentials from a large number of iOS apps, we conduct the following steps to fulfill the target. Initially, our system takes the Mach-O executables from the app set as input. Then, the locater of iCredFinder utilizes some features to match the candidate secrets, and at the same time locates their addresses in the assembly code. After that, the filter of iCredFinder further selects the strings according to their function name and invocation. Finally, the system validates all the strings to find out the valid secrets. In the following paragraphs, we will introduce the detailed steps.

1) Locating Credential Candidates: The analysis of SDK credential starts from locating the credential candidates. Through the disassembling and analysis process, we can get the assembly code as well as the string set of the program. The SDK credential of the third-party SDKs has distinct features, it is simply a 32-character hex string. We can easily exclude most of the useless strings utilizing this feature.

The next thing to do is locating the addresses of the strings. In this step, the *XrefsTo* function in IDA Python is used to check all the references of the given address. Specifically, most of the strings we found in the string set are *cstrings*. To locate the references of the string, the following steps are executed.

(i) Given one *cstring* and its address, *XrefsTo* is called to locate its referenced address.

(ii) Through step (i) we can get the referenced *cfstring* address of the original string. Then, *XrefsTo* is called again to locate its referenced addresses in the assembly code.

(iii) From the previous steps, we can get the addresses of the assembly code where the string is embedded. We map the given string with these addresses as output.

Note that in some cases a *cstring* is directly referenced by the assembly code, so step (ii) is unnecessary. Through these steps, we can select many candidate secrets and locate their addresses in the assembly code.

2) Filtering Credential with Features: The collection of secrets we obtain from the previous step is too large to be validated because it still contains a large number of useless strings. In this step, we will filter the secrets based on the features. According to our observation, the function name and invocation of the secrets often contain useful semantic information. For example, secrets often appear at some "set" and "get" functions, and are frequently called by the SDK initialize functions like "SSDKSetupWechatByAppId:appsecret". Therefore we take the function name and invocation as our feature to filter the SDK secrets.

After the locating of credentials, we obtain a set of candidates as well as their addresses. we then develop scripts based on IDAPython and make use of the provided APIs to fulfill specific functions, such as disassembling and decompilation. The *GetFunctionName* and *get_pesudocode* function provided by IDAPython enable us to get the function name and the pesudocode based on the given addresses. Note that in the pesudocode, the invocation statement are often cut into several lines and thus we need to recover it to be a complement statement. The decompilation step is implemented by the Hex-rays decompiler of IDA Pro. The detailed steps to obtain features are displayed in Algorithm 1.

Algorithm 1 Feature Extraction

Input: secret, address
  for each $line$ in pseudocode **do**
    $targetLine \leftarrow$ line where secret appears;
  end for
  for $line$ between $targetLine$ and $endLine$ **do**
    if $line$ contains ";" **then**
      $endIndex \leftarrow$ index of $line$
    end if
  end for
  for $line$ between $targetLine$ and $startLine$ **do**
    if $line$ contains "$objc\_msgSend''$ **then**
      $startIndex \leftarrow$ index of $line$
    else
      if $line$ contains ";" **then**
        $startIndex \leftarrow$ index of $line$ - 1
      end if
    end if
  end for
Output: functionName, invocation

We go through all the lines in the pesudocode to locate the credential. The invocation of functions in Objective-C is implemented through the delivery of messages, specifically in the assembly code, when function *objc_msgSend* is called. This function takes a class, a selector and the function arguments as parameters. As a result, our target focuses on the *objc_msgSend* functions where the credential is taken as a parameter.

To extract the invocation of the credential, we starts from the line of code where credential appears and search forward until a semicolon is found, which indicates the stop of a statement and use $endIndex$ to mark the end of an invocation. Then, we start again from the code line of the credential and search backward until *objc_msgSend* is detected to locate the start

point which is denoted by $startIndex$. Finally, we recover the whole invocation and record it.

After we obtain the function name as well as the invocation, we filter the credential based on keyword matching. If both of the features do not match any keywords, the credential will be abandoned. Otherwise, it is selected as a potential credential. Our keywords are selected based on our observation, like "secret", "login", etc. During the matching process, we convert all the strings to lower cases.

It is worth mentioning that since some functions are obfuscated or encrypted during the development, meaningful information is lost and IDA can not correctly recover their name. These functions are those with names starting with "sub_". However, our approach is resilient to this issue because we use both of the features for judgment.

C. Credential Validation

The validation of credentials makes use of the official authentication API which aims at getting the access token. The authentication process is designed based on the OAuth2 protocol. Developers need to get that token through the HTTP GET or POST method with some necessary parameters. Our validation process is implemented by a light-weighted command line tool *curl* to simulate the HTTP requests.

During our experiments, we notice an important rule that the arguments of the request are checked orderly. For example, when the return message indicates the second argument is invalid, it also means the first argument has passed the check on server and thus is valid. Specifically, many web servers of third-party SDKs check the parameters in the order of *appid*, *secret* and then *code* or *redirct_URL*. As long as a correct *appid* is provided, we can send several authentication requests to see which credential is valid. Fortunately, the *appid* often have distinct features and can be easily found. For example, the *appid* of WeChat SDK is a string starts with "wx" and has a length of 18. From the previous step of locating credentials, our system also extracts some strings that are likely to be *appids*. During validation, we try them one by one until a valid *appid* is found. Then we test each of the credentials to find out the valid credentials.

In real cases, when an invalid credential is tested, the return message from the server will indicate that the credential is incorrect. On the contrary, the return message will show that there is an error on *redirect_URL* or *code*, meaning that the credential has passed the validation on the server and is a valid one.

IV. Evaluation

In this section, we will present our evaluation of the analysis of SDK credentials. We introduce our target SDKs and app samples in IV-A, present the evaluation results on third-party SDKs in IV-B, and finally discuss the effectiveness and performance of iCredFinder in IV-C as well as the suggested best practices in IV-D.

A. Analysis Targets

Prior to this study, we have built a dataset of 20,000 iOS app executables. We conduct a search on those apps to find the most frequently used SDKs. We extracted all classes whose names contain keywords like "SDK", "API", and then determine what SDKs are used. The result reveals that the top two most frequently integrated SDKs are the WeChat and Weibo SDKs that are adopted by about 80% of the apps, and own over 40 million users. Other SDKs such as QQ SDK, Jingdong SDK, and Alipay SDK are used only by less than 40% of the apps. As a result, our experiments focus on those two most popular SDKs.

We analyzed 100 most downloaded apps with at least one SDK integrated into our dataset. Overall, among the 100 app samples, 94 of them have integrated WeChat SDK, and 78 of them have integrated Weibo SDK. 72 of them have integrated both of the SDKs.

B. Experimental Results

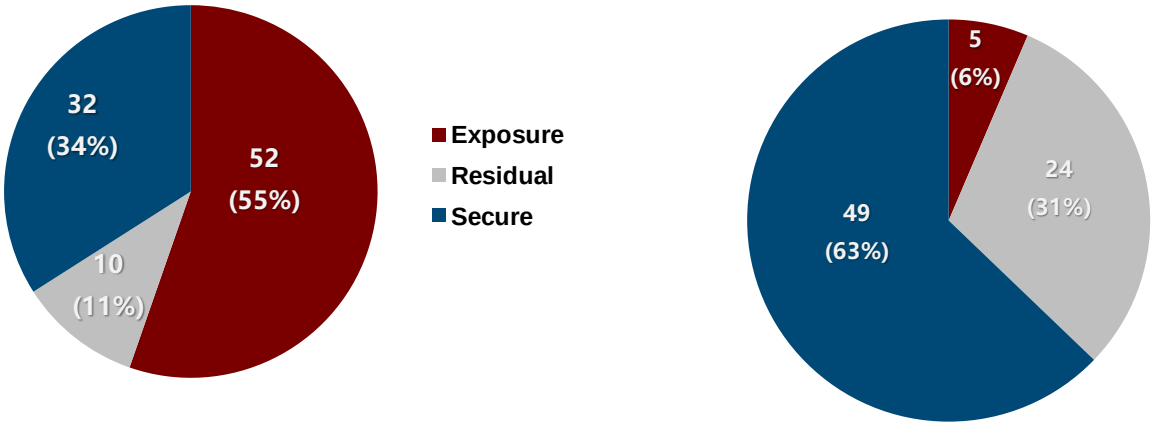
During our evaluations, we respectively conducted analysis on apps that have integrated WeChat and Weibo SDK. The overall results are presented in Figure 5. To sum up, we have drawn the following conclusions based on our experiments, and then we will respectively present results on each of the SDKs in detail.

- Among the 100 app samples, a majority of them suffer from the direct exposure or the residual credential issues where some apps can still work normally when the credentials are removed. It is revealed that many developers are unaware of carefully preserving the SDK credentials.
- The exposure of credentials is mainly caused by the direct request for access token on the client side. As for the residual problems, various reasons may lead to this consequence, such as the consideration of compatibility and the carelessness of the developers.
- The extracted credentials of Weibo SDK are less than those of WeChat SDK. The major reason is that most of the developers follow the official guide and implemented the login function based on the SSO login module.
- The credentials discovered by iCredFinder are all in plain text. They are directly used without any protection such as encryption or encoding.

Note that the official guides of those two SDKs have emphasized the importance of credential protection and suggested, and developers are expected to adopt secure practices to preserve them [3], [4]. To better understand the root causes of the frequently occurred misuses, in the following, we detail the analysis for two SDKs, respectively.

1) WeChat SDK Credentials. We analyze the credentials of WeChat SDK on 94 iOS apps. As shown in Figure 5a, we extracted one or more valid credentials from 62 apps. Among them, 52 (55%) apps directly expose the credentials in the program, 10 (11%) apps suffer from the residual secret issue, while only 32 (34%) apps securely preserve their credentials.

- **Direct exposure.** Among the 52 apps which directly expose their credentials in plain text, iCredFinder extracts



(a) WeChat SDK (b) Weibo SDK

Fig. 5: Evaluation Results of WeChat and Weibo Credentials

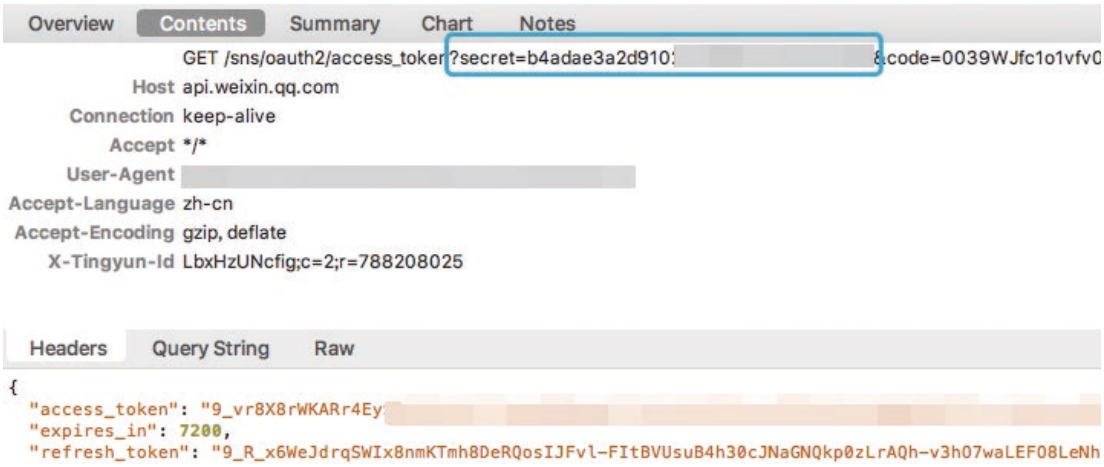


Fig. 6: Direct Exposure of Secret

the credentials and performs validation. Most of them adopt the same insecure coding practice, which requests for the access token directly on the client side through the official OAuth2 APIs. With that access token, developers can implement functions like sending messages and sharing information. According to our dynamic analysis on the packages shown in Figure 6, a secret is directly used as a parameter in the client's request, and as a result is likely to be exposed directly in the program. Some other developers although do not invoke the official API in the authentication process, still take credentials as parameters in other network requests, which also leads to the credentials exposure issue.

- **Residual credential issue.** There are 10 apps suffer from the residual credential issue. Unlike the direct exposure issue, these apps can still work normally even when the credentials are removed. Through our observation, the functions taking credentials as parameters are never used or even have been removed. Nevertheless, most of the credentials are extracted by iCredFinder and are proved to be valid ones, while some others are expired. Compared with the direct exposure problem, it is totally possible for the developers to address this vulnerability during the develop process. For one thing, during software iteration, careless developers may forget to delete useless credentials after modifying the login module. For another, the developers may preserve different login modules in the program for compatibility considerations.
- **Secure practice.** As for the remaining 32 apps free from the vulnerabilities, the developers do not request for

access token directly on the client side through the official OAuth2 API. Instead, them upload the authentication code to their own servers to let them start the request, as is shown in Figure 7. When the authentication process is done, the servers send back the access token to the client side for the access of third-party service. The sensitive credentials are well-preserved on the cloud and attackers are not able to get them through reserve engineering the mobile apps. This is a secure practice advised by the official guide.

2) Weibo SDK Credentials. We analyze the credentials on 78 iOS apps integrated with Weibo SDK. We successfully extracted one or more credentials from 29 apps, as depicted in Figure 5b. Among these apps, 5 (6%) apps directly expose their credentials, 24 (31%) of them suffer from the residual credential issue, while the other 49 (63%) apps are free from secret vulnerabilities.

- **Direct exposure.** Among our samples, only 5 apps directly expose the credentials in the program, accounting for only 6% of the apps. The way of exposing credentials is the same as that in WeChat SDK, i.e., developers request for access token directly on the client side and take the credential as a parameter.
  The authentication process of Weibo SDK is similar to WeChat SDK, as both of them adopt the OAuth2 protocol. However, the results of these two SDKs are significantly different. The number of exposed credentials is much less because a majority of developers adopt Single Sign On (SSO) login module which is also provided by the Weibo SDK. Our analysis result is shown in Figure 8. Credentials are not needed in the SSO login process, so these apps are free from the exposure of credentials. According to the official guide of Weibo SDK, developers are suggested to implement the login module based on SSO. Therefore, thanks to this suggestion, the direct exposure of credentials are eliminated in most of the apps.
- **Residual credential issue.** Although most of the developers have avoided the direct exposure vulnerability of credentials, there are also 24 apps suffer from the residual credential issue, which is the major cause of the misuse of Weibo SDK secrets. A reasonable explanation is that the OAuth2 authentication module is reserved for compatibility consideration.
  It is also worth mentioning that although there are 4 apps that did not implement the login function, we still successfully extracted valid Weibo SDK credentials from them. These outliers are not recorded in our result. Through validation, the credentials and appids are proved to be valid and belong to the apps themselves. This may be as a result of requirement changes and the carelessness of developers.
- **Secure practice.** Lastly, a majority of apps do not have vulnerabilities on their credentials, accounting for 63% of the total. The developers of these apps properly implemented the login module based on SSO and carefully follow the official guide. Some others although adopted their own implementation of the authentication process, did not directly embed the credentials in the program. Therefore, these credentials cannot be extracted through revese engineering.

C. Effectiveness and Performance

We implement iCredFinder mainly based on the state-of-the-art disassembler IDA Pro and its script subsystem IDAPython. iCredFinder reuses the disassembly output from IDA Pro and utilizes IDAPython to manipulate those outputs. It works stably on all 100 samples without facing any disassembling and decompilation error. There are no false positives in our evaluation results, since we tested all the credentials through the official API and can make sure their validness. False negatives may exist but it is hard to automatically find out all these credentials. The validation process depends on the correctly provided *appids*, while in some cases, developers may purposely hide them and our system fails to extract them out. Besides, some developers may adopt encryption or encoding to protect *appid* and *secret*, which thus adds to our false negatives.

The performance of our analysis is efficient: most of the analysis tasks can be finished in less than 30 minutes. In addition, the analyses of different apps can be fulfilled in parallel. The most time-consuming part of our analysis is code disassembling and decompilation, whose execution time depends on the size of an app. While for the validation process, only about tens of potential credentials are selected for final validation in each app on average, so the execution time of the validation process may take just a few seconds when the network condition is fine and thus can be ignored.

D. Best Practices

From the perspective of developers, we also suggest some secure practices that help protect SDK credentials from being extracted through reverse engineering.

- **Do not directly embed important credentials in the client-side, preserve them on the cloud.** When requesting for authentication, the mobile app should gain the permission of users and upload the authentication code to the cloud. The cloud server then requests for the access token and return it to the client side.
- **Delete useless secrets.** Developers should always remove any credentials from the program even if a credential is expired.
- **Use encryption and encoding to protect SDK credentials.** Credentials embedded in the client-side program should be carefully preserved. The key for decryption can be obtained from the cloud server or can be generated from a complex algorithm. These can make it very hard for the attackers to get SDK secrets through reverse engineering.
- **Clean up the memory after authentication.** Some residual secrets and access token may be still in the

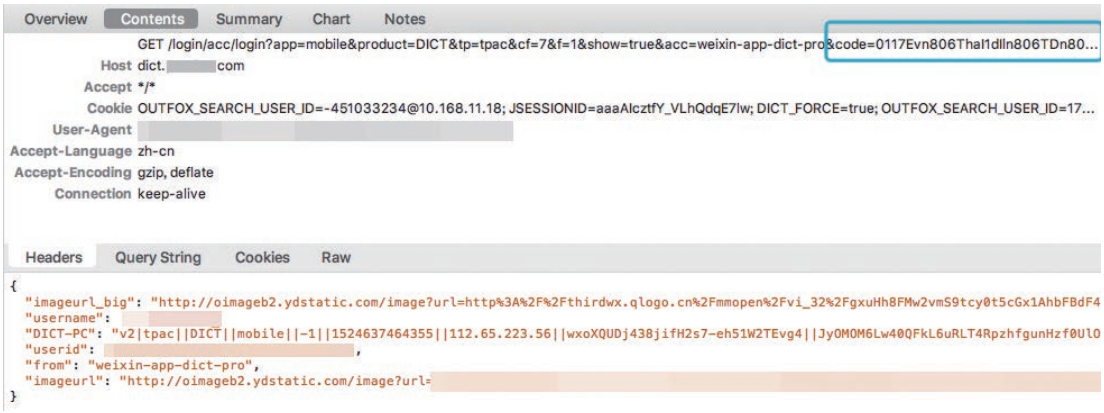


Fig. 7: Secure Practice to Preserve Secret

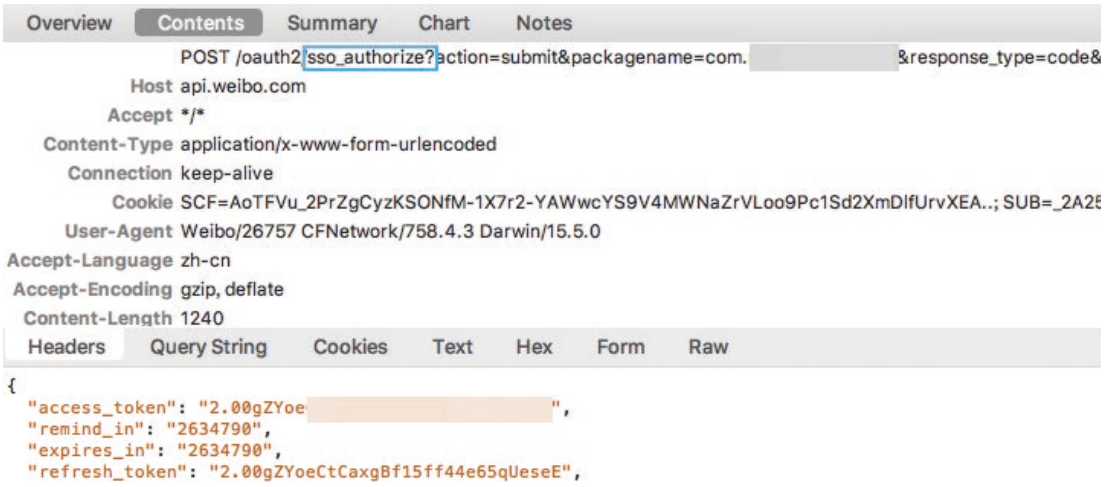


Fig. 8: SSO Login of Weibo SDK

memory and thus are vulnerable to attackers in some cases.

V. Related Work

Security Analysis on iOS Platform. Though iOS is a relatively mature and secure mobile OS, there are still plenty of security concerns on it. Many research focused on the security issues on iOS mobile apps and adopted static and dynamic analysis to identify vulnerabilities in these apps. The uncovered concerning problems include the leakage of privacy and insecure developer practices. PiOS [5] and PSiOS [6] are designed to detect the privacy leakage and the latter can also help address the vulnerable against attacks. More recently, researches have discovered that the privacy in iOS apps is also vulnerable to effective OS-level side-channel attacks [7].

Besides the detection of privacy leakage, many efforts have been made on the identification of insecure developer practices. For example, iCryptoTracer [8] is proposed to identify the misuses of cryptography functions. CRIOS [9], a system aiming at large-scale app analysis, can uncover the library usages and network security problems.

In addition, there are also other security vulnerabilities on iOS platform. Obfuscation in iOS apps is studied about their motivations and pitfalls. The authors also propose several possible obfuscation approaches to hide the sensitive information in iOS apps [10]. The sandbox mechanism is systematically studied, as the researches put forward a novel approach to analyze the sandbox profiles and also present ways to bypass it to obtain system level information [11]. Deng et al. focused on the vetting procedure. They discovered the security issues on Apple's apps vetting process and suggested a comprehensive and secure vetting approach [12].

Vulnerabilities Identification on Mobile Apps. Mobile apps on other platforms also have security concerns. Besides iOS, most of the researches are conducted on the Android platform. Due to the huge differences between these two operating systems, the problems studied and the techniques used are also different.

Third-party SDKs are widely used today by a large number of apps and may have security issues. CredMiner [1] is designed as an effective tool to extract SDK credentials through static analysis and then validate them. The researchers focused on storage and mail SDKs and successfully recovered some

developer credentials. Yang et al. concentrates on payment SDKs and concluded that many developers violate security rules, which may cause financial loss [13]. ClueFinder [14] is an NLP-based learning system to identify privacy leakage from apps to untrusted third-party libraries. It scales well on a large number of Android apps.

Some studies focused on the network security of web-based mobile apps. AUTOFORGE [15] is an automatic tool to forge valid request messages from the client side, and have uncovered the vulnerabilities of servers due to insufficient checks. Mendoza et al. implemented a static analysis-based web API reconnaissance approach to investigate the inconsistencies of validation logic between client and web side [16]. SMARTGEN [17] can automatically expose hidden server URLs and thus can judge whether these URLs are harmful.

Other vulnerabilities on mobile apps include the violation of privacy requirements [18], origin message stripping during the delivery of messages in webviews [19], residual of TLS keys in the memory [20], and code injection attacks [21]. In this paper, we differ from the efforts above in investigating the misuses of SDK secrets of SDKs which have never been studied before. Our work is conducted on iOS platform and we designed an effective analysis approach to extract and study these misuses of secrets.

VI. Conclusion

In this paper, we conducted an empirical study on the misuse of SDK secrets in iOS apps. We implemented iCredFinder to automatically extract SDK credentials within apps and then validate them. For evaluation, we selected two widely-used SDKs, the SDK of WeChat and Weibo, as our targets. Our experiments on 100 iOS apps succeeded in extracting valid SDK credentials from 66% (62/94) and 37% (29/78) of them, respectively. Our evaluation has also uncovered that some credentials are left unused in the apps. The exposure of SDK credentials can cause security threats like the leakage of user information and app data. Given the fact that a large number of apps suffer from the exposure credential vulnerability, it is urgent for the careless developers to adopt secure practices to preserve these exposed SDK credentials.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their valuable comments and helpful suggestions. The work was partially supported by the Key Program of National Natural Science Foundation of China under Grant No.:U1636217, the General Program of National Natural Science Foundation of China under Grant No.:61872237, and the National Key Research and Development Program of China under Grant No.: 2016YFB0801200. We especially thank the Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial joint Institution of FinTech Security*.

References

[1] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2015, p. 23.

[2] "Wechat and weibo release their financial reports for q3 2017," https://chozan.co/2017/11/16/wechat-data-weibo-data-q3-2017/.

[3] "Providing wechat login in your mobile app," http://open.wechat.com/cgi-bin/newreadtemplate?t=overseas_open/docs/mobile/login/guide.

[4] "Weibo sdk," http://open.weibo.com/wiki/SDK/en.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *NDSS*, 2011, pp. 177–183.

[6] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz, "Psios: bring your own privacy & security to ios devices," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 13–24.

[7] X. Zhang, X. Wang, X. Bai, Y. Zhang, and X. Wang, "Os-level side channels without procfs: Exploring cross-app information leakage on ios," in *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018). Internet Society*, 2018.

[8] Y. Li, Y. Zhang, J. Li, and D. Gu, "icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications," in *International Conference on Network and System Security*. Springer, 2014, pp. 349–362.

[9] D. Orikogbo, M. Büchler, and M. Egele, "Crios: toward large-scale ios application analysis," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2016, pp. 33–42.

[10] P. Wang, D. Wu, Z. Chen, and T. Wei, "Protecting million-user ios apps with obfuscation: motivations, pitfalls, and experience," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 235–244.

[11] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi, "Sandscout: Automatic detection of flaws in ios sandbox profiles," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 704–716.

[12] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 44–56.

[13] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! finding flawed implementations of third-party in-app payment in android apps," in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[14] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[15] C. Zuo, W. Wang, Z. Lin, and R. Wang, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services." in *NDSS*, 2016.

[16] A. Mendoza and G. Gu, "Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," in *S&P 2018: 39th IEEE Symposium on Security and Privacy*, 2018.

[17] C. Zuo and Z. Lin, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 867–876.

[18] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *24th Network & Distributed System Security Symposium (NDSS 2017), NDSS*, 2017.

[19] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications," in *S&P 2018: 39th IEEE Symposium on Security and Privacy*, 2018.

[20] J. Lee and D. S. Wallach, "Removing secrets from android's tls," in *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[21] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 66–77.