On the Insecurity of SMS One-Time Password Messages against Local Attackers in Modern Mobile Devices

Zeyu Lei[1], Yuhong Nan[1], Yanick Fratantonio[2], and Antonio Bianchi[1]

[1]Purdue University, [2]EURECOM & Cisco Talos
[1]{*lei76, nan1, antoniob*}*@purdue.edu*, [2]*yanick@fratantonio.me*

***Abstract*—SMS messages containing One-Time Passwords (OTPs) are a widely used mechanism for performing authentication in mobile applications. In fact, many popular apps use OTPs received via SMS as the only authentication factor, entirely replacing password-based authentication schemes. Although SMS OTP authentication mechanisms provide significant convenience to end-users, they also have significant security implications. In this paper, we study these mobile apps' authentication schemes based on SMS OTPs, and, in particular, we perform a systematic study on the threats posed by "local attacks," a scenario in which an attacker has control over an unprivileged third-party app on the victim's device.**

This study was carried out using a combination of reverse engineering, formal verification, user studies, and large-scale automated analysis. Our work not only revealed vulnerabilities in third-party apps, but it also uncovered several new design and implementation flaws in core APIs implemented by the mobile operating systems themselves. For instance, we found two official Android APIs to be vulnerable by design, i.e., APIs that inevitably lead to the implementation of insecure authentication schemes, even when used according to their documentation. Moreover, we found that other APIs are prone to be used unsafely by apps' developers.

Our large-scale study found 36 apps, sharing hundreds of millions of installations, that misuse these APIs, allowing a malicious local attacker to completely hijack their accounts. Such vulnerable apps include Telegram and KakaoTalk, some of the most popular messaging apps worldwide. Finally, we proposed a new and safer mechanism to perform SMS-based authentication, and we prove its safety using formal verification.

I. Introduction

SMS messages (also known as "text messages") are widely used to Two-Factor Authentication mechanisms (2FA). In these scenarios, a user is asked to provide a token received via SMS *in addition* to their username and password. In addition, within the realm of mobile apps, the SMS channel has also started being used as One-Factor One-Time Passwords (1FA OTPs, in short). In these scenarios, the user is asked to "prove" the ownership or their own telephone number, which acts as the main user identifier: "owning a telephone number" equates to "owning the user account" associated with it. This mechanism is implemented by first asking the user's phone number, and then sending an *authentication code* to this number. Finally, either the user is asked to insert the received authentication code, or the app automatically reads it from the incoming SMS, at which point the app can send the code back to the app's backend. This procedure proves ownership of a specific phone number (and of the corresponding SIM card). We note how this protocol effectively uses the SMS channel as the *only* "factor" to authenticate to a user's account.

This mechanism brings significant convenience to users, especially since users no longer need to create and remember a new password for each app requiring authentication. It thus does not come as a surprise that this mechanism is widely used, including top-popular messaging apps such as Telegram [48]. In particular, we found that among the top 100 Android apps from the communication category in Google Play, 24 apps support using SMS OTP code as the *only* "factor" for user account authentication. This is a growing pattern, to the point that Google has introduced three new APIs in the last few years to support this specific use case.

Unfortunately, this mechanism also brings significant security implications, since the SMS communication channel has been proven insecure on many occasions. Multiple times, attackers have been able to target the telephony networks and successfully redirect the OTP messages to an unintended receiver. The most prominent example is SIM-swapping [39] attack, in which an attacker can lure the telephony company to obtain a SIM card associated with the victim's phone number. Another instance of this issue is the exploitation of the SS7 network (the network internally used of telecom company to route calls and SMS messages) [21]. Moreover, it is known that state-level attackers can intercept SMS OTP messages, by interfering with local telecom companies [59]. Most recently, a study [45] has shown that many Android apps insecurely implement the generation and verification of SMS OTPs, used in their authentication protocols. For example, some apps generate SMS OTPs with insufficient randomness or insufficient length.

Local attacks against SMS OTP authentication schemes. While previous works have focused on exploiting vulnerabilities in the SMS channel itself, in this paper, we focus on a different class of attacks against SMS OTP messages, which we call *local attacks*. With local attacks, we indicate a threat model in which an attacker has control over a malicious third-party app installed on the victim's device. The goal of the malicious app is then to "steal" authentication codes sent via SMS. These attacks are devastating for apps using the aforementioned 1FA OTP authentication mechanism since they allow obtaining the only factor used to authenticate a user. In addition, they also weaken the security of

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24212
www.ndss-symposium.org

traditional SMS-based 2FA solutions, since they allow obtaining one of the two factors.

Although these attacks have been previously studied [10], [18], [20], [32], [46], modern devices and operating systems implement new APIs and mechanisms to ease developers and users' lives, when implementing and using SMS OTP authentication schemes. To the best of our knowledge, *no study has systematically analyzed the security of these new mechanisms, and no study has performed a large-scale evaluation to determine the relevance and the impact of these threats.* Our research aims at filling this gap.

Recent operating systems' changes to support SMS OTP authentication. To move toward a more secure way for SMS OTP authentication, both Android and iOS employ protection mechanisms to prevent SMS OTP messages from being read by unauthorized apps. For example, iOS does not allow any third-party apps to read or access SMS messages. On the other hand, in Android devices, third-party apps can request permission to read received SMS messages, including those messages potentially containing OTPs. However, starting from January 2019, restrictions have been put in place to limit the usage of SMS-relevant permissions [12].

Preventing apps from arbitrary reading SMS messages improves the security of SMS-based authentication schemes since it blocks automated exploitation (i.e., without user involvement). However, it also affects the overall user experience, since users may be required to manually type these OTP messages character by character in the legitimate app requesting them. For this reason, new mechanisms for handling OTP messages have been introduced by iOS [58] and Android [27], [28], [30], aiming at providing, at the same time, more security guarantees and better user experience.[1]

Although these mechanisms are more feasible and promising as the future mainstream for SMS OTP authentication, little has been done regarding understanding whether these mechanisms are indeed secure, which is the most critical thing for SMS OTP authentication. Unfortunately, as we will show in this paper, these new mechanisms introduce a new set of weaknesses and vulnerabilities in mobile apps using them. We believe our findings are important and concerning, particularly because these recent OTP-protecting security mechanisms were *specifically* designed to protect from the local attacker threat model.

Our work. In this paper, we conduct the first in-depth systematic analysis of the security of SMS authentication usage in modern mobile platforms. Our study started with analyzing how modern mobile operating systems allow users and apps to access SMS OTP messages. This analysis was carried out using a combination of studying the official documentation and sample code, reverse engineering, and formal verification (using ProVerif [11]). These analyses led to surprising results, identifying a set of attack surfaces that are either new or existing for a long time but still have not been fixed yet.

Our research shows that the newly introduced mechanisms to access SMS OTP messages, while designed to improve the security and the usability of this authentication method, severely hinder the security of the application using them.

For example, we show that some of the APIs introduced to ease the usage of these authentication schemes suffer from profound design bugs that make it *impossible* to be used safely. Therefore, apps that use them are susceptible to account hijacking attacks (see Section VII), which can be carried out by a malicious third-party app that *does not request any permission*.

In addition, we also performed two user studies to evaluate if and how a malicious app can lure a victim user to perform actions allowing an attacker to steal SMS OTP messages.

To evaluate the severity and real-world impact of these security threats, we then conducted a large-scale measurement on 140,586 Android apps, to understand how developers use SMS-authentication-related APIs and semi-automatically detect vulnerable apps. This large-scale measurement revealed common pitfalls in the usage of these APIs that make several apps vulnerable, including extremely popular communication apps such as Telegram [48] and KakaoTalk [47]. A successful adversary who obtains the SMS OTP messages could take full control of the victim's account (see Section IX-B).

Compared to prior research, we systematically studied the ways that enable an adversary to obtain the SMS OTP messages through a malicious app running on a victim's device, running a modern mobile operating system. Our study not only shows that the existing threats are not well mitigated by current security enhancements, but it also reveals a set of new threats that are possible due to design and implementation errors in new mechanisms to access SMS OTP messages, introduced by modern mobile operating systems.

Lastly, we propose modifications and improvements to the currently available APIs in mobile operating systems to implement a more secure scheme for SMS-based authentication. We show that, differently than previously thought, there *is* a technical way to achieve the sweet spot between usability and security. Our design provides stronger security guarantees without losing any practicality. For example, our design, similar to the current APIs, does not require user interaction or permissions. At the same time, it prevents malicious third-party apps from reading SMS used for authentication by legitimate apps (see Section X).

In summary, this paper makes the following contributions:

- We systematize how the different mechanisms offered by the mobile operating system to implement SMS-based authentication can be attacked. We uncover design bugs in recent Android APIs that were introduced specifically to protect from the threat of *local attackers*, which we consider in this paper.
- We perform a user study and a large-scale automated study to show the impact of these attacks in practice. Our studies lead to the discovery of several critical vulnerabilities, affecting several highly popular mobile applications, including Telegram and KakaoTalk.
- We propose improvements and modifications to the APIs currently available in mobile operating systems to implement SMS-based authentication.

II. Threat Model and 1FA SMS Schemes

In this section, we first discuss the threat model and assumptions considered in our research, and we then present an overview of 1FA OTP SMS-based authentication schemes adopted by many popular apps and how a local attacker can exploit their weaknesses.

[1] We note that since Android provides to developers most of the innovative ways to access SMS OTP code, our research is mostly focused on Android. However, our paper also discusses the limited scenarios specific to iOS (Section III-A).

A. Threat Model

Attacker model and capabilities. This paper focuses on *local* attacks. These attacks consist of an adversary who controls a malicious app installed on a victim's mobile device. We note that this is a common threat model in the field of mobile security, in-line with prior works [9], [10], [49], [50], [64]. In fact, it is unfortunately feasible and practical for an attacker to include malicious functionality in a seemingly benign and useful app, and distribute it through app markets [34], [42].

We also assume that the malicious app can communicate through the Internet to send the SMS OTP message (or the code it contains) to the attacker. In Android, this requires the Internet permission. Note that this permission is not suspicious as it is requested by the vast majority of mobile apps (e.g., a recent report [8] showed that 83% apps in Google Play Store require this permission). Additionally, since the Internet permission is considered "normal," obtaining it does not require any user interaction, and it is not listed in the menu allowing users to grant/revoke permissions.

To trigger the delivery of an SMS OTP message to the victim's device, we also assume that the attacker knows the phone number of the victim. This assumption is in-line with similar attacks performed in previous work [10]. There are a number of ways an attacker can obtain a victim's phone number. For example, many messaging apps use the phone number as the main user identifier. In these apps, having a victim as a "friend" automatically reveals their phone number. Moreover, in Android, a malicious app may use the Android API `getLine1Number()` to get the phone number (requiring the READ_PHONE_STATE permission). Furthermore, a malicious app installed on the victim's phone could show a legitimate-looking UI, requesting a phone number for authentication purposes. We acknowledge that this is a non-trivial requirement. Thus, we do not envision our attacks to be used in large-scale automated campaigns, but rather as part of targeted attacks.

While the aforementioned threat model and assumptions are valid throughout the entire paper, some of the attacks we will present are subjected to different constraints. In particular, some attacks we will present are harder and less likely to be carried out since, for example, they require specific interaction with the victim (see Section V) or they require a specific permission (see Section VI). Conversely, others can be carried out in a completely automated fashion (see Section VII). We will precisely specify the additional assumptions when we discuss each type of attack.

Assumptions on OS integrity. Throughout this paper, we make some additional assumptions. These assumptions are important when discussing the relevance of our attacks (e.g., some attacks are trivial if some of these assumptions do not hold) and when discussing the reliability of our proposed defense mechanism.

In this paper, we assume that the Android OS, including the kernel and the various system components (e.g., Android system services) are not compromised. Thus, we assume that app isolation and permissions are properly enforced. In our paper we also assume that the OS and apps can establish a secure Inter-process-communication (IPC) channel, and that, by using dedicated APIs (e.g., binding to system services [26], [55], pending intents [25]), an app can safely communicate with a system service, and, at the same time, a system service can know the identity of the app it is communicating with.

Previous works have shown how the OS and apps may be vulnerable to a number of attacks, such as broadcast injection and broadcast receiver hijacking [15], component hijacking vulnerabilities [43], and component leaks and service hijacking [36], [41]. These are well-studied classes of vulnerabilities and we assume that apps are protected from them.

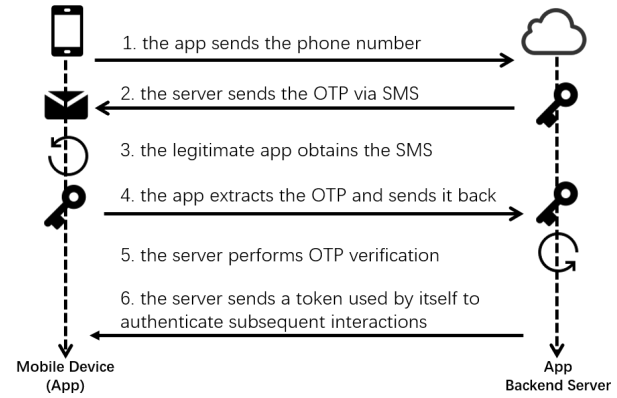


Fig. 1. A typical scenario using SMS OTP for 1FA.

Finally, we consider out-of-scope UI-based attacks that try to surreptitiously hijack user interaction, such as activity hijacking [13], [50] or accessibility-based attacks [22]. Modern Android versions enable several protection mechanisms to prevent these attacks [24]. For instance, the user now sees a warning if any overlay is present, and a malicious app cannot know anymore the app a user is interacting with. While it could be theoretically possible to still use UI-level attacks under some circumstances, we consider exploring these attacks as orthogonal to our main contributions.

B. SMS-based 1-factor-authentication Schemes

A common pattern used by popular messaging apps is to identify a user with a phone number, for which the user supposedly owns the corresponding SIM card. This aims to replace the more traditional username. In this scenario, an SMS OTP message is used to prove ownership of a specific phone number. Such an authentication scheme is widely adopted by many popular apps nowadays. Specifically, in a preliminary study we have done in May 2020, we have inspected the top 100 popular apps from the communication category in Google Play and found that 24 apps are indeed using the SMS OTP message as the only factor for authentication.

Figure 1 shows the multiple steps of the SMS OTP authentication procedure. As can be seen, in this scenario, the client (the legitimate app) first sends the phone number to the app's backend server (Step 1). Then, the app's backend server generates the OTP code and sends it to the phone which requested to authenticate, using an SMS message (Step 2). Later, at the mobile device side, the legitimate app obtains the SMS (Step 3), extracts the OTP code and sends it back to the app's server through a network connection (Step 4). Finally, the app's backend server performs the verification by comparing the received OTP code with the previous one, which it sends out via SMS (Step 5). Once the verification is passed, the backend server will send a token used by itself to authenticate subsequent interactions with the mobile app for further communications.

During this process, different properties need to be true to ensure a secure verification scheme. For instance, during Step 2, the generated OTP needs to have enough entropy to be resistant to a brute force attack by an adversary. Also, during Step 5, the

generated OTP needs to be compared against the one provided by the app. These security properties have been previously studied and discussed [51] and found to be violated in some apps' implementations [45].

Specifically, AUTH-EYE [45] shows that, in many cases, the SMS OTP is generated incorrectly (e.g., insufficient OTP randomness or insufficient OTP length), or that the OTP code is incorrectly verified (e.g., too many allowed retry attempts or too long renewal interval). Our work is complementary to AUTH-EYE.

In fact, in our work we assume that the OTP generation and the OTP verification are implemented correctly, and, instead, we focus on how the SMS OTP is delivered to the app and read from it. In particular, we focus on the potential weakness in *OTP confidentiality* used during the authentication process. More specifically, in Step 3 of Figure 1, it is crucial that *only* the legitimate app can read the OTP message sent by the app's backend, rather than other apps hosted on the device.

C. Example of an End-to-end Attack Scenario.

If an attacker-controlled malicious app is able to obtain the OTP targeting another app, an attacker can easily bypass SMS-based 1-factor-authentication schemes. In particular, the attacker controlling a malicious app on a victim's phone can achieve this malicious goal through the following three steps:

1) The attacker initiates authentication with a victim app's backend, specifying as phone number the victim's one;
2) The attacker uses the malicious app to steal the OTP that is received via SMS by the victim's device, and sends the OTP out (e.g., to an attacker-controlled device) through the malicious app;
3) When asked to insert the OTP in the authentication procedure initiated during the first step, the attacker uses the stolen OTP to complete the authentication procedure with the app's backend, effectively pretending to be the victim.

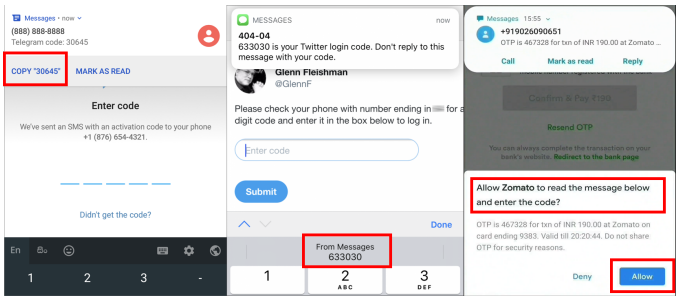
We note that our paper is focused on explaining how an attacker can maliciously obtain the SMS OTP messages, i.e., the second step in the list above. The other two steps, while slightly different for each victim's app, can be trivially performed (and potentially also fully automated), using, for instance, an attacker-controlled device. Nevertheless, in Section IX-B, we will provide concrete examples of this attack.

We also note that the different presented attacks may introduce a delay between the time a SMS OTP is sent to the victim's device and the time the attacker sends the OTP to the app's backend server. While properly implemented SMS-based 1-factor-authentication schemes should set a maximum validity time for OTPs, this value is typically in the order of minutes, while the delays our attacks introduce are in the order of seconds.

III. Legitimate Methods to Access SMS OTP Messages

In this section, we enumerate the different methods that apps can use to access received SMS OTP messages, on both Android and iOS. Later, we will then describe how the different presented methods are susceptible to different attacks.

In this paper we consider features available in Android and iOS up to their versions 10 and 12, respectively. While for iOS the available methods to SMS OTP messages are very limited, Android offers a plethora of different methods, backed by different



(a) Android (Ask-to-copy) (b) iOS (Ask-to-copy) (c) Android (One-Tap SMS)

Fig. 2. Illustration of different methods for accessing SMS OTP code with user-interaction

APIs, with sometimes subtle or poorly documented differences. This section discusses these methods in detail (for both OSes), and it is organized in three parts, according to the type of interaction (if any) required by the user: methods requiring per-message user interaction (Section III-A), methods requiring SMS-related permissions (Section III-B), and fully-automated methods that do not require user interaction (Section III-C).

A. Access with User Interactions

An app may ask users to manually copy the OTP code (i.e., inserting character-by-character) from the received SMS to the app's user interface. This method does not require any particular app permission nor the usage of any specific API. However, this kind of user interaction is susceptible to "phishing attacks," in which a user may be tricked into inserting the just-received OTP into the app in foreground, which may be malicious. In addition, this procedure involves a somewhat laborious user interaction, introducing "friction" in the app authentication procedure.

Therefore, to enhance the user experience, modern versions of both Android and iOS provide a more user-friendly interface for SMS OTP verification. Specifically, in both Android (starting from its default messaging app with version 3.3+) and iOS (starting from iOS 12), the system provides an "ask-to-copy" feature, which can automatically parse each incoming SMS message to see if they contain OTP tokens [58], [61]. As shown in Figure 2(a), for Android, once the OTP code is identified from the incoming SMS message, the system will show on the system keyboard a "copy" option that automatically copies the OTP in the device's clipboard. This enables the user to paste the OTP to the current waiting app. Similarly, for iOS (Figure 2(b)), the parsed OTP code will be listed in the input keyboard. Once the user clicks it, the OTP code will be automatically filled into the current input field of the app. The two functionalities allow users to no longer keep switching between the SMS inbox and the app waiting for the OTP, streamlining the authentication procedure.

We note that in iOS, the "ask-to-copy" feature is the only option available for boosting SMS OTP input. For instance, iOS does not offer third-party apps any mechanism to read messages stored in the SMS inbox.

One-Tap SMS verification. Android also provides an alternative approach, which is called *One-Tap SMS* for SMS OTP authentication. Compared to approaches as mentioned earlier, the One-Tap SMS provides additional user prompt and eliminates the copy-paste interaction. Specifically, as shown in Figure 2(c), this mechanism presents a pop-up window as a user-consent,

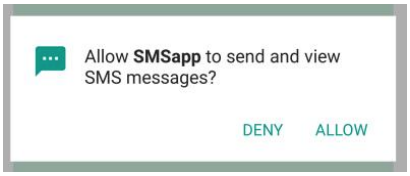


Fig. 3. Illustration of an app requesting SMS-related permissions at runtime.

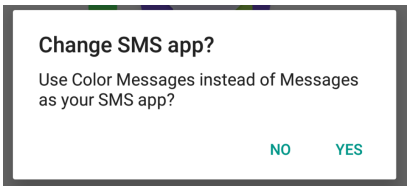


Fig. 4. Illustration of an app request setting itself as the default SMS app.

to directly fill a given OTP code to a specific app. If the user allows the app to access and parse the incoming message containing the OTP code, the code will be directly delivered to the app. However, this approach requires additional app-side implementation by developers rather than the Android operating system. It is important to note that different from the "ask-to-copy" mechanism; the One-Tap mechanism shows the app name to the user, in which the message is going to be copied into (compare Figure 2 (a) and Figure 2 (c)). This design may help in preventing *phishing attacks*, but it does not solve the problem at its roots.

In Section V, we will study the impact that these more streamlined OTP insertion user interfaces (allowing the user to copy the OTP automatically) have on the security of an SMS OTP authentication procedure.

B. Access by Requesting SMS Permissions

In Android, apps can request SMS-related permissions to access the SMS inbox programmatically. The two relevant permissions are `READ_SMS` and `RECEIVE_SMS`: the first allows an app to read the SMS inbox at any time, while the second allows the app to only read the "new" incoming messages just before they are saved into the inbox.

Both these permissions provide sensitive (in terms of security and privacy) capabilities to an app since they allow it to read arbitrary SMS messages, even when these messages are not relevant to the app functionality. For this reason, these permissions are classified by the Android OS as *Dangerous* [12] (starting from Android 6) and require a one-off user-consent at runtime (as shown in Figure 3).

Additional restrictions. Given the security relevance of these permissions, starting from January 2019, Google has introduced additional restrictions for any app uploaded to the Play Store that requests SMS-related permissions [17], [57]. The first restriction is that these permissions can only be requested by apps that are designed to be an "SMS handler," i.e., *a replacement of the default SMS handler app*. Apps that aim to be "SMS handlers" must follow additional policies:

- The app needs to be suitable for being an SMS handler app (e.g., it needs to allow the user to read received text messages, send new ones, etc.).
- The app must ask the user if they want to change the current default SMS app to this one, upon its first usage.

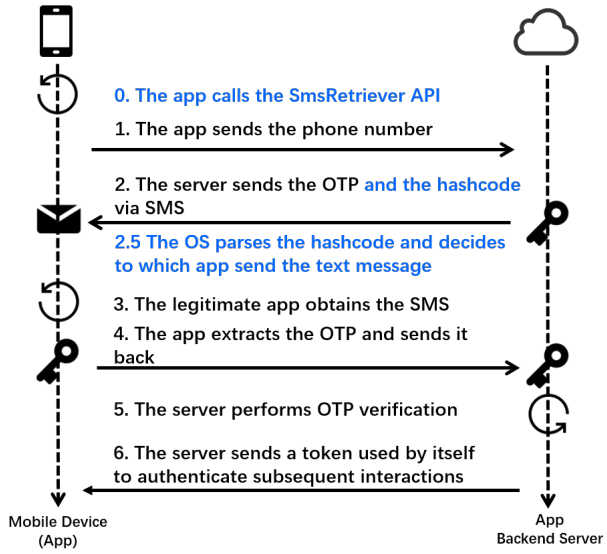


Fig. 5. Authentication process for SMS Retriever.

To enforce these restrictions, the market performs manual, human-assisted verification of every app requiring these permissions. Unfortunately, even with these restrictions in place, we will show that a developer can still manage to publish their malicious app to the Google Play Store (see Section VI-B). In addition, our study also shows that users are not aware of the security implications of granting the permission to read SMS messages to an app (see Section VI-A).

C. Fully-automated Access via Modern SMS APIs

To further improve the user experience of the app, Android introduced three different APIs to support the SMS OTP authentication in a fully automated way without requiring any permission and user interaction. In the rest of the paper, we will collectively refer to these APIs as "Modern SMS APIs."

Key design and benefits. The overall idea of these modern SMS APIs is to require the app server to send the authentication SMS message with an additional identifiable string (e.g., a hash obtained from a cryptographic signature) which would indicate the intended receiver of the OTP code. The identifiable string allows the OS to only deliver the SMS message to the intended target app rather than the others. Recalling Figure 1, these mechanisms streamline Step 3 of the described procedure in an automated way instead of user intervention (i.e., asking the user to input the OTP code manually). In addition, the design of these mechanisms *does not require the app asking for any Android permission*. These benefits further encourage the adoption of these methods by app developers to implement SMS OTP 1FA, since apps are no longer subjected to the limitations and restrictions by requiring permissions, nor they require user interaction to copy the received message content.

Here, we present the details of these three different APIs. For simplicity, we will refer to these APIs with `SMS Retriever`, `SMS Token`, and `SMS Token+`, respectively. The details of these three APIs are summarized in Table I.

We note that with the term "API," we refer not to a specific Java method, but to a specific authentication functionality provided by the Android framework (which usage may require calling multiple

TABLE I. SAMPLE AND HIGHLIGHTED DIFFERENCE BETWEEN THE MODERN APIS FOR SMS OTP AUTHENTICATION.

API	Saved to inbox?	Sample SMS	App identifier	Key code snippet used in app for implementation
SMS Retriever	Yes	Your ExampleApp code is: 1234 FA+9qCX9VSu	Hashcode, fixed string	SmsRetriever.getClient(context).startSmsRetriever() Intent.extras.get(SmsRetriever.EXTRA_SMS_MESSAGE);
SMS Token	No*	Your ExampleApp code is: 1234 Eqhn_SOxhzw	Token, randomly genrated.	createAppSpecificSmsToken (intent)
SMS Token+	No*	[WhatsApp]: Your code is: 1234 dF4Sse6U7d	Same as in SMS Token (Based on the documentation)	createAppSpecificSmsTokenWithPackageInfo (prefixes, intent)

* The SMS will not be saved in the inbox if it can be correctly delivered to the intended app, see Section VIII-A for more details.

methods). We will use the term "method" instead when referring to a specific Java method (e.g., `startSMSRetriever()`).

SMS Retriever. The `SMS Retriever` API uses a *hashcode* (e.g., `FA+9qCX9VSu`) to specify to which app an SMS OTP should be delivered. This app-specific hashcode is computed starting from the app's signing certificate and its package name. Specifically, it is computed using the following formula:

```
x = concat(app_package_name, app_signing_certificate)
hashcode = truncate(base64encode(SHA256(x)), 11)
```

where `concat` stands for string concatenation and `truncate(string, X)` stands for string truncation after `X` characters.

The signing certificate in Android is embedded in the app's package file, and it is associated with the private key used by the developer to sign the app. The package name is a string that uniquely identifies an app on a mobile device and on the Google Play Store (that is, two apps on the same device or on the Google Play Store cannot have the same package name). Since the developer is supposed to securely holding the private key, an attacker cannot create a valid signed APK with the same app certificate. In turn, this should imply that an attacker could not easily create a malicious app so that its hashcode collides with the hashcode of the benign app (due to the SHA256 second pre-image resistance [38]) [2].

As shown in Figure 5, after an app invokes the method `startSmsRetriever()` and register for the `SmsRetriever.SMS_RETRIEVED_ACTION` Android listener (the Step 0), subsequent text messages containing the app-specific hashcode will be automatically routed to the app. In other words, an app does not need any Android permission to read a message containing its own hashcode. For this reason, when an app uses the `SMS Retriever` API, in Step 2, the server sends a message containing both the OTP code and the hashcode.

SMS Token. One limitation for `SMS Retriever` is that it is only supported when the Google Play Service is available. This limitation makes it not feasible for some Android devices in specific scenarios, e.g., Android devices in China, which do not come with Google Play Service installed. As a result, starting from August 2017, Google provides a new API to implement automatic SMS OTP authentication, which is available for *all* devices with Android version 8 or higher. In this paper, we refer to this API as `SMS Token`.

The intended usage of SMS Token is similar to SMS Retriever. However, SMS Token differs in how it generates the app-specific token that, when included in an SMS, causes the OS to redirect it to the intended app. Specifically, as shown in Figure 6, while

[2]Unfortunately, the formula truncates the hash to 11 base64 characters, potentially allowing malicious collisions, as we will explain in Section VIII-B.

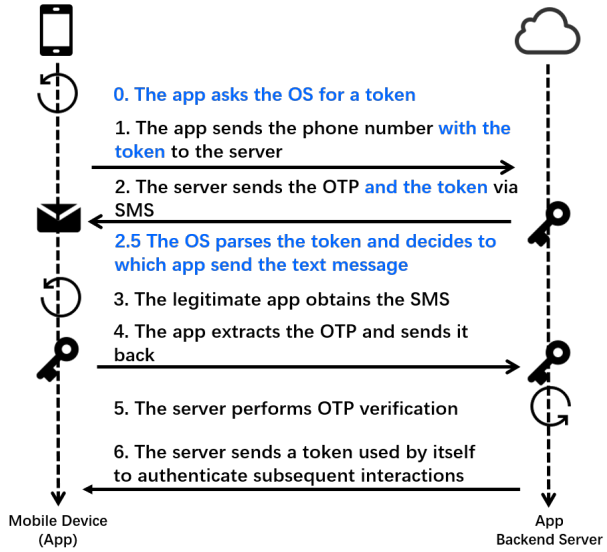


Fig. 6. Authentication process for SMS Token (and SMS Token+).

`SMS Retriever` uses a fixed, app-specific hashcode, the SMS Token API generates a *random token* every time the `createAppSpecificSmsToken()` method is invoked (Step 0). This token is supposed to be sent to the backend server (in Step 1) from the phone. Further, the SMS OTP message is delivered to the legitimate app, based on the token included in the SMS message. For example, if an app A obtained the random token *Eqhn_SOxhzw* as a return value of the `createAppSpecificSmsToken()` method invocation, a subsequently received SMS message containing the text *Eqhn_SOxhzw* will be delivered to the app A.

Unfortunately, as we will explain in Section VII-B, this API is *inherently unsafe* due to its design flaw.

SMS Token+. Android 10 introduced yet another SMS-related API, invoked by using the *createAppSpecificSmsTokenWithPackageInfo()* method, which we refer to as `SMS Token+`, in short. This API generates another type of token for automatic SMS authentication. Based on the API description from the Developer guide [27], one may think that this API works in a very similar way as the `SMS Token` API.

In fact, according to the documentation, the *only* difference is that this new API allows a developer to provide an additional list of prefixes for SMS filtering. Only SMS starting with one of these specified prefixes will be delivered to the calling app.

The reader may note how the possibility of specifying a prefix should *not* affect the security of this API. Thus, one would think

that this API is as secure (or as insecure) as the previous SMS Token.

To our surprise, we found that this is *not* the case: while the two APIs *appear* to work in the same way and they have a *very* similar documentation (with the only stated difference of being able to filter based on prefixes), *these two APIs are internally implemented in a very different way*. What is even more interesting is that our analysis found that `SMS Token+` *is still vulnerable to our attacks, but due to different reasons*. We discuss these attacks in Section VII-C.

IV. How to Maliciously Obtain SMS OTPs

As explained in the previous section, in modern platforms, there are a variety of ways for an app to read SMS OTP messages used for authentication (Step 3 in Figure 1). Unfortunately, as we explain in detail in the following sections, all these approaches include potential pitfalls. This is true even for those methods that were designed to make accessing SMS OTPs more secure.

Specifically, in Section V we show that the newly introduced mechanisms to streamline the copy-and-paste of OTPs from SMS messages do not improve the overall security, since they are still prone to deception attacks.

Then, in Section VI, we will show that, users may be willing to give the `READ_SMS` permission to malicious apps. Our results are complementary to previous studies on how users (mis-)understand permissions. In fact, our study shows that, for the specific case of the `READ_SMS` permission, most of the users have a general understanding of how it works, but they are unaware that an app able to read SMS can also potentially compromise other apps' accounts.

Finally, in Section VII we will show that the new APIs introduced to automatically read SMS OTP messages in Android are either error prone or intrinsically unsafe. Consequently, many apps using these APIs are vulnerable. In addition, attacking these vulnerable apps can be done automatically, without requiring any user interaction. In fact, while the attacks presented in Section V and Section VI require to deceive users to perform some kind of UI interaction (either to copy-and-paste an OTP or to grant a permission), these attacks do not require user interaction.

V. Getting SMS OTPs by Deceiving Users

In this section, we elaborate on the attack in which an adversary can deceive users and obtain the SMS OTP message through a malicious installed app *without any SMS-related permissions*. Different from the previous phishing attacks in which the malicious app mimics the authentic UI and steal user inputs [14], [20], here, we focus on understanding how the newly introduced user-interaction-based mechanisms (e.g., the One-Tap mechanism) can affect the efficacy of UI deception attacks. Specifically, we first elaborate on the attack scenario and its root cause. Then, we perform a user study to objectively measure potential weaknesses of such user-interaction-based mechanisms for SMS OTP authentication.

A. Assumptions

The attack presented in this section assumes that an attacker is able to deceive the user. To deceive the user, the attacker controls a malicious app that requests an SMS OTP message in a seemingly legitimate scenario (e.g., while registering a new account in an SMS OTP 1FA scenario). Note that as mentioned earlier in Section II-A, this attack is orthogonal to other UI deception attacks that exploits the OS-level weakness (e.g., task hijacking or activity hijacking [13], [50]). Specifically, the attacker first asks the user for their phone number, as it typically happens when interacting with an app using SMS OTP for the first time. Then, the attacker remotely requests the SMS OTP message of a user's legitimate account (e.g., Facebook) by contacting the legitimate app's backend server and specifying as phone number the one just inserted by the victim. At this point, the victim will receive an SMS message containing the legitimate account's OTP, and the attacker-controlled app will try to deceive the user into inserting the received OTP into it. Notice that in here, since the victim will receive one and only one message after the insertion of their phone number, it is impossible for the victim to distinguish the attack by the timing of the message. Since the incoming message exactly meets the user expectation of receiving an SMS OTP message, the victim user will likely input the OTP code in the phishing app. This deception attack can be made even worse by the fact that the SMS sent by the legitimate app's backend may not contain a clear indication of the name of the app that it is targeting.

B. Understanding User Reactions to Deception Attacks

Previous studies have already explored this type of attack and empirically showed that users are likely to be lured by them [23]. However, *no one has studied if the newly introduced mechanisms to ease the OTP copy-and-paste user interaction mitigate this threat.* (recall the "ask-to-copy" and One-Tap mechanisms described in Section III-A)

Therefore, in this section, we evaluate the impact that these new mechanisms have in mitigating this attack. To this aim, we performed a user study to measure the effectiveness of these deceiving attacks in different scenarios, including situations in which these newly introduced OTP code insertion mechanisms are used.

Design of the user study. To perform the user study, we obtained IRB approval from our institution, and we used subjects recruited using mTurk [4], a popular task recruitment platform, successfully used in similar security-related user studies [9]. By using a dedicated mTurk feature, which allows us to know user experience with different OSes, we selected users familiar with iOS to test iOS-specific scenarios, and users familiar with Android for Android-specific scenarios. During our evaluation, we removed those subjects not completing the assigned task or incorrectly answering our validation questions.

The participating subjects were asked to use their browser to access an interactive environment for a usability study, simulating an Android phone or an iOS phone. We implemented this interactive environment by using an app prototyping online tool named Marvel App [2]. This interactive environment was used to simulate a phishing scenario in which the user is required to register a new account in a malicious app. The registration process requires the user to input the OTP received in an SMS message. During this process, the malicious app uses the user's phone number to request an OTP message from another account (in our test, Telegram, a popular communication app). In this simulated environment, the participants can either input the OTP by clicking the OS-provided copy button or manually type it through the on-screen keyboard. We present the details of these scenarios step-by-step in Appendix (Figure 11 and Figure 12).

We divided subjects into four groups: Android-Manual, Android-One-Tap, iOS-Manual, and iOS-Autofill. Subjects in

the two "Manual" groups had to insert the OTP by exclusively using the system keyboard. Subjects in the groups of Android-One-Tap and iOS-Autofill were respectively offered the two new mechanisms in Android and iOS for SMS OTP authentication. We consider the subjects' accounts as compromised once they inserted the OTP code into the malicious app.

TABLE II. USER REACTIONS TO DIFFERENT PHISHING SCENARIOS FOR STEALING SMS OTP MESSAGES BY A MALICIOUS APP

Test Case	Total #	Escaped users # (%)	Compromised users # (%)
Android-Manual	51	28 (55%)	23 (45%)
Android-One-Tap	61	26 (43%)	35 (57%)
iOS-Manual	51	15 (29%)	36 (71%)
iOS-Autofill	51	16 (31%)	35 (69%)

Results and findings. As shown in Table II, our study first confirms that the percentage of subjects inserting the OTP in the malicious app is high over the different scenarios, ranging between 45% to 71%. More importantly, our study shows that the percentage of subjects that escaped the attacks does not change significantly between the manual input and the new mechanisms (i.e., Android-One-Tap and iOS-Autofill). Therefore, we conclude that *all current available methods that require user-interaction to acquire an OTP, including the newly introduced ones, are highly vulnerable to deception attacks*.

As a more concrete example, consider the Android *One-Tap SMS* mechanism. Recall that, as shown in Figure 4(c), the interface of this mechanism clearly shows the app name when the user is about to insert the OTP code. In our simulation, we set this name to be "FunnyChat" instead of being "Telegram". In other words, the name appearing in the One-Tap interface is the name of the malicious app trying to steal the OTP. Nevertheless, based on the results of our user study, this additional indication still did not prevent users from inserting the OTP into the malicious app.

In addition, there is another issue making this mechanism more prone to deception attacks. Specifically, although the One-Tap API clearly shows the user *the name of the app* in which the text message is going to be inserted (see Figure 2(c)), we verified that *a malicious app can arbitrarily name itself as the target app*, and hence making itself harder to detect when requesting the OTP code of the target app. In fact, we were able to create an app showing the name "Telegram" on its One-Tap interface and get it published on the Google Play Store.

Design Weakness #1: Users do not have a reliable way to identify the identity of the app in which they are asked to copy an SMS OTP message.

In summary, we conclude that users can be easily deceived in inserting an OTP into a malicious app, and the newly introduced mechanisms (i.e., the "ask-to-copy" and One-Tap mechanism) do not improve the user's ability to detect these deceiving attacks.

VI. BYPASSING OS PERMISSIONS AND MARKET RESTRICTIONS TO ACCESS SMS MESSAGES

For an attacker, the most straightforward way to steal the SMS OTP in Android is to have the permissions to read/intercept all the incoming SMS sent to a device. This attack scenario has been widely known and discussed by previous research [32], [46], and modern OSes has employed extra restrictions as countermeasures [12], [17], [57]. Specifically, for an attacker, obtaining this permission is hindered by both OS-level restrictions, imposing that the user has to explicitly grant this permission to an app, and by Market-level restrictions, which mandates additional vetting for apps requesting this permission. However, in the rest of this section, we will show how, and under which conditions, it is still possible to bypass these restrictions due to the various related design weaknesses.

A. Bypassing OS-level Restrictions

Exploiting users' misunderstanding about SMS permissions. The permission to read SMS is categorized as *Dangerous* in Android. Therefore, an app wanting to obtain it has to show a specific system-managed dialog box on which the user has to press the button "ALLOW" (shown in Figure 3).

Prior studies have demonstrated that due to the lack of understanding about technical details, certain users will either totally ignore the warnings about the permission usage or fail to understand the meaning of the different permissions [35]. However, we claim that, in the specific case of the permission to read text messages, the situation is even more worrisome. In this case, the majority of the users do not understand the full consequences of pressing the "ALLOW" button in the interface shown in Figure 3.

To empirically investigate this claim, we performed a second user study, in which we asked Android users what the consequences of pressing the button "ALLOW" (in the dialog box shown in Figure 3) are. This user study has been performed under the same setting as the previous user study (described in Section V-B), using mTurk [4] for recruiting participants and Marvel App [2] for simulating the interaction with a phone.

In this study, we first show the home screen of an Android phone, in which a set of well-known popular apps (e.g., WhatsApp, TikTok, and Amazon) are installed. Then, we show that this device is installing a new app requiring the permissions to send and read SMS messages with a system dialog window (see Figure 4). Finally, we ask our subjects to answer a set of questions (as shown in the left column of Table III) about the potential consequences of pressing the "ALLOW" button in this dialog window. Subjects were allowed to select multiple options. Details of the survey are presented in Appendix (Figure 10).

TABLE III. STATISTICAL RESULTS OF THE SURVEY WITH 57 VALID PARTICIPANTS.

Available options	# Subjects choosing True
1) Read all my SMS messages	52 (91%)
2) Get access to my accounts on WhatsApp or Telegram	22 (39%)
3) Modify any SMS messages in my inbox	23 (60%)
4) Destroy my phone	16 (28%)
5) Leak my location	33 (58%)

1. Question – Please choose "True" for all potential consequences of clicking "ALLOW" in the prompt window.
2. Option 1 and 2 are correct and the others are incorrect.

In total, we collected answers from 57 valid participants. As can be seen from Table III, since 91% (52/57) of our subjects correctly selected `option-1`, we can conclude that it is clear to the vast majority of the users that pressing the "ALLOW" button in the permission prompt allows an app to read all SMS messages on the device. While previous studies [35] have shown that many users struggle to understand Android permissions, our study shows that for the specific case of the `READ_SMS` permission, most of the users have a general understanding of how it works. However, most of them are unaware that by allowing an app to read SMS, the app can potentially compromise accounts of other applications. In fact, most of the subjects know how the permission system works in general, but only 39% of them know

that an app that can read SMS messages can also read OTPs and, therefore, compromise accounts of popular apps. We believe this is an indication that many users are unaware of this specific threat when taking permission-related decisions in Android.

We believe that this is due to the fact that the permission dialog box does not provide a sufficient explanation of the security consequences of this choice. More specifically, the dialog box simply says "Allow ⟨app name⟩ to send and view SMS messages?", but it does not mention that having the ability to read SMS messages also allows a malicious app to compromise accounts of those apps using 1FA SMS OTPs.

In summary, we conclude that the SMS permission prompt does not provide sufficient information regarding the security consequences of pressing its "ALLOW" button.

Design Weakness #2: The system interface asking for the permission to read SMS do not explain the severe security implications that this choice can have.

B. Bypassing Market-level Restrictions

App version update. As mentioned earlier, an app requiring the permission to read the SMS inbox is subjected to additional policies when uploaded to the Google Play Store. Specifically, the app is manually checked by Google Play to ensure it is suitable for being an SMS handler app. However, we found that the enforcement of this policy can be easily bypassed. In fact, we were able to publish on the Google Play Store an app not following these requirements.

To achieve this goal, we first implemented an app that respects these requirements. Then, we uploaded it to the market and waited for its approval. Once approved, we modified it (by publishing an update), transforming it in an app that does not show the default SMS prompt (Figure 4) and that it is able to silently read text messages and upload them online [3]. While the initial approval took several days (suggesting a comprehensive analysis performed by market operators), the updated was accepted *in a few hours*. Theoretically, it is also possible to detect the malicious behavior we added to the uploaded app through static analysis. Unfortunately, this was not the case in our experiment. We also tried to upload an application requesting the permission to read text messages, but not following the market policies. In this case, the app was rejected.

To this end, our experiments suggest that, most likely, the human-assisted verification is *not performed* when the app is updated, since app updates are accepted in a few hours while new submissions are accepted after days. We acknowledge that the Play Store may be treating differently apps than have a meaningful number of users. However, as of November 2020, our uploaded app has been downloaded more than 100 times and it even received one comment from a legitimate user. Still, it does not exhibit any sign of further verification.

Design Weakness #3: Market-level policies are not verified for updated versions of an already published app.

Requesting alternative permission. We also found another way to bypass this vetting process. An Android app can obtain a permission named `BIND_NOTIFICATION_LISTENER_SERVICE` to read notifications received by the user. All received SMS messages trigger a notification containing their content. Therefore, an app having the permission to read notifications can effectively read all the incoming messages, including those containing OTPs.

Indeed, recent researchers [52] found malware in the Google Play Store, which specifically utilizes the notification system as a sidestep to SMS-related permission restrictions, and hence steal the OTP messages of other accounts. In addition to what already found by this work, we noticed inconsistencies on how the permission to read notifications is handled compared to the `READ_SMS` permissions.

Specifically, the "reading notification" permission is considered as *Special* by the Android OS [29]. To obtain this permission, the app has to ask the user to open a dedicated interface and select the name of the app. Thus, from the perspective of convincing the user to grant this permission to a malicious app, obtaining the permission to read notifications is *harder* than obtaining the read SMS permission, since it requires complex user interaction (not just pressing an "ALLOW" button). Surprisingly, publishing on the Google Play Store an app asking the permission to read notifications is *easier* than publishing an app requesting the permission to read SMS messages, since it does not trigger any extra vetting. We confirmed this by submitting to the market an app whose only behavior is to ask the permission to read notifications. The app was accepted without any particular request or delay.

Design Weakness #4: Market-level policies and OS-level policies are not aligned.

VII. Exploiting Modern SMS APIs

In previous sections, we have described attacks requiring some form of user interaction, either in the form of the user copy-pasting the OTP, or granting specific permission to an app. Instead, in this section, we will show how to abuse recent APIs in modern Android versions (the ones discussed in Section III-C) to perform automated, stealthy, and user-interaction-free attacks in various circumstances. We will also report the results of a large-scale study on how many apps are affected by these attacks in Section IX.

Assumptions. For this type of attack, we assume that the victim app uses one of the system-provided APIs for authentication. However, we do *not* assume our malicious app to have any permission other than `Internet` (for sending out the SMS OTP code), and we do *not* assume any user interaction. The permission-less and interaction-free nature of these attacks makes them more worrisome than the previous ones.

Preliminary observations. There are two fundamental observations that lay the basis for our attacks against these modern SMS APIs. The first observation is that, in modern Android versions, *if a malicious app can control part of the content of an SMS OTP message, the malicious app can read the entire message, without requiring any permission nor user interaction*. This surprising behavior is due to the existence of the three APIs described in Section III-C allowing to access SMS messages without requiring any permission depending on whether they contain specific strings (i.e., the token or the hashcode). For example, consider Figure 7, and suppose that an attacker is able to lure (in Step 1) a victim app's backend server to send an SMS OTP message (in Step 2) whose content includes the hashcode associated to the malicious app: in this scenario, the OS would *automatically* redirect the OTP-carrying SMS to the malicious app, without the need of requesting any permission.

[3] Due to ethical considerations, we implemented the app so that it only exhibits the malicious behavior when running on our testing devices.

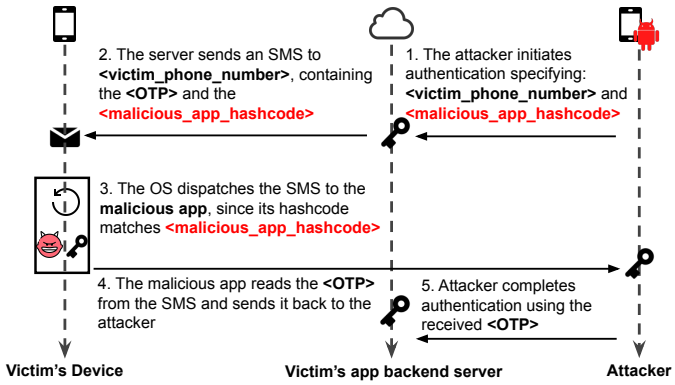


Fig. 7. Illustration of how an attacker can obtain the SMS OTP code from a benign app if its backend server is incorrectly implemented.

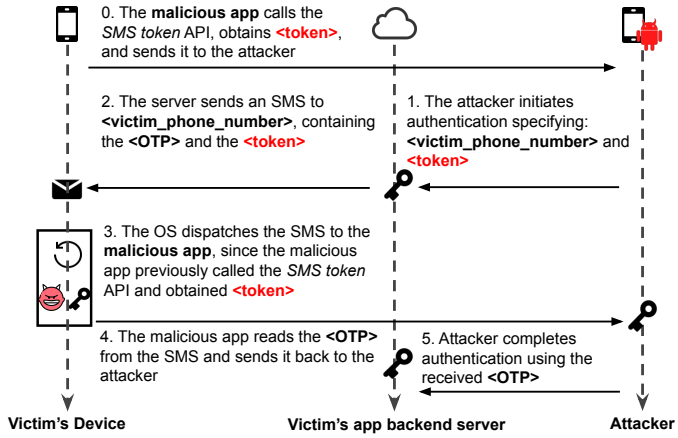


Fig. 8. Illustration of how an attacker can obtain the SMS OTP code from a benign app if it uses the `SMS Token` API.

The second observation is that, due to how these modern SMS APIs work, if a victim app (and its associated backend) use `SMS Retriever`, `SMS Token`, or `SMS Token+`, *an attacker could have a chance to lure the app's backend in delivering a partially-attacker-controlled OTP-carrying SMS message, thus giving an opportunity to the malicious app to intercept it*. The remainder of this section discusses the technical details on how a malicious app can perform full end-to-end attacks, depending on which of the three modern APIs is used and how the backend server's logic is implemented.

A. Attacking Apps using `SMS Retriever`

The `SMS Retriever` API requires the app's backend server to save the app's hashcode and include it in the SMS OTP messages. If implemented properly, an attacker would have no way to control the content of the delivered message. In this case, the presence of the hashcode will cryptographically ensure that only the legitimate app will be able to read the SMS OTP message without requiring any permission. This property is true under the assumption that the hash algorithm used is second-preimage resistant (we will discuss this assumption in Section VIII-B).

However, this API could be used unsafely if the app developers implement their backend's logic in a different and vulnerable way. Specifically, the backend may be implemented so that it receives the app's hashcode from the app itself, and then it inserts the hashcode to the generated SMS OTP message. At this point, the legitimate app's backend will generate an SMS OTP message (containing the malicious app hashcode) that will be delivered by the OS to the malicious app, instead of the legitimate one.

We found this issue to be surprisingly common. We speculate that this happens because computing the hashcode of an app is surprisingly unintuitive. The official documentation [28] presents a fairly complicated seven-step procedure to obtain this value, which involves downloading a signing key from the Google Play Store. Although possible to implement, we did not find any publicly available code that, given an APK file (the file format used to distribute an app), returns its hashcode. On the contrary, the official documentation provides some code that allows an app itself to obtain its own hashcode. While the same documentation also warns *not to* include the provided code in the app, we found that many developers include the code and use it to compute the hashcode. As a result, the locally generated hashcode is sent to the backend server and used for the SMS OTP message generation.

We also noticed another common way in which developers use this API unsafely. Instead of dynamically computing the hashcode in the app and sending it to the backend server, they hardcode the hashcode in the app and send the hardcoded string to the server, which then uses it to generate the SMS OTP message. While Google documentation specifically states: "Do not use hash strings dynamically computed on the client in your verification messages," it does not explicitly state not to trust a hardcoded and app-provided value in an app's backend code.

Design Weakness #5: The usage of the `SMS Retriever` API is error prone.

B. Attacking Apps using `SMS Token`

The `SMS Token` API is intrinsically unsafe regardless of how developers use it. Recall from Section III-C, that to use the `SMS Token` API, an app A has to call the `createAppSpecificSmsToken` API to obtain a random token T. The app then sends this token to its backend server, which will answer by sending an SMS containing the OTP, together with the token T. The OS will then detect that the received SMS contains the token T, and it will dispatch the message to the app A.

However, a malicious app installed on the victim's device can use the same mechanism to lure the victim app's backend. As shown in Figure 8, in fact, the malicious app can first call the `SMS Token` API to obtain a token T_2. Then, the malicious app sends this token to the attacker (Step 0). The attacker can initiate authentication with the victim app's backend server, specifying the victim's phone number as phone number, and T_2 as token (Step 1). Since the token returned by this API is random and always changing, the app's backend cannot tell whether the provided token comes from a legitimate user or an attacker, and it needs to *trust* whatever received during the authentication request. As a result, the backend will send an SMS OTP to the victim's phone, containing T_2 (Step 2). This SMS message will be dispatched by the OS to the malicious app (Step 3).

While it is possible (although error prone) to use the `SMS Retriever` API correctly, in the case of the `SMS Token` API, the impossibility of recognizing the legitimacy of the provided token makes the use of this API *inherently unsafe* for authentication purposes.

To further corroborate our claim, we used ProVerif [11], [37] to demonstrate the fundamental design flaw of the `SMS Token` API. ProVerif is a software for automated reasoning and verification about the security properties of a given cryptographic or communication protocol. In our case, we modeled the interaction between an app using the `SMS Token` API and a corresponding backend server, according to what shown in Figure 6. ProVerif was able to show how an attacker can obtain the OTP from the server. The details of the proof can be found in Appendix (Figure 13).

Design Weakness #6: The usage of the `SMS Token` API (`createAppSpecificSmsToken`) for authentication purposes is unsafe due to its vulnerable design.

Unfortunately, the official documentation of this API does not warn against the dangerousness of its usage. For this reason, we found developers of widely used apps (e.g., Telegram) to use it as part of their SMS-based authentication, making their app vulnerable to the aforementioned attack.

C. Attacking Apps using `SMS Token+`

The official documentation of the `SMS Token+` API suggests using this API in the same way as the `SMS Token` API. In this case, the app using this API is vulnerable to the same attack as in `SMS Token`, because the only described difference is about the presence of a potential custom prefix in the SMS OTP message, which is irrelevant from a security standpoint.

However, it is possible to use it in a safer way, but this mechanism is not explained in the official documentation. In other words, the vulnerability in this API is due to the incorrect documentation rather than its implementation (as in `SMS Token`). By reverse-engineering the implementation of this API, we found that its internal behavior is very similar to `SMS Retriever`. Specifically, the returned token is always equal to the calling app's hashcode rather than a random one. Likewise, a received message containing a hashcode A is only delivered to the app whose hashcode is A.

Consequently, the correct way to use this API safely is to ignore its return value and, instead, place the app's hashcode in the SMS OTP messages generated by the app's backend server. Unfortunately, this critical information is never mentioned in the documentation, and hence app developer will still implement this API in the same vulnerable way as in `SMS Token`. On the contrary, the official documentation states: "The token is only good for one use within a reasonable amount of time," hinting to the fact that the token is randomly generated or, at least, it may change.

Design Weakness #7: The `SMS Token+` API (`createAppSpecificSmsTokenWithPackageInfo`) documentation suggests using it in the same way of the `SMS Token` API, thus making its usage equally insecure.

D. Responsible Disclosure and Developers' Response

We reported our findings about the `SMS Token` and `SMS Token+` APIs to the Android Security team. We had a meeting with several Google engineers, they acknowledged the issues, and we discuss possible mitigations. Later, Google informed us that, in an upcoming quarterly update of Android, they plan to fix or deprecate these APIs.

VIII. Additional Design Weaknesses of the Modern APIs

In addition to what described in the previous sections, we identified other design weaknesses affecting the `SMS Retriever`, `SMS Token`, and `SMS Token+` APIs.

A. Modern APIs' Inbox Management

The `SMS Retriever`, `SMS Token`, and `SMS Token+` APIs are designed to deliver the content of an SMS OTP only to a specific app. Therefore, SMS received by using these APIs should not be stored in the SMS inbox. Otherwise, a malicious app able to obtain the Android permission to read SMS (as explained in Section VI) can read them and obtain the OTPs they contain.

Unfortunately, we found that this is not the case. Specifically, the `SMS Retriever` API always stores the received SMS messages (i.e., messages containing the app's hashcode) in the SMS inbox. Interestingly, for the other two APIs (i.e., `SMS Token` and `SMS Token+`), the received message does not go *normally* in the SMS inbox, but an attacker can force this to happen. In fact, these APIs avoid storing the received message in the inbox if and only if the following two conditions are both true:

- an app on the victim's device has previously called these APIs and obtained a token T;
- the incoming SMS message contains T.

An attacker can easily exploit this behavior by requesting to the backend server of an app using these APIs an SMS OTP specifying a random token, rather than a token returned by these APIs. Once received, the message containing a random token will be stored in the SMS inbox, and hence it will be readable by apps allowed to read text messages.

Design Weakness #8: SMS OTP Messages are stored in the SMS inbox (making them readable by any app with proper permissions), even when obtained with APIs designed to deliver them only to a specific app.

B. Cryptographic Weaknesses

From a cryptographic standpoint, the `SMS Retriever` API does not follow proper guidelines. Recall that the hashcode is computed by converting a SHA256 hash to a base64 string and truncating it to 11 characters. Effectively, this reduces the strength of the hashing algorithm to 66 bits (since a base64 character has 6 bits of entropy). Although truncating a hash in itself is not a security problem, NIST guidelines [16] mandate not to truncate a SHA256 hash to less than 224 bits. In fact, an attacker could be able to craft a malicious application having a specially crafted package name so that it has the same hashcode of a victim's app. This attack requires finding a second pre-image of a 66-bit hash, which, although hard, it could be computationally feasible for a determined attacker.

To test how Android and the Google Play Store behave in case of hashcode collision, we created two applications having the same hashcode. We note that due to the Birthday Attack, creating two colliding applications only required a few hours of CPU time. Interestingly, we were able to upload and get approved both the applications on the Google Play Store. Therefore, we concluded that market operators do not verify the absence of hashcode collisions among published apps.

Once two apps with the same hashcode are installed on the same device, we noticed that both apps stop receiving any message delivered with the `SMS Retriever` API. However, if a malicious app (with a colliding hashcode) is installed on a device in which the collided legitimate app is not installed, the malicious app can receive any SMS OTP message delivered by the legitimate app's backend.

Design Weakness #9: The `SMS Retriever` API does not respect security guidelines in terms of hashing strength. The market does not check for hashcode collisions.

IX. Large-scale App Measurement

To better understand how apps use the modern APIs for SMS OTP authentication, we perform a large-scale measurement analysis over Android apps in the Google Play Store. Our results show a number of highly popular apps confirmed to be vulnerable due to the usage of these APIs.

Dataset. To build our dataset, we obtain the package names of all available apps in Google Play Store using AndroidZoo [3]. Starting from this list, we downloaded all those apps with more than 50,000 downloads, based on the app's information shown in the Google Play Store. To boost our app collecting process, we downloaded the apps' APK files from both Google Play and third-party websites (e.g., APKPure [5]) based on their unique package names. Our final dataset includes a total number of 140,586 apps, downloaded between December 2019 and February 2020.

A. Vulnerable App Identification

We use a mix of static and dynamic analysis mechanisms to find apps that are vulnerable due to their usage of the modern SMS authentication APIs.

Static analysis. Our tool uses FlowDroid [7], together with a set of heuristics for locating those apps that are highly likely to be vulnerable.

To detect the usage of the `SMS Token` and `SMS Token+`, our analysis first checks for the method signatures (i.e., `createAppSpecificToken()` and `create...WithPackageInfo()`) along with the call graph constructed by FlowDroid. The call graph helps us to eliminate those dead code which are actually not invoked by the app. These two APIs are intrinsically unsafe if used according to their documentation (as explained in Section VII-B and Section VII-C). As a result, their usage is an indication of a possible vulnerable authentication scheme.

For apps using the `SMS Retriever` mechanism, our static analysis attempts to detect if (1) the app either contains its own hashcode or it dynamically computes it, and (2) sends it to a server. These features strongly indicate that the backend server may use the obtained hashcode from the app to create an SMS OTP message (making the app vulnerable, as explained in Section VII-A).

To detect the presence of a hardcoded hashcode, we compute the app's hashcode on our own, and we use string matching to find its presence in the app's code. Besides, to detect if an app dynamically computes its own hashcode, we check if the app invokes specific APIs needed for obtaining its own signing certificate (as shown in Figure 9) and how the results of these APIs are chained together. Lastly, we use the data flow analysis provided by FlowDroid to detect if the hashcode is indeed sent out through a network API.

```
android.content.ContextWrapper: getPackageName()
android.content.pm.Signature: toCharsString()
java.security.MessageDigest: update(byte[])
java.util.Arrays: copyOfRange(byte[],int,int)
android.util.Base64: encodeToString(byte[],int)
```

Fig. 9. Method signatures for dynamically generating an app's hashcode.

We note that developing a static analysis tool that can detect vulnerable apps with high precision is outside this paper's scope. Our goal is to have a tool that we can use to focus our subsequent dynamic analysis on those apps that are potentially vulnerable.

Dynamic confirmation. Further, we use dynamic analysis and manual reverse engineering to confirm if the candidate apps detected by our static analysis are indeed vulnerable. Specifically, we reverse engineer the apps to confirm the usage of the detected APIs as part of their authentication scheme. We classify apps passing our reverse engineering analysis as "Suspicious". Then, to confirm that an app is vulnerable (i.e., a malicious app can steal its OTP without requiring any permission nor user interaction), we verified that it is possible to lure the app's backend to generate an OTP message in a way in which an attacker can control its content. This property implies that a malicious app can steal the OTP, as we explained in Section VII.

To dynamically verify this property, we instrument the app (using re-packaging and the Xposed instrumentation tool [1]), to modify the hashcode (in case of `SMS Retriever`) or Token (in case of `SMS Token` and `SMS Token+`). Then, we manually interact with the app, triggering its authentication procedure. Finally, we classify the app as "Confirmed" if the app's backend sends us an SMS OTP containing a modified hashcode or Token. Also note that, in some of the apps, the backend server code logic got updated after we notified the developers of the vulnerability affecting their authentication scheme. We classified those apps as "Fixed".

Our dynamic verification reveals that most of the false positives reported by the static analysis are caused by those apps using hashcode for app integrity check (e.g., re-packaging detection) rather than implementing the `SMS Retriever` API.

Measurement results. Table IV summarizes our findings. We found 20 apps confirmed as vulnerable (Column 3 in Table IV) by the time of our dynamic analysis, which accounts for a total number of more than 133 million installations in the Google Play store. Meanwhile, we have found 16 apps (Column 4 in Table IV), which we previously confirmed as vulnerable and got a server-side fix, after we reported our findings to them. In summary, by considering both "Confirmed" and "Fixed" apps, we had 36 vulnerable apps, sharing more than 230 million installations.

Note that, due to several reasons, there are certain apps for which we cannot trigger the authentication procedure. For instance, some apps' backend servers only send SMS messages to international phone numbers, which we cannot obtain. For this reason, even if our reverse engineering suggested that their authentication scheme is vulnerable, we flagged them as Suspicious. We found 0 apps using the `SMS Token+` mechanism. We believe that this is because this API has only been introduced recently in Android 10.

TABLE IV. RESULTS OF OUR ANALYSIS OF 140,586 ANDROID APPS. NOTE THAT THE NUMBER OF UNIQUE APPS CAN BE LOWER THAN THE SUM OF THE NUMBERS IN EACH CATEGORY SINCE SOME APPS EXHIBIT MULTIPLE FEATURES.

	Candidates	Suspicious	Confirmed	Fixed
SMS Retriever: dynamic hashcode	56	20	9	0
SMS Retriever: hardcoded hashcode	38	7	4	3
SMS Token (createAppSpecificToken)	38	2	7	13
SMS Token+ (createAppSpecificTokenWithPackageInfo)	0	0	0	0
Total number of unique apps	129	29	20	16

B. Case Studies

1) KakaoTalk: KakaoTalk [47] is a popular instant messaging app, used by 93% of the smartphone owners in South Korea. The app is also extremely popular in other Asian countries [62]. We found that KakaoTalk's backend uses `SMS Retriever` with an app-provided hashcode. Specifically, the app's code contains a hardcoded hashcode that is sent to the app's backend and used by the backend to generate the SMS OTP message. Hence, its implementation is vulnerable (as we described in Section VII). For this reason, an attacker can create an account associated with a phone number that they do not own and impersonating the legitimate user.

We have recorded a demo video[4] to illustrate an end-to-end attack against KakaoTalk. The attack is carried out with the follow steps:

1) On the victim's device, the installed malicious app (`BadAppForVictim`) invokes `SMS Retriever`.
2) On the attacker's device, the attacker starts the sign up in the KakaoTalk app, specifying the victim's phone number.
3) On the attacker's device, the attacker alters the KakaoTalk app behavior (e.g., through the Xposed framework [1]), and sends the hashcode of `BadAppForVictim` to KakaoTalk's backend server.
4) The KakaoTalk's backend server sends the verification text message to the victim's device, inserting the hashcode of `BadAppForVictim`. Consequently, this text message can be read by `BadAppForVictim`.
5) On the victim's device, the `BadAppForVictim` sends the received SMS OTP message back to the attacker's device via Internet.
6) On the attacker's device, BadAppForHacker spoofs an incoming text message containing the stolen SMS OTP message. Consequently, on the attacker's device, the KakaoTalk app signs in as the victim.

Through the steps above, the attacker has successfully signed up using the victim's phone number and can now act as the victim to receive and response incoming messages. Someone adding to their KakaoTalk contact list the victim's phone number will end up communicating with the attacker's device, instead of the victim's device.

2) Telegram: Telegram is one of the most popular instant messaging apps in mobile platforms. As of January 2020, it has more than 100 million downloads in the Google Play [48]. The app was identified as vulnerable in a previous run of our experiment, performed in June 2019. During our research, we found that the SMS OTP authentication process in Telegram used both `SMS Retriever` and `SMS Token` (based on the Android version). The usage of the `SMS Token` API made the app vulnerable, as explained in Section VII-B.

[4]https://pursec.cs.purdue.edu/projects/sms_mobile.html

We also noted that on the Google Play Store, there exist many Telegram unofficial clients. These apps allow users to chat with other Telegram users and connect to the Telegram backend server. We found that many of these apps did not update their code as quickly as the official Telegram client. This aspect explains the apps that we classified as "Fixed" in Table IV, since these apps still contain the unpatched Telegram code, but they cannot be exploited, since they use the, now patched, Telegram backend server.

3) Sinch Library: The Sinch Library is an Android authentication library, targeting Android apps' developers [53]. We found that Sinch Library provides SMS authentication functionalities that not only uses the vulnerable `SMS Token` mechanism but also uses the `SMS Retriever` mechanism in a vulnerable way. Specifically, one of our "Confirmed" app is vulnerable because it uses a Sinch Library's function that internally uses the `SMS Token` API.

Furthermore, another app we found was vulnerable because it uses a Sinch Library's function that internally uses the `SMS Retriever` API incorrectly. Specifically, we found that the documentation of the Sinch Library *explicitly instructs developers to insert their hardcoded hashcode* as an argument of a function used to start the library-provided SMS authentication functionality. Then, the library sends the hashcode to the library-provided backend server [54]. This app-provided hashcode is used to generate the SMS OTP message, making the app vulnerable to the attack described in Section VII-A.

C. Responsible Disclosure and Developers' Response

For all apps identified as vulnerable in our study, we have contacted their developers. Telegram, KakaoTalk, and the developers of the Sinch Library acknowledged our findings. Both the developers of KakaoTalk and Telegram offered us bug bounties.

As of our submission, the Sinch Library developers have not released any update to fix the found issues yet. For KakaoTalk, the developers have updated its server-side implementation to no longer trust the hashcode received from the client-side mobile app. For Telegram, after our notification to the developers, the backend's code was quickly updated, not to include in the sent SMS the token used by the `SMS Token` API. Later, the app's code was updated [60], removing the usage of both the `SMS Token` API and the `SMS Retriever` API.

X. MITIGATION STRATEGIES

Throughout this paper, we have discussed many different proposals for secure mechanisms and APIs to implement SMS-related authentication functionality. However, each of these proposals has some security concerns and explores different trade-offs in the design space. In this section, we start by systematically enumerating all the "ideal features" that such a security mechanism and API should have.

We note that none of these features, when considered independently, is novel per se. For instance, the idea of using a dedicated channel for SMS OTP messages and the idea of filtering OTP messages based on their content was initially explored by Mulliner et al. [46]. However, this solution does not cryptographically link the delivered SMS with the app it targets.

Given that this research area is well explored and that there have been several proposals by both academia and industry (including Google's several attempts to provide such APIs) [9], [10], [12], [17], [23], [27], [28], [30], [46], [57], one may think that it is not possible to obtain a solution that achieves all these properties at the same time, and that there necessarily is some sort of trade-off. We believe that is not the case, and we offer a proposal that satisfies all these properties.

Ideal properties. An "ideal" API should implement (note: "W`n`" refers to Design Weakness #n): 1) the OTP-carrying SMS should be automatically forwarded to the appropriate app (no manual insertion), making W1 irrelevant; 2) the SMS should only be delivered to the proper app using an `SMS Retriever`-like method, making W6 and W7 irrelevant; 3) it should have appropriate documentation (addressing W5) and use proper crypto (addressing W9); 4) an OTP-carrying SMS should *never* reach the SMS inbox (addressing W2, W3, W4, and W8); 5) the user should be able to see the received messages, so to prevent that the presence of this functionality in a device can be exploited to silently send text messages to a phone number, which could result in financial damage; 6) it should be easily usable by existing apps on existing mobile devices.

Our proposal. We now present a safer variant of existing APIs that can be used by an app to receive OTP-carrying SMS messages and satisfy the ideal properties discussed above. Our proposed API relies on the assumption that the Android OS can established a secure communication channel with third party apps. Also, we assume that a system service can reliably identify the app it communicates with (and its signature). A system service can achieve this goal by using the `Binder.getCallingUid()` API. These assumptions are in-line with our threat model (Section II-A).

Our proposed API works similarly as the `SMS Retriever` API, but with the following modifications:

1) SMS OTP messages using this API must start with a precise prefix (e.g., "$<OTP>$").
2) Messages starting with the specific prefix, under no circumstances, are delivered to the SMS inbox.
3) Messages starting with the specific prefix can be visualized by the user using a dedicated system app.
4) Messages are delivered to the app whose hashcode is contained in the message itself.
5) The hashcode is computed as in the current `SMS Retriever` API, but its length is truncated to 38 base64 characters instead of 11 (ensuring 228 bits of entropy, as suggested by NIST guidelines).

We now explain how each of these modifications satisfies the ideal properties listed above. Condition 1 and Condition 2 enforce that SMS OTP messages are unequivocally flagged and never delivered to the SMS inbox. In this way, a malicious app, even if able to obtain the permission to read text messages, cannot access them. Condition 3 avoids that the presence of this functionality in a device can be exploited to silently send text messages (which can potentially cause financial damage) to a phone number. In fact, even if normal apps cannot access these messages, the user will always be notified of their arrival and able to see them. Condition 4 and Condition 5 makes this API deliver messages like the current `SMS Retriever` API. However, the longer hashcode ensures that a malicious app cannot obtain the same hashcode of a legitimate app. In turn, this property, thanks to the SHA256 pre-image resistance, guarantee that an app's backend server can be sure that the SMS will only be delivered to the app itself. Considering the required prefix, the typical length of an OTP, the length of the hashcode, and the fact that an SMS message can be long up to 160 characters (without incurring in any extra cost), the SMS OTP message still has about 100 characters freely usable by a developer.

We implemented the aforementioned system using ProVerif, and we verified that a malicious app, even if able to read the content of the normal inbox, cannot obtain unauthorized access to an OTP. Details of our ProVerif implementation and proof are provided in Appendix (Figure 14).

Compared to previously presented solutions, including the Google's implementation of the `SMS Retriever` API, our solution is able to achieve all the aforementioned "Ideal Properties." In addition, to the best of our knowledge, this is the first work formally verifying the properties of an API to access SMS OTPs.

Additional recommendations. As we explained in Section VII-A, developers have difficulties in computing the required hashcode, and this aspect leads them to mistakenly implement backend servers that, instead of hardcoding the correct hashcode, obtain it from the app. Therefore, we recommend that, in addition to implement the proposed API:

- The current documentation is updated to clearly state that, the backend server *should not* obtain the hashcode value from an app. Alternatively, in case supporting several legitimate client apps is needed, the server *must* verify that the hashcode sent by the app matches one of the legittimate apps' hashcodes.
- Developers should be offered a tool to easily compute the hashcode of a given app (starting from its APK file). The hashcode should also be shown in standard development tools, such as Android Studio and the Developer Console on the Google Play Store.

Note that the above recommendations are not part of our proposed defense mechanism, but they aim to prevent the misuses of the current APIs.

XI. Related Work

SMS-based authentication issues. Previous studies have identified a set of implementation issues [10], [45], which can result in a vulnerable SMS-based authentication scheme. For example, whether the OTP code generated with less entropy or with longer expiration time. In contrast, other works identify and summarize the different channels that can leak SMS OTP messages. Specifically, there exist vulnerabilities allowing the adversary to obtain the SMS OTP messages by compromising the telephony networks, including the SIM swapping attack [39], as well as the wireless interception attacks (e.g., SS7 network exploitation [21]). Other than these methods, a more straightforward way, as we studied in our research, is to obtain the message from the mobile device itself [18], [20], [32], [46]. Following this line of research, in earlier years, research highlighted various attack channels. This includes physical access to the device [46], mobile malware which steals the SMS OTP message by requesting the less-restricted SMS permissions [12], as well as phishing attacks [9], [13], [14] that can

get the SMS OTP code from the user input. Different from prior research, we systematically studied the practical ways an adversary can use to obtain the SMS OTP message through a malicious app running on a victim's device, dealing with the various new features introduced in modern mobile operating systems. In addition, our identified vulnerabilities in the automatic SMS APIs follow the observation of previous research [10], that is, any device-public controlled information (e.g., the content of accessible SMS messages) should not be used in any authentication scheme.

Understanding the security implications of SMS OTP messages. Another line of research focused on better understanding the real-world security implications of SMS OTP authentication issues [23], [35], [45], [63]. Specifically, AUTH-EYE [45] proposed a fully automated approach to identify and detect the implementation flaws of apps using the SMS OTP authentication scheme on a large scale. Their analysis focuses on whether the SMS OTP code is securely generated (e.g., the OTP randomness, length) and verified (e.g., allowed retry attempts, renewal interval). The results highlighted that 98.5% of apps violate different security rules during the SMS OTP authentication scheme.

Yoo et al. [63] studied the vulnerable SMS OTP implementations for bank apps in South Korea, while Gutmann et al. [31] discussed the security risks of the security code autofill mechanism in iOS and macOS. Besides, Fahl et al. [19] demonstrated that a malicious app can monitor the clipboard and steal passwords during a copy-and-paste user interaction. This attack vector could also be used for stealing SMS OTP code. In contrast, our work focuses primarily on how a local attacker can obtain SMS OTP code by exploiting weaknesses in the implementations of apps and mobile operating systems.

Mitigation and defense mechanisms. As countermeasures, some research proposed different defense mechanisms against attacks in SMS OTP authentication [33], [40], [44], [46], [56]. Among them, the mechanisms proposed by Mulliner et al. [46] (i.e., using a dedicated channel for SMS OTP delivery) is similar to what implemented by Google in the modern APIs, in which our work highlights several implementation issues and pitfalls. Besides, DroidPAD [44] proposed a heuristic-based approach for identifying malicious apps based on their pattern when reading SMS messages. As more fundamental solutions, Hamdare et al. [33] proposed encryption-based mechanisms to secure the process of OTP transmission, while CodeTracker [40] employs dynamic taint analysis to track and protect the flow of SMS OTP messages runtime using pre-defined policies. Moreover, TrustOTP [56] used TrustZone to isolate the OTP code at the mobile OS-level. This system provides a security guarantee to the integrity of the OTP code even when the system is compromised by attackers. Unfortunately, these approaches have not yet been adopted by mobile OSes in practice, due to various limitations and requirements. For instance, the usage of TrustZone [6] might not be feasible in all mobile devices, and it requires integrating SMS reading capabilities within the trusted computing base. In comparison, our proposed mechanism extends the existing authentication mechanisms and does not rely on any hardware feature.

XII. Conclusion

In this paper, we conduct the first in-depth, systematic study on the specific ways in which a malicious local app can obtain unauthorized access to SMS OTP messages in modern mobile operating systems. Our research identified a set of new attack channels that are primarily caused by newly introduced mechanisms. While these mechanisms were developed to allow more usable and safer SMS-based authentication, in reality, they introduce new attack opportunities. To better understand the real-world impact of these security issues, we performed both user-studies and a large-scale measurement study over 140,586 apps. Our measurement found 36 apps (sharing hundreds of millions of installations) that are vulnerable to the identified attacks, including the popular messaging apps Telegram and KakaoTalk. Furthermore, we provided suggestions on how to mitigate this threat to both app developers, as well as OS vendors.

Acknowledgments

We are grateful to our shepherd, William Enck, and to the anonymous reviewers for their insightful feedback and suggestions.

This material is based upon work supported in part by the NSF under Grant No. NS-1849803. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "Xposed Installer," http://repo.xposed.info/module/de.robv.android.xposed.installer, 2017.

[2] "Marvel - the design platform for digital products," https://marvelapp.com/, 2020.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR)*, 2016.

[4] Amazon, "Amazon mechanical turk," https://www.mturk.com/, 2020.

[5] APKPure, "Apkpure, download apk free online." https://apkpure.com/, 2020.

[6] ARM, "ARM TrustZone," https://www.arm.com/products/security-on-arm/trustzone, 2017.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[8] M. Atkinson, "An analysis of android app permissions," https://www.pewresearch.org/internet/2015/11/10/an-analysis-of-android-app-permissions/, 2015.

[9] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[10] A. Bianchi, E. Gustafson, Y. Fratantonio, C. Kruegel, and G. Vigna, "Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[11] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "Proverif 2.00: Automatic cryptographic protocol verifier," *User Manual and Tutorial*, 2018.

[12] A. D. P. Center, "Additional requirements for the use of specific permissions." https://play.google.com/about/privacy-security-deception/permissions/, 2019.

[13] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks," in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2014.

[14] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: Automated generation of android phishing apps," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[15] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[16] Q. Dang, "Recommendation for applications using approved hash algorithms," https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf, 2020.

[17] P. A. Dev, "No more sms and call log permissions, now what?" https://proandroiddev.com/no-more-sms-call-log-permissions-now-what-9b8226de7827, 2019.

[18] A. Dmitrienko, C. Liebchen, C. Rossow, and A. Sadeghi, "On the (In)Security of Mobile Two-Factor Authentication," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[19] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, you, get off of my clipboard," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2013.

[20] A. P. Felt and D. Wagner, "Phishing on Mobile Devices," in *Proceedings of the IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*, 2011.

[21] T. Fox-Brewster, "Watch as hackers hijack whatsapp accounts via critical telecoms flaws." https://www.forbes.com/sites/thomasbrewster/2016/06/01/whatsapp-telegram-ss7-hacks/, 2016.

[22] Y. Fratantonio, C. Qian, P. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[23] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan, "The password reset mitm attack," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[24] Google, "Android-behavior-changes," https://developer.android.com/about/versions/10/behavior-changes-all, 2020.

[25] ——, "Android-pendingintent," https://developer.android.com/reference/android/app/PendingIntent, 2020.

[26] ——, "Use binder and messenger interfaces," https://developer.android.com/training/articles/security-tips#IPC, 2020.

[27] A. D. Guide, "Android sms manager," https://developer.android.com/reference/android/telephony/SmsManager.html#createAppSpecificSmsToken, 2020.

[28] ——, "Automatic sms verification with the sms retriever api," https://developers.google.com/identity/sms-retriever/, 2020.

[29] ——, "Notificationlistenerservice," https://developer.android.com/reference/android/service/notification/NotificationListenerService, 2020.

[30] ——, "One-tap sms verification with the sms user consent api," https://developers.google.com/identity/sms-retriever/user-consent/overview, 2020.

[31] A. Gutmann and S. J. Murdoch, "Taken out of context: Security risks with security code autofill in ios & macos," in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2019.

[32] K. Hamandi, A. Chehab, I. Elhajj, and A. Kayssi, "Android SMS Malware: Vulnerability and Mitigation," in *Proceedings of the Advanced Information Networking and Applications (AINA)*, 2013.

[33] S. Hamdare, V. Nagpurkar, and J. Mittal, "Securing sms based one time password technique from man in the middle attack," *arXiv preprint arXiv:1405.4828*, 2014.

[34] Y. Z. X. Jiang and Z. Xuxian, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[35] Z. Jorgensen, J. Chen, C. S. Gates, N. Li, R. W. Proctor, and T. Yu, "Dimensions of risk in mobile applications: A user study," in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[36] D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing attack surfaces for intra-application communication in android," in *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[37] R. Küsters and T. Truderung, "Using proverif to analyze protocols with diffie-hellman exponentiation," in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, 2009.

[38] S. Landau, "Find me a hash," *Notices of the AMS*, vol. 53, no. 3, 2006.

[39] K. Lee, B. Kaiser, J. Mayer, and A. Narayanan, "An empirical study of wireless carrier authentication for sim swaps."

[40] J. Li, Y. Ye, Y. Zhou, and J. Ma, "Codetracker: A lightweight approach to track and protect authorization codes in sms messages," *IEEE Access*, 2018.

[41] L. Li, A. Bartel, J. Klein, and Y. Le Traon, "Automatically exploiting potential component leaks in android applications," in *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2014.

[42] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," in *Proceedings of the third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*, 2014.

[43] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[44] W. Luo, S. Xu, and X. Jiang, "Real-time Detection and Prevention of Android SMS Permission Abuses," in *Proceedings of the Security in Embedded Systems and Smartphones Workshop: Preface (SESP)*, 2013.

[45] S. Ma, R. Feng, J. Li, Y. Liu, S. Nepal, E. Bertino, R. H. Deng, Z. Ma, and S. Jha, "An empirical study of sms one-time password authentication in android apps," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2019.

[46] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert, "SMS-Based One-Time Passwords: Attacks and Defense," in *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2013.

[47] G. Play, "Kakaotalk on google play," https://play.google.com/store/apps/details?id=com.kakao.talk, 2020.

[48] ——, "Telegram on google play," https://play.google.com/store/apps/details?id=org.telegram.messenger, 2020.

[49] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2019.

[50] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards Discovering and Understanding Task Hijacking in Android," in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2015.

[51] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. R. Weippl, "Guess who's texting you? evaluating the security of smartphone messaging applications." in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[52] W. L. Security, "Malware sidesteps google permissions policy with new 2fa bypass technique," https://www.welivesecurity.com/2019/06/17/malware-google-permissions-2fa-bypass/, 2020.

[53] Sinch, "Introduction," https://developers.sinch.com/docs/verification-introduction, 2020.

[54] ——, "The verification process," https://developers.sinch.com/docs/verification-android-the-verification-process#sms-verification, 2020.

[55] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android." in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[56] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[57] A. D. Support, "Declare permissions for your app," https://support.google.com/googleplay/android-developer/answer/9214102?hl=en, 2020.

[58] A. Support, "Automatically fill in sms passcodes on iphone," https://support.apple.com/guide/iphone/automatically-fill-in-sms-passcodes-on-iphone-iphc89a3a3af/ios, 2020.

[59] Telegram, "Keep Calm and Send Telegrams!" https://telegram.org/blog/15million-reuters, 2016.

[60] Telegram, "Update to 5.10.0," https://github.com/DrKLO/Telegram/commit/53e04b55fbb665fcb3859f54f15ae203179a88c2#diff-1fdb2a1cb7f751eeb5964c9d9c3e6957, 2019.

[61] T. Verge, "Android messages now makes it really easy to copy two-factor codes," https://www.theverge.com/2018/5/11/17345016/android-messages-copy-two-factor-codes-update, 2018.

[62] Wikipedia, "Kaokaotalk," https://en.wikipedia.org/wiki/KakaoTalk, 2020.

[63] C. Yoo, B.-T. Kang, and H. K. Kim, "Case study of the vulnerability of otp implemented in internet banking systems of south korea," *Multimedia Tools and Applications*, vol. 74, no. 10, pp. 3289–3303, 2015.

[64] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

APPENDIX

Figure 10 shows the questionnaire used to evaluate users' understanding of the SMS permission. Figure 11 and Figure 12 show the simulated scenarios of a phishing app requesting SMS OTP code with the android One-Tap SMS and iOS Autofill feature, respectively.

Figure 13 shows ProVerif output run on a model encoding the `SMS Token` API behavior. The output indicates the existence of a way to access the OTP code, which coincides with the attack we explained. Figure 14 shows ProVerif output run on a model encoding the improve `SMS Retriever` API, which we propose to implement. The output indicates the there is no way in which an attacker can access the OTP code.

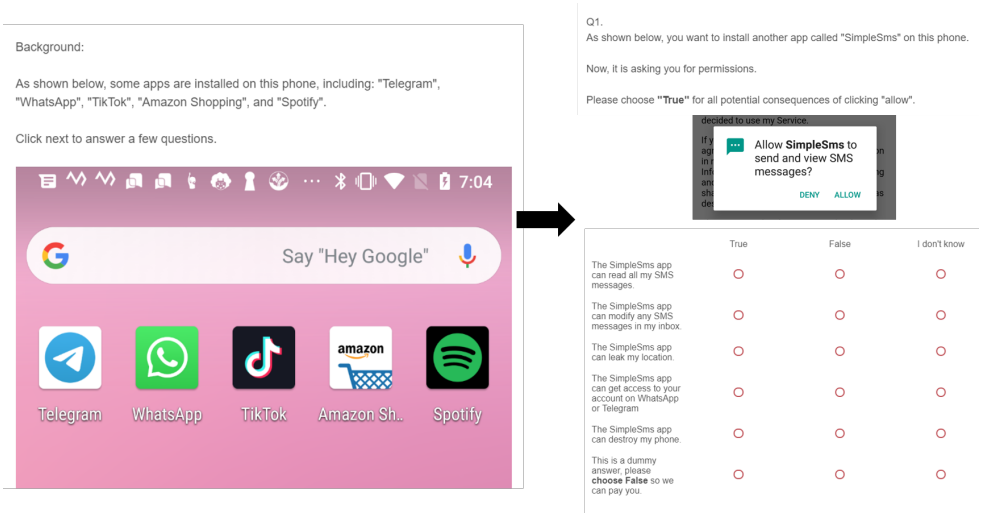


Fig. 10. Designed questionnaire for asking user understanding about SMS permission request.

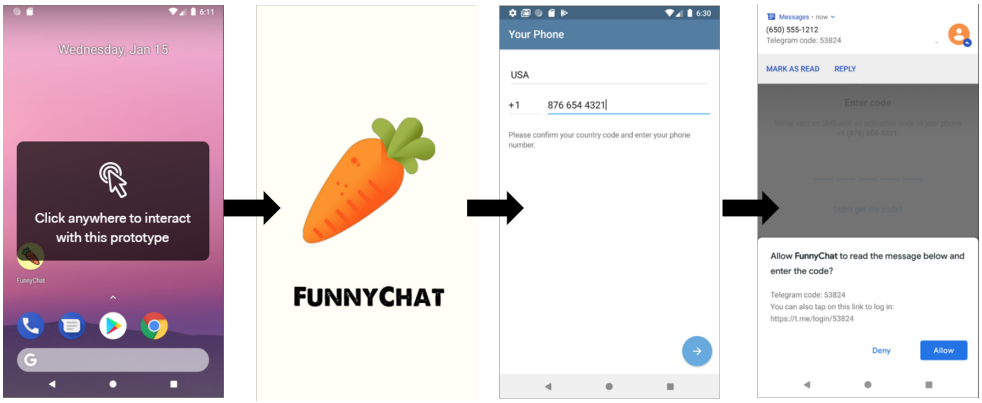


Fig. 11. Simulated phishing scenario with the Android One-Tap SMS feature.

Fig. 12. Simulated phishing scenario with the iOS AutoFill OTP code prompt feature.

```
Process:
{1} in(client_to_server_channel, auth_request: OTP_REQUEST);
{2} let number_30: NUMBER = get_number_from_request(auth_request) in
{3} let token_31: TOKEN = get_token_from_request(auth_request) in
{4} if (number_30 = victim_number) then
{5} let auth_sms: SMS_MSG = sms(secret_otp,token_31) in
{6} out(server_to_victim_channel, auth_sms);
{7} if (get_calling_app(read_token_from_sms(auth_sms)) = malicious_app) then
    {8} out(broadcast_to_malicious_app_channel, auth_sms)
else
    {9} if (get_calling_app(read_token_from_sms(auth_sms)) = official_app) then
        {10} out(broadcast_to_official_app_channel, auth_sms)
    else
        {11} if (get_calling_app(read_token_from_sms(auth_sms)) = any_other_app) then
            {12} out(broadcast_to_any_other_app_channel, auth_sms)


-- Query not attacker(secret_otp[])
Completing...
Starting query not attacker(secret_otp[])
goal reachable: attacker(secret_otp[])
1.  The attacker initially knows victim_number[].
attacker(victim_number[]).
2.  The attacker initially knows malicious_app[].
attacker(malicious_app[]).
3.  By 2, the attacker may know malicious_app[].
    Using the function generate_token the attacker may obtain generate_token(malicious_app[]).
attacker(generate_token(malicious_app[])).
4.  By 3, the attacker may know generate_token(malicious_app[]).
    By 1, the attacker may know victim_number[].
    Using the function request, the attacker may obtain request(generate_token(malicious_app[]),victim_number[]).
attacker(request(generate_token(malicious_app[]),victim_number[])).
5.  The message request(generate_token(malicious_app[]),victim_number[]) that the attacker may have by 4 may be received at input {1}.
So the message sms(secret_otp[],generate_token(malicious_app[])) may be sent to the attacker at output {8}.
attacker(sms(secret_otp[],generate_token(malicious_app[]))).
6.  By 5, the attacker may know sms(secret_otp[],generate_token(malicious_app[])).
    Using the function read_otp_from_sms the attacker may obtain secret_otp[].
attacker(secret_otp[]).

Could not find a trace corresponding to this derivation.
RESULT not attacker(secret_otp[]) cannot be proved.
```

Fig. 13. ProVerif verification process for SMS Token.

```
Process:
{1} in(client_to_server_channel, request(number: NUMBER));
{2} if (number = victim_number) then
(
    {3} let auth_sms: SMS_MSG = sms(otp_prefix,secret_otp,hash_of_GoodApp) in
    {4} out(server_to_victim_channel, auth_sms);
    {5} if (read_prefix_from_sms(auth_sms) = otp_prefix) then
        {6} out(broadcast_to_OtpInbox_channel, auth_sms)
    else
        {7} if (read_prefix_from_sms(auth_sms) = no_prefix) then
        {8} out(broadcast_to_Inbox_channel, auth_sms);
        {9} if (read_suffix_from_sms(auth_sms) = hash_of_BadApp) then
            {10} out(broadcast_to_BadApp_channel, auth_sms)
        else
            {11} if (read_suffix_from_sms(auth_sms) = hash_of_GoodApp) then
            {12} out(broadcast_to_GoodApp_channel, auth_sms)
) | (
    {13} let trivial_sms: SMS_MSG = sms(no_prefix,sms_text,no_suffix) in
    {14} out(server_to_victim_channel, trivial_sms);
    {15} if (read_prefix_from_sms(trivial_sms) = otp_prefix) then
        {16} out(broadcast_to_OtpInbox_channel, trivial_sms)
    else
        {17} if (read_prefix_from_sms(trivial_sms) = no_prefix) then
        {18} out(broadcast_to_Inbox_channel, trivial_sms);
        {19} if (read_suffix_from_sms(trivial_sms) = hash_of_BadApp) then
            {20} out(broadcast_to_BadApp_channel, trivial_sms)
        else
            {21} if (read_suffix_from_sms(trivial_sms) = hash_of_GoodApp) then
            {22} out(broadcast_to_GoodApp_channel, trivial_sms)
)


-- Query not attacker(sms_text)
Completing...
Starting query not attacker(sms_text)
goal reachable: attacker(sms_text)
1. Using the function victim_number the attacker may obtain victim_number.
attacker(victim_number).
2. By 1, the attacker may know victim_number.
    Using the function request the attacker may obtain request(victim_number).
attacker(request(victim_number)).
3. The message request(victim_number) that the attacker may have by 2 may be received at input {1}.
    So the message sms(no_prefix,sms_text,no_suffix) may be sent to the attacker at output {18}.
attacker(sms(no_prefix,sms_text,no_suffix)).
4. By 3, the attacker may know sms(no_prefix,sms_text,no_suffix).
    Using the function read_text_from_sms the attacker may obtain sms_text.
attacker(sms_text).

Could not find a trace corresponding to this derivation.
RESULT not attacker(sms_text) cannot be proved.


-- Query not attacker(secret_otp)
Completing...
Starting query not attacker(secret_otp)

RESULT not attacker(secret_otp) is true.
```

Fig. 14. ProVerif verification process for our proposed secure SMS authentication scheme.