Wireless Networks (2024) 30:6451–6464
https://doi.org/10.1007/s11276-023-03455-w

App-based detection of vulnerable implementations of OTP SMS APIs in the banking sector

Amador Aparicio[1] · M. Mercedes Martínez-González[1] · Valentín Cardeñoso-Payo[1]

Accepted: 10 July 2023 / Published online: 22 July 2023

Abstract
Two Factor Authentication (2FA) using One Time Password (OTP) codes via SMS messages is widely used. In order to improve user experience, Google has proposed APIs that allow the automatic verification of the SMS messages without the intervention of the users themselves. They reduce the risks of user error, but they also have vulnerabilities. One of these APIs is the SMS Retriever API for Android devices. This article presents a method to study the vulnerabilities of these OTP exchange APIs in a given sector. The most popular API in the sector is selected, and different scenarios of interaction between mobile apps and SMS OTP servers are posed to determine which implementations are vulnerable. The proposed methodology, applied here to the banking sector, is nevertheless simple enough to be applied to any other sector, or to other SMS OTP APIs. One of its advantages is that it proposes a method for detecting bad implementations on the server side, based on analyses of the apps, which boosts reusability and replicability, while offering a guide to developers to prevent errors that cause vulnerabilities. Our study focuses on Spain's banking sector, in which the SMS Retriever API is the most popular. The results suggest that there are vulnerable implementations which would allow cybercriminals to steal the users SMS OTP codes. This suggests that a revision of the equilibrium between ease of use and security would apply in order to maintain the high level of security which has traditionally characterized this sector.

Keywords SMS · OTP · 2FA · Android · Security · Banking · Apps

1 Introduction

Directive (EU) 2015/2366 of the European Parliament and of the Council of 25 November 2015 on payment services in the internal market, also known as Payment Services Directive (PSD2) establishes the obligation for the banking sector to use additional strong customer authentication factors, such as SMS One Time Passwors (SMS OTP) codes, to ensure the safety of online transactions [1].

2FA systems permit the user's identity to be verified by sending an OTP code to their device. Only the user who receives the OTP code can verifytheir identity. This allows the user's identity to be confirmed through the mobile device, but it also means that the apps have to be able to access these OTP codes through the SMS service. Therefore, a bad implementation of 2FA in the apps means that this authentication scheme becomes vulnerable [2].

In this article, a study is presented that analyzes the handling of the SMS OTP codes by the APIs in the banking sector apps. It includes the methods followed to design the study and to apply it to this sector. The datasets generated in this study are available with this article. Given that it is not usual to have access to the SMS OTP servers used by the banking apps, we have focused our proposal on looking for bad implementations of the automatic SMS verification API in the apps. This provides generality and allows the methodology to be applied to cases, such as ours, when there is no access to the server of the OTP codes. Our hypothesis is that it is possible to infer the risks a server is exposed to by detecting bad implementations on the apps side with a static analysis.

✉ Amador Aparicio
amador@infor.uva.es

M. Mercedes Martínez-González
mercedes@infor.uva.es

Valentín Cardeñoso-Payo
valen@infor.uva.es

[1] Departamento de Informática, Universidad de Valladolid, P.o de Belén, 15, 47011 Valladolid, Spain

The rest of the article is organized as follows: Sect. 2 presents the related work; Sect. 3 shows the APIs that handle the SMS messages used in the banking sector; Sect. 4 presents a general methodology, applicable to any sector, for detecting bad implementations of the SMS Retriever API; Sect. 5 presents the results obtained after applying the methodology to the most commonly used banking sector apps in Spain; and Sect. 7 sets out the conclusions. The Data Availability section includes the references where the datasets generated can be downloaded.

2 Related work

OTP SMS based authentication systems provide an additional security layer which ensures that only the owner of the device whose phone number is registered on the OTP server can access the information on the server and interact with it in order to carry out a transaction. For this reason, OTPs are widely used by mobile apps as 2FA systems [3–7]. The banking sector also uses them in its electronic banking applications.

Research related to *malware*[1] in mobile apps shows that there is a large number of variants targeting the banking sector [8–11]. These variants are aimed at accessing and stealing OTP codes intended for the authorization of banking transactions. Having these codes would allow fraudulent transactions to be carried out. Given that the banking sector represents the largest segment of cyber attacks, it is of the utmost importance to have mechanisms for the early detection of misuse by developers of SMS OTP code management APIs that could impact the OTP code servers used by apps.

On the other hand, research has shown that SMS OTP code exchange systems have vulnerabilities that can be exploited by malware [2, 3, 12]. Vulnerabilities can be due to several causes. Using a resource such as the SMS inbox opens up the possibility that any app that has SMS read permissions can obtain the SMS OTP intended for another app. This vulnerability is related to the design of the APIs [2, 6, 13–15]. However, there are other vulnerabilities, due to developers making bad implementations of the SMS OTP code API [2, 13]. It is in the latter case where developers have the greatest ability to prevent risks. The methods used to detect vulnerabilities in OTP-based 2FA authentication systems include the use of dynamic analysis and static analysis. The first approach is to perform a dynamic analysis to monitor the exchange of messages between apps and OTP code servers [12, 16]. The limitation of this approach is that they require a specific monitoring app to be installed on the same mobile device on which the app being analyzed is running. This poses a risk to the privacy of its user that may cause reluctance, which means it would be difficult to put into practice in real scenarios. This is even more critical in the case of banking, since having a bank account with the entity that owns the OTP server would also be required, something which seems difficult to obtain for external researchers and auditors. Another important limitation of this approach in real scenarios comes from the use of a secure channel for message exchange. A secure channel implies that it is actually impossible to know the content of the messages exchanged, since they travel encrypted.

Other works of research propose strengthening the SMS OTP-based authentication schemes [17–26]. Applying symmetric and public key cryptography to protect the SMS OTP is one of the proposed measured. However, these solutions also have vulnerabilities. Both schemes require the apps receiving the OTP code to know the key to decrypt the OTP. There is therefore a risk that the key may be exposed because it is present in the source code of the apps. In addition, once the SMS OTP has been decrypted, it goes to the device's SMS tray. Once in this tray, any app with SMS reading permissions could have access to the OTP code if the SMS OTP code management API does not remove it from the SMS message tray. This problem has been mentioned above as part of the vulnerabilities due to API design.

Finally, as an alternative to the use of SMS OTP as 2FA in sectors such as electronic banking, Aloul, Fadi and Zahidi, Syed and El Hajj, Wassim [27] propose using the mobile device as an OTP generator with unique factors that identify the device, such as IMEI.[2] and IMSI.[3]. However, access by an app to the IMEI, IMSI values needs the explicit consent of the user [8]. This again introduces limitations to the ease of use that the banking industry has sought through OTP exchange APIs. Granting permissions, or accessing values such as IMEI or IMSI, are operations that most electronic banking users are not comfortable with.

[1] Any program or code that harms or makes undesired changes to a computer system. These programs can steal confidential information, damage files or systems and even control a device without the user's knowledge or permission.

[2] *International Mobile Equipment Identity* Identifies users with device.

[3] *Mobile Subscriber Identity* Unique number associated with GSM and Universal Mobile Telecommunications System (UMTS) cell phone networks. It is stored on the cell phone card (SIM).

3 One-time password used as 2FA

2FA schemes with OTP are a widely used method in electronic banking [1, 17]. The OTP is a unique, time-limited token that is sent to a user device by the OTP server. Knowing a user's login password is not enough to access their account, the OTP that has been sent to their device is also required. That is, it is necessary to be in possession of device. Figure 1 shows the basic authentication process using an OTP.

To free users from this interaction, consisting of collecting the OTP from the SMS tray and entering it in the app, Google developed the SMS OTP management APIs [2, 6]. They represent an improvement in the ease of use over the authentication scheme of Fig. 1. In Fig. 2, the operation of the SMS Retriever API is shown, highlighting in blue the key differences with the scheme in Fig. 1. The SMS Retriever API is the one studied in Sect. 5.2.

With this API, for the server to know which device and which app it has to send the SMS OTP code to, it needs to know the telephone number of the mobile device and an alphanumeric chain (*hash*) that identifies the receiving app of the SMS OTP code. When the server sends the SMS OTP code to the device that made the request, it also sends the *hash* that identifies the app which must receive the SMS OTP code. This *hash* allows the device's operating system to deliver the SMS OTP code to the receiving app only, and no other [2, 6]. The user is not required to get the OTP code.

4 Proposal

A generic working methodology is proposed that allows a vulnerability study to be applied to any sector where the exchange of OTP codes by SMS as 2FA is used. In this work, it has been applied to a specific sector, the banking sector. The methodology is developed in Sect. 4.1.

Part of this methodology is the design of a set of scenarios that cover the possibilities that can occur in an OTP exchange interaction between client and server apps when there is a malicious app that tries to pervert this interaction to appropriate the OTP code. These scenarios are presented in Sect. 4.2. This section characterizes each type of actor and the interaction associated with each scenario. In this way, we can unequivocally identify the risk scenarios and, consequently, know what type of actor is associated with these risks. Finally, in Sect. 4.3, we associate the relevant actions of these interactions with a set of patterns, recognizable in the source code of the apps. In this way, we associate each type of app with the set of patterns that it presents, and that we will be able to recognize through a static analysis of the apps under study.

We start from the premise that benign apps and the corresponding servers are designed and developed within the same ecosystem, which in the case of the banking sector means that it is the same information technology team that deals with both developments. It is logical to assume, in this case, that the possible errors in the use of the OTP exchange APIs have had repercussions on both sides of the interaction: backend server and client app. Our hypothesis is therefore that *it is possible to recognize the servers that include design errors by performing a static analysis of the apps that were developed within the same ecosystem*. This allows an audit of risks in vulnerable scenarios with an analysis of the apps, without the need to access the servers.

4.1 The proposed method

Figure 3 graphically represents the methodology used. It is divided into the following blocks: selection of the category set (and set of apps) under study, selection of the SMS OTP code exchange method (API), preparation of the methods to be applied to characterize the possible uses of the API selected, and the consistent evaluation of its application to the apps selected in the first phase. Each of these stages contains a series of steps, which are explained below. The rectangles represent actions and the arrows the results of the actions.

The breakdown of each of these phases into a sequence of steps leads to the proposal detailed in Fig. 3.

The orientation of this design is to focus the study on those apps that account for most of the market in the sector and on the most popular method, that is, to focus during the following steps on an authentication method (API). The breakdown of each of these phases into a sequence of steps leads to the proposal detailed in Fig. 3. The first two steps are the selection of a repository, followed by the selection of a category of apps. We have worked with the official market of Android, that is, Google Play. For the selection of a category we propose using the metadata provided by the market to select the apps under study. Next, in step 3, the most popular ones in the repository are selected. Subsequently, in step 4, a **static analysis** is carried out to know which automatic verification method of SMS messages is used in each of them. The most popular one is filtered in step 5. This implies a second filter: only the apps that use this API will be used in the evaluation. Once the method or API has been decided, its **scenarios** are designed and the behavior patterns that characterize each of them are identified. The scenarios collect the possible interactions during an OTP SMS request This is step 6. The next step is **pattern design**, to formalize these actions as source code patterns that can be recognized in apps. A well-implemented app is one whose actions correspond to a good use

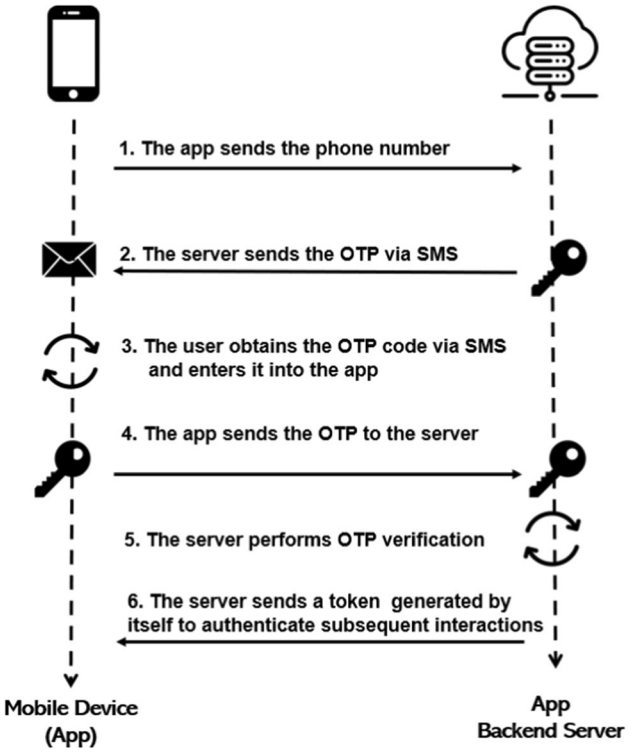


Fig. 1 A typical scenario using OTP SMS for 2FA

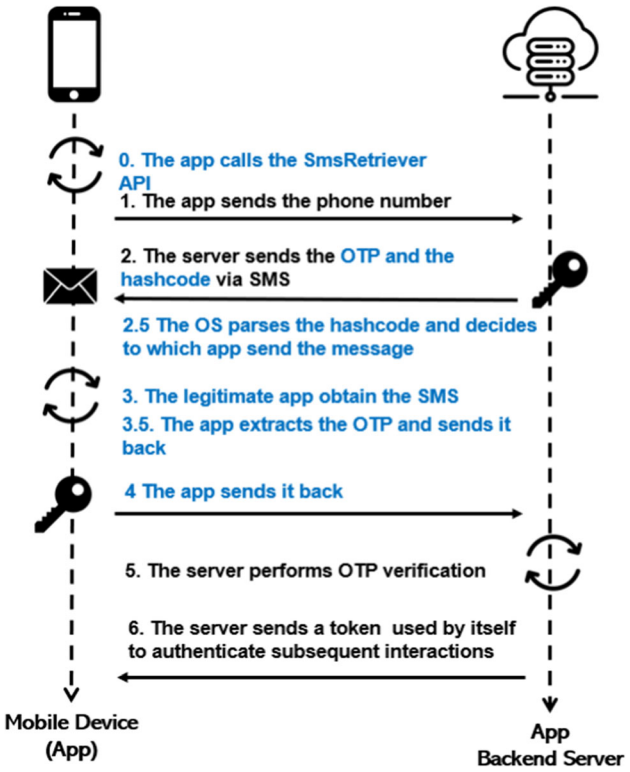


Fig. 2 Authentication process for *SMS Retriever*

of the API, according to the guidelines provided by Google in [2] for the verification method under study. A poorly implemented or failed app is one that deviates from this ideal behavior. They can be characterized with the **static analysis** carried out in step 7, during which the code patterns that characterize each of them are searched. The Evaluation phase starts with this step. The conclusions are obtained when the last step, *results analysis* is performed.

4.2 Scenarios

After the analysis of the apps carried out in step 5 of the methodology, the most widely used OTP SMS code automatic management API is obtained. In our case, the most used API is *SMS Retriever*. The normal interaction between an app and the corresponding server with this API, when there are no malicious apps trying to steal the OTP, is the one shown in Fig. 2 of Sect. 3.

A set of scenarios is proposed to obtain the patterns that characterize the interactions of banking apps that request an OTP. The situation with which we have worked is one in which a malicious app, installed on the same device as the benign app, tries to obtain the OTP code. Since both the benign app and the malicious app are installed on the same mobile device, they share the same phone number. To facilitate the understanding of the scenarios, the intervention of an additional app is proposed, an attacking app, which acts in coordination with the malicious app. This app can be the same malicious app, or some other app or element installed on a different device, depending on the type of malware with which the malicious app is associated. However, what is relevant to our investigation are the messages it exchanges with the server, the device from which it intervenes being irrelevant. There are six possible scenarios. They compile the set of possible interactions between apps and bank servers.

In these scenarios, the intervention of four actors is proposed: 1) The OTP code server; 2) The benign app developed to interact with the server; 3) A malicious app whose goal is to receive the OTP code; and 4) An attacking app, part of the same ecosystem as the malicious app, that tries to cause the OTP code to be sent to the malicious app. For both the server and the benign app, two types are considered: the well-developed server/app, that is, following the recommendations given by Google for the correct use of the API [7], as reflected in Fig. 2, and the poorly developed server/app, that is, deviating from Google's recommendations. We refer to well-developed servers and apps as S_B and APP_B, respectively. In the case of badly

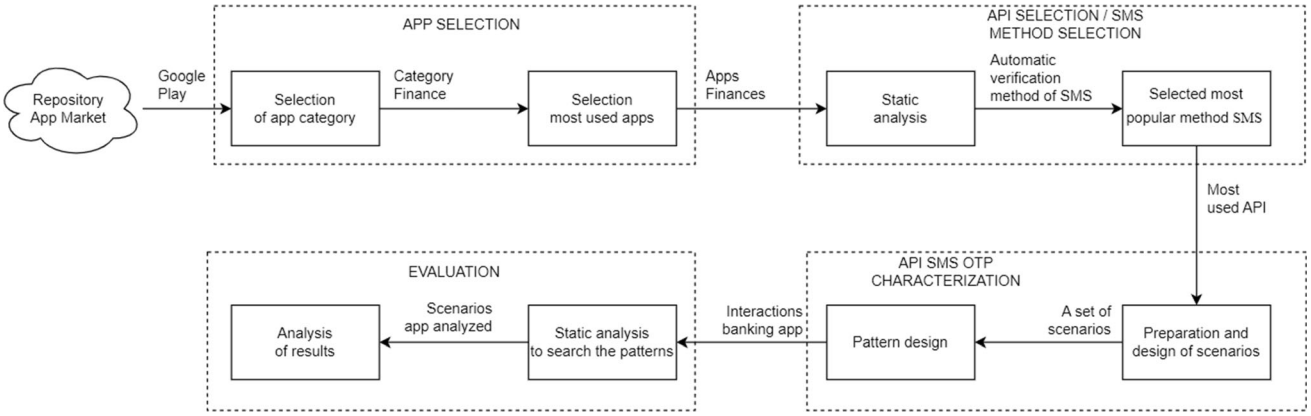


Fig. 3 Graphic representation of the methodology

Table 1 Actors that are related in the different scenarios

Mobile apps (frontend)	OTP SMS code servers (backend)
APP_B, APP_F, APP_M, APP_A	S_B, S_F

Table 2 Actors/scenarios that are related in the different scenarios

Actor	Legend
APP_B	App well (B) programmed
APP_F	Failed app (F), poorly implemented
APP_M	Malicious app (M)
APP_A	Attacker app (A)
S_B	OTP SMS server well (B) implemented
S_F	SMS OTP server failed (F), poorly implemented

designed or failed ones, we call them S_F and APP_F. As far as the attacking app is concerned, we refer to it as APP_A. Table 1 classifies the actors based on their location, mobile device (frontend) or server (backend). Table 2 compiles the name and legend for each of the actors presented above.

Bad implementations of banking apps and servers correspond to the vulnerabilities detected by researchers when the *SMS Retriever* API was studied [2, 13]. As for the app, a bad implementation consists of sending the app identifier (*hash*) to the SMS OTP server. After studying the source codes of various applications, it has become clear that this may happen because the hash appears in the source code of the app or because the app itself generates it dynamically. As for the server, a bad implementation consists of accepting the reception the *hash* of the app, which indicates that the server did not generate it or did not have it registered. A well designed server stores the *hashcode*s that identify the apps to which OTP codes are beign sent. That is, it does not accept *hashcode*s if OTP requests are sent. In turn, an app designed as indicated by the recommendations for the use of this API will only send the phone number in its request (Fig. 4).

Table 3 summarizes the scenarios, the actors that participate in each of them, and the result of the interaction in terms of OTP theft. All the possibilities have been considered, for the aim of completeness, even those in which the benign app and the server are not designed within the same ecosystem, despite we insist this is not what should be expected in a banking environment. These scenarios are scenarios 2 and 3, in Figs. 5 and 6 respectively. The scenario 1 covers the interaction between a well designed app, APP_B, and a well designed server, S_B. Its interaction is shown in Fig. 4. What is relevant is that a well designed server will always send the OTP to the app whose hashcode it stores. This guarantees that no other application will be able to get access to it. This is the situation in scenario 1, 3 and 5 (indeed, the interaction in scenario 5 is very similar to the interaction in scenario 3: identical step 2 onwards). As for the other scenarios, scenarios 2, 4, and 6, where the poorly designed server intervenes, the success of an OTP theft attack will depend on the other actors that interact with the server. In scenario 2 it fails. The well designed app, APP_B, does not send a hashcode. Therefore, the server does not know what app it should send the OTP to. As a consequence, it does not send anything.

It is interesting to pay attention to scenario 4, in Fig. 7, which looks ideal for the theft attack to succeed. However, it does not. Despite the server sending the OTP to the app whose hashcode it receives with the OTP request, this hashcode is the one of the benign app, APP_F, not the one of the malicious one, APP_M. Therefore, only the benign app, APP_F, receives the hashcode. The only opportunities for the attack to succeed would be vulnerabilities in the hashcode generation, or in the SMS OTP API, such as those referred to in [2, 13]. However, we do not go into these in any detail, as these types of vulnerabilities are beyond the scope of this study.

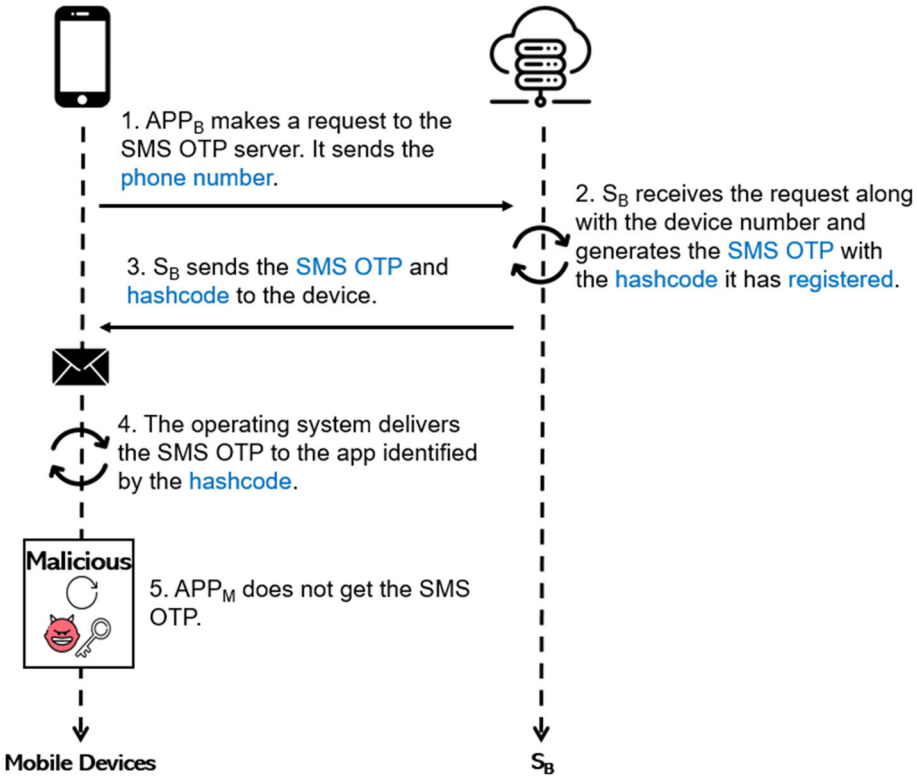


Fig. 4 Interaction diagram for scenario 1

Table 3 Scenarios, actors and results of the attempted theft of the OTP

Scenario	Actors	Theft OTP
1	APP_B, APP_M, S_B	No
2	APP_B, APP_M, S_F	No
3	APP_F, APP_M, S_B	No
4	APP_F, APP_M, S_F	No
5	APP_A, S_B	No
6	APP_A, S_F, APP_M	Yes

The only scenario in which the theft attack is successful is scenario 6, described in Fig. 8, the one in which the attacker app, APP_A, intervenes. A poorly designed server, S_F, receives a request from APP_A, which refers to the hashcode of its collaborator, APP_M. In such a case, the server will send the OTP to the malicious app, APP_M. It is worth noting the difference with scenario 5, in which the APP_A also intervenes. However, in this scenario, a well designed server guarantees that the attack fails.

4.3 Patterns

The patterns characterize benign apps and they can be obtained from Google's instructions for the use of the API and the scenarios designed in the previous stage.

In this first design phase, the patterns are independent of the language or platform used to develop the apps, they are behavior patterns (*behavioral*). Section 4.3.1 is dedicated to them. In the next phase, we associate these behavior patterns with code patterns that can be recognized in an app's source code. First of all, in Sect. 4.3.2, the patterns that allow us to know which APIs are used by the apps are presented. These patterns are applied in step 4 of the methodology. Next, in Sect. 4.3.3, the characteristic patterns of the *SMS Retriever API* that should appear in any application are dealt with. Finally, in Sect. 4.3.4, patterns that indicate some of the undesirable behavior patterns, characteristic of a poorly designed app, are collected: examples include generating its *hashcode*, and sending the *hashcode* in the OTP request.

4.3.1 Behavioral patterns

The different actions of an app that uses the *SMS Retriever API*, whether well or badly designed, are: Declare the SMS OTP API used; Get the phone number of the device on which it is installed; Prepare to retrieve the OTP SMS that the server sends; Send the phone number; Receive the SMS OTP; and, Send the OTP to the server. The correspondences with code patterns follow in Sects. 4.3.2 and 4.3.3. Up to here, these are actions that a benign app must perform in any case and that we extract from Google's recommendations for this API. The actions that represent a

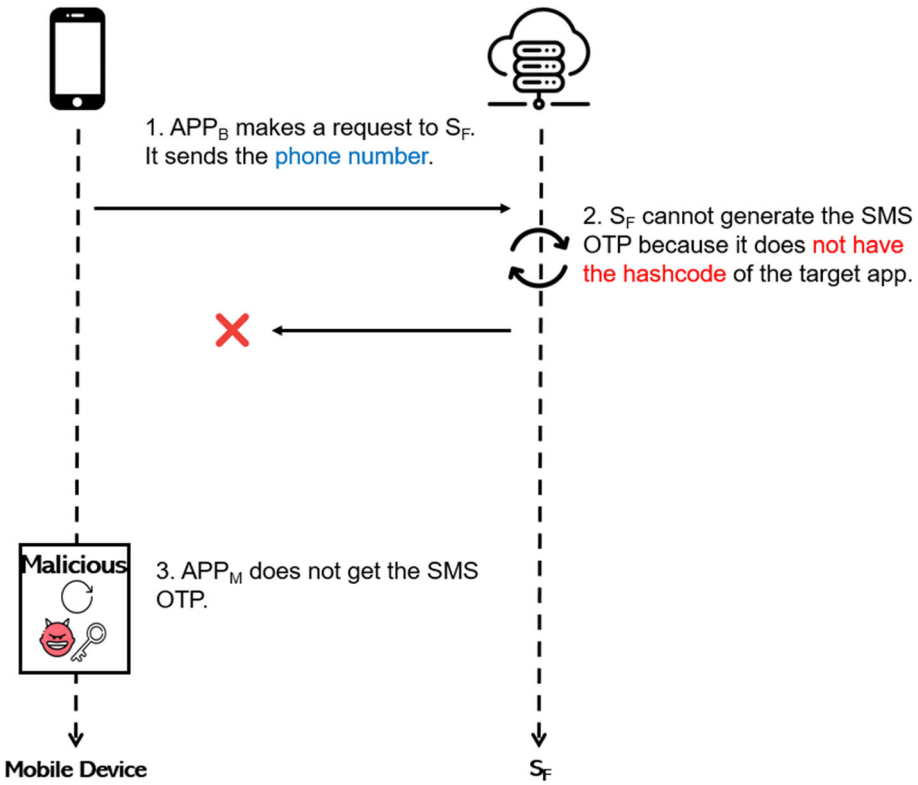


Fig. 5 Interaction diagram for scenario 2

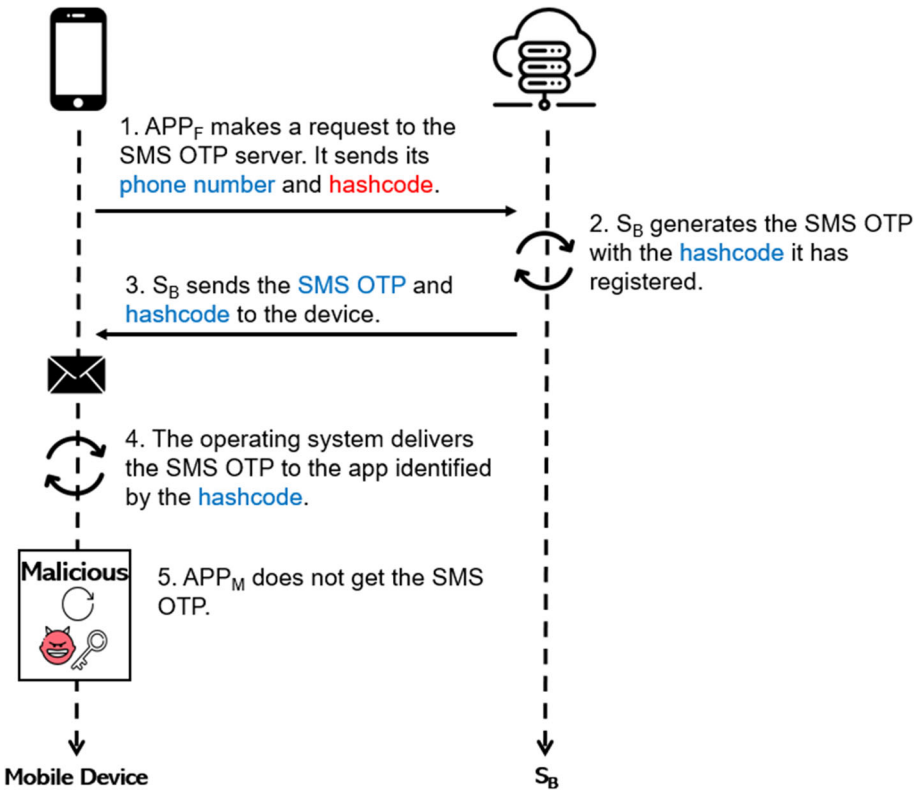


Fig. 6 Interaction diagram for scenario 3

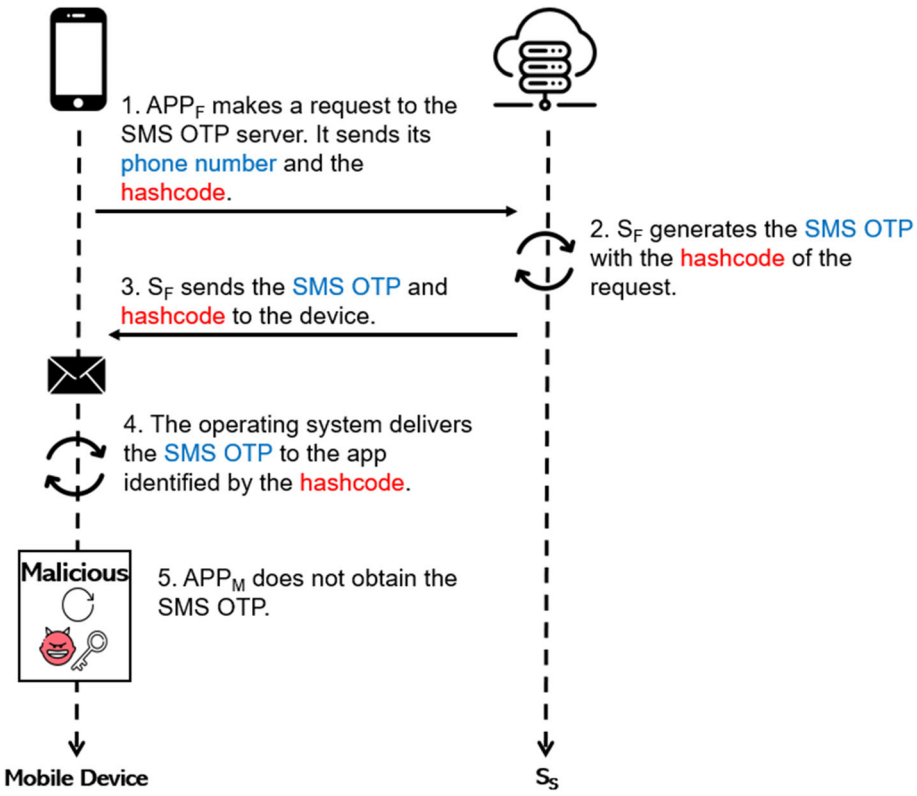


Fig. 7 Interaction diagram for scenario 4

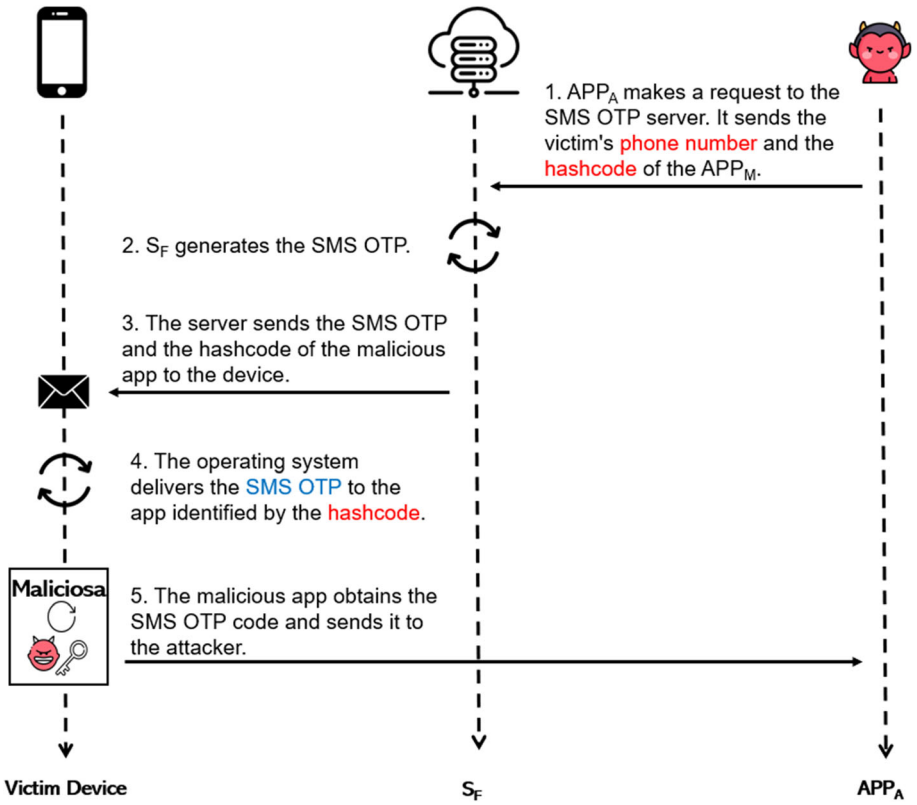


Fig. 8 Interaction diagram for scenario 6

Table 4 Strings of code that identify the use of the SMS verification API

SMS verification API	Code chains
SMS Retriever	SmsRetriever.API | SmsRetrieverClient | SmsRetriever.getClient
SMS Token	createAppspecificToken() | WithPackageInfo()
SMS Token+	createAppspecificSmsToken | WithPackageInfo()
SMS Token	sendSMS()

deviation from the standard behavior indicated by Google for this API are listed below; thus they are characteristic of poorly designed apps, APPF. They are: Generation of hashcode that identifies it within the app itself; and, Shipment of hashcode to the server.

4.3.2 Declaration of the SMS OTP API being used

Table 4 shows code fragments that must be used to implement the different SMS verification APIs in the apps.[4] Any of them is considered an indicator of the use of the corresponding API.

4.3.3 Patterns associated with the use of *SMS Retriever API*

Table 5 shows the code patterns[5] that can be associated with the use of the SMS Retriever API. For each behavior pattern, the corresponding code patterns are collected. The appearance of any of them is an indicator of the associated behavior or action.

4.3.4 Patterns associated with generating and sending a hash

Generating and sending a hash in the app a deviation from the normal use intended for this API. Any possible technique for generating and sending hashes can be used; they are not tied to the use of an API for sending SMS OTP codes.

Table 6 lists the steps of the pattern in the generation of a *hash*. Each of these steps requires the use of a package or class, which must be declared initially. The code pattern will be the joint appearance of two subpatterns: the one that declares the use of the class or package and the one corresponding to the method invocation.

Table 7 shows the last pattern, P12, which corresponds to the sending of a *hash*. The sending is done through an HTTPS request [7] and a JSON object in which the message includes the hash generated, as occurs in the OTP code sending pattern, P6, and in the telephone number sending pattern, P4.

5 Evaluation and results

In this section, we present the results obtained in the application of our methodology to the apps of the banking sector.

5.1 Selection of app category and SMS OTP API

Our category for searches in the Google Market repository in step 2 is "Finance". In the case of this study, the category is "finance" from *Google Play*[6]. Next, we searched the most used apps in Spain for Android devices in this category. They are six, shown in Table 8.

In step 5, after carrying out the static analysis of the apps mentioned above[7], we detected that the most used APIs for managing SMS OTP codes are *SMS Retriever* and *One-Tap SMS verification* API. To do this, we checked the appearance of the characteristic strings of the use of the SMS management APIs described in Table 4. The result was that four of them, those corresponding to the banks CaixaBank, Bankia, Banco Santander and Caja Rural, use the SMS Retriever API; while the remaining two, BBVA and ING, use the API One-Tap SMS verification. Column 2 in Table 9 shows the code chain used by each app. We detected that two of the apps analyzed use the *One-Tap SMS verification* API. Column 2 of Table 10 shows the code chain used by these apps.

5.2 Evaluation of the use of the SMS Retriever API

Knowing that the API *SMS Retriever* is the most used by the apps selected in step 3, the static analysis in step 8 of our methodology is performed on the apps described in Table 10. During this analysis, the patterns described in Table 7 are applied to see which of them are found in the

[4] We use the character "|" to indicate the alternative between several strings of code. The appearance of any of the strings will indicate the use of the SMS verification API. The character "|"does not appear in the SMS verification API source code.

[5] We use the character "|" to indicate the alternative between several strings of code. The appearance of any of the strings will indicate the presence of that pattern. The character "|" does not appear in the source code of the analyzed app.

[6] https://play.google.com/store/apps/category/FINANCE?hl=es &gl=ES

[7] The apps are available for download and analysis in the URL: https://doi.org/10.5281/zenodo.7655968

Table 5 Patterns associated with the use of SMS Retriever API

Behaviour patterns: SMS Retriever API actions		Code patterns
#Pattern	Description	
P1	Use of the SMS Retriever API	SmsRetriever.API
P2	Get user's phone number	setPhoneNumberIdentifierSupported(true) | getParcelableExtra(Credential.EXTRA_KEY)
P3	Start of OTP SMS recovery	SmsRetriever.getClient
P4	Sending phone number to the server	With an HTTPS request by POST[2] to a JSON object: JSONObject postData = new JSONObject()| postData.put("tel", nTel) | postData.put("id")
P5	Reception of the OTP SMS code	SmsRetriever.SMS_RETRIEVED_ACTION | SmsRetriever.EXTRA_STATUS | SmsRetriever.EXTRA_SMS_MESSAGE | com.google.android.gms.auth.api.phone.internal.ISmsRetrieverApiService
P6	Sending the OTP to the server	With an HTTPS request by POST[2] to a JSON object: JSONObject postData = new JSONObject()| postData.put("tel", nTel) | postData.put("hash", "nXzGtk7rNLW") | jSONObject.put("tokenId

Table 6 Patterns associated with the generation of a hashcode from an app

Behaviour patterns: Generation of hash		Code patterns	
#Pattern	Description step	Package/Class	Method
P7	Step 1: Get the package name	*android.content.ContextWrapper*	getPackageName()
P8	Step 2: Obtaining the signing certificate	*android.content.pm.Signature*	toCharsString()
P9	Step 3: Calculation of the SHA256 of the concatenation of the package name and the signing certificate	*MessageDigest digest*	MessageDigest. getInstance("SHA256") | digest.digest((message + appSignature).getBytes
P10	Step 4: Step 4: Coding the hash SHA256 en Base64	*android.util.Base64*	encodeToString()
P11	Step 5: Obtaining the hash: the first 11 characters of the summary obtained by applying the SHA256 function	*java.util.Arrays*	copyOfRange()

Table 7 Patterns associated with sending a hashcode from an app

Behaviour patterns: Send of hash		Code patterns
#Pattern	Description	
P12	Shipment of hash to server	With an HTTPS request by POST[2] to a JSON object: JSONObject postData = new JSONObject () postData.put("hash", "nXzGtk7rNLW")

source code of the apps. This allowed us to know how the app sends the parameters to the server. If any of them is the hash, it will allow a bad implementation by the app developers to be detected. Column 2 of Table 11 shows which patterns, indicative of the use of the API *SMS Retriever* (patterns in Table 7), are present in the source code of the analyzed apps.

It can be seen that the app "Nueva Santander_8.1.1_apkcombo.com.apk" is the only one that presents all the patterns described in Table 6. The P2 and P6 patterns (obtaining the user's phone number, sending the OTP code to the server) do not appear in the rest of the apps analyzed. Another objective of this static analysis is to detect the patterns that show the generation of the hash in

Table 8 Most popular online banking apps in Spain present in Google Play

Bank	Version	Downloads	Evaluations	Stars
CaixaBank	5.41.0	+10 M	+500	4,5
BBVA	Varies according to device	+10 M	+100K	4,4
Bankia	Varies according to device	+5 M	+200K	4,2
Banco Santander	8.6.13	+5 M$	+90K	3,6
ING	3.6.1	+1 M	+70K	4
Caja Rural	5.0.3	+1 M	10K	2,9

Table 9 Banking apps that use the SMS Retriever API

APP	Chain of the SMS Retriever API
caixabanknow.apk	SmsRetriever.API
bankia.apk	SmsRetriever.API
Nueva Santander_8.1.1_apkcombo.com.apk	SmsRetriever.API
ruralvia.apk	SmsRetriever.API

Table 10 Banking apps that use the API One-Tap SMS verification

APP	Chain of the API One-Tap SMS verification
bbva.apk	sendSMS(String phoneNumber, String body) sendSMSFromUri("smsto:" + phoneNumber, body)
ING.apk	sendSMS(String phoneNumber, String body) tas_client_info.setClientKey(key_cliente)

Table 11 Patterns of use of the SMS Retriever API in selected apps

APP	Patterns of SMS Retriever API
caixabanknow.apk	P1, P3, P4, P5
bankia.apk	P1, P3, P4, P6
Nueva Santander_8.1.1_apkcombo.com.apk	P1, P2, P3, P4, P5, P6
ruralvia.apk	P1, P3, P4, P5

Table 12 Patterns of use of the SMS Retriever API in selected apps

APP	Patterns of SMS Retriever API
caixabanknow.apk	
bankia.apk	
Nueva Santander_8.1.1_apkcombo.com.apk	P9, P10, P11
ruralvia.apk	

the apps, which demonstrates that the developer of the app has not carried out a correct implementation of the SMS Retriever API. Column 2 of Table 12 shows the patterns collected in Table 7 that are present in the source code of the selected apps. Blank cells mean that no evidence of the appearance of any of them has been found.

It can be seen that the app "Nueva Santander_8.1.1_apkcombo.com.apk" is the only one where most of the patterns described in Table 8 appear within the same scope. Figure 9 shows a screen-shot of the code snippet where they appear.

6 Discussion

Our proposal has focused on looking for bad implementations of the automatic SMS verification APIs in the apps. In this section, the advantages and limitations of our approach are discussed.

Our proposal is characterized by using a static analysis of the applications, where certain patterns are sought. In turn, the patterns are defined in two phases or levels of abstraction. The advantage of this approach is that future changes at the development level would only affect this last

```
public static String c(String str) {
    try {
        MessageDigest instance = MessageDigest.getInstance("SHA-256");
        byte[] bytes = str.getBytes(HTTP.UTF_8);
        instance.update(bytes, 0, bytes.length);
        return Base64.encodeToString(instance.digest(), 2);
    } catch (NoSuchAlgorithmException e2) {
        e = e2;
        if (!f5293d) {
            return null;
        }
        Log.w(f5294e, "Problem generating hash", e);
        return null;
    } catch (UnsupportedEncodingException e3) {
        e = e3;
        if (!f5293d) {
            return null;
        }
        Log.w(f5294e, "Problem generating hash", e);
        return null;
    }
}
```

Fig. 9 Hash generation in the banking app of the Banco Santander

phase: it would be enough to conveniently extend the set of code patterns considered in Tables 5, 6 and 7.

As explained in Sect. 4, during the presentation, this proposal is applicable in cases where there is no access to the OTP code servers. This is the situation of external auditors looking for a methodology to analyze the security of apps exchanging OTP codes. The proposal is also useful for researchers interested in developing security risk analysis methodologies for mobile app development. However, it is also useful for developers who use these OTP exchange APIs. They can use it to know which errors they should avoid in their developments. As the research the use of OTP exchange APIs mentioned in Sect. 3 indicates, these errors are much more frequent than one might initially assume. This may be because developers are not fully aware that some practices actually involve bugs that lead to security vulnerabilities that impact the users of their developments. Works such as ours offer a guide to avoid them.

The question arises as to why a dynamic analysis has not been proposed. As mentioned in Sect. 2, a dynamic analysis requires the possibility of viewing the requests that an app sends to an OTP code server, as well as having a customer account in the bank of the app under analysis. That is, to be able to observe an interaction between an app and the associated banking server. This interaction must be done through a secure channel[8], which makes it very difficult to access the information sent from the app to the SMS server, as it travels encrypted. However, apps are more readily available to any researcher or analyst, as they are available in the markets.

The limitations of this approach stem from the difficulties in locating the code patterns presented in Sect. 4. This may be due to several reasons. First, it is possible that certain code fragments may appear obfuscated during static analysis. In some cases, as can be seen in the study presented in Sect. 5, this implies that there are signs of the appearance of the patterns sought, although no definitive conclusions can be drawn. It is also possible that the patterns appear in different files when the code is deployed. This is particularly relevant when looking for hash generation, an operation performed in several steps. In our study, we decided to use the scope as reference, so we only consider a sign to be sufficient when the patterns appear in the same scope. Working with evidence is, however, common in external audits.

There are several possibilities to reuse the proposed methodology. The first is to adapt the scenarios for other APIs, or other methods of exchanging OTPs. The second is to adapt them for other *malware* variants. Here, we have considered an app that captures the OTP because it is the recipient of the SMS message. This *malware* variant has been chosen because it is one of the most common and exploited vulnerabilities when using the *SMS Retriever* API. However, there exists *malware* that exploits other vulnerabilities, such as the aforementioned vulnerability that makes SMS messages accessible to any application with access to this tray. Any adaptation would begin with the design of the scenarios.

7 Conclusions and future work

We have proposed a method to find out if an SMS OTP server that uses the SMS Retriever API is vulnerable. The originality of the proposal is that the method is based on a static analysis of the apps, which provides a generality that

[8] Request SMS verification in an Android app. https://developers.google.com/identity/sms-retriever/request?hl=es-419.

methods based on analyses of the servers lack. Gaining access to the apps is much easier for anyone interested in this research than attempting to access the servers.

We have seen that the most commonly used method for the automatic verification of SMS OTP messages in the analyzed apps belonging to Spain's banking sector is the *SMS Retriever* API. Moreover, we have found indications suggesting that some apps do not implement this API correctly. This points to the question of whether using this API is the most appropriate decision for this sector. The results obtained lead us to question whether the balance between the use of using banking apps and their security could be better. Security has historically been a hallmark of the banking sector, and it should be the same with the use of online banking apps.

We propose as future work the extension to other OTP code verification APIs, as well as the application of the proposed methodology to authentication methods other than SMS OTP.

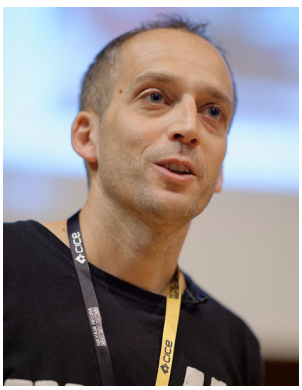
Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability The datasets generated during and/or analyzed during the current study are available in the ZENODO repository: Android Apps banking sector: https://doi.org/10.5281/zenodo.7655968 Screenshots Android API OTP SMS: https://doi.org/10.5281/zenodo.7656663

References

1. Authority, E. B. (2015). Directive (EU) 2015/2366 of the European Parliament and of the Council of 25 November 2015 on payment services in the internal market, amending Directives 2002/65/EC, 2009/110/EC and 2013/36/EU and Regulation (EU) No 1093/2010, and repealing Directive 2007/64/EC (Text with EEA relevance) (2015). https://eur-lex.europa.eu/eli/dir/2015/2366/oj
2. Lei, Z., Nan, Y., Fratantonio, Y., Bianchi, A., & Talos, C. (2021). On the insecurity of SMS one-time password messages against local attackers in modern mobile devices. *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2021.24212
3. Zhou, Y., Hu, L., & Chu, J. (2017). An enhanced SMS-based OTP scheme. In *Proceedings of the 2017 2nd international conference on automation, mechanical control and computational engineering (AMCCE 2017)*, pp. 1091– 1094. Atlantis Press, (2017/03). https://doi.org/10.2991/amcce-17.2017.196
4. Aloul, F., Zahidi, S., & El-Hajj, W. (2009). Two factor authentication using mobile phones. In *2009 IEEE/ACS international conference on computer systems and applications*, pp. 641– 644. https://doi.org/10.1109/AICCSA.2009.5069395
5. Eldefrawy, M. H., Alghathbar, K., & Khan, M. K. (2011). Otp-based two-factor authentication using mobile phones. In *2011 eighth international conference on information technology: new generations*, pp. 327– 331. https://doi.org/10.1109/ITNG.2011.64
6. Developers, G. Automatic SMS Verification with the SMS Retriever API. https://developers.google.com/identity/sms-retriever/overview
7. Developers, G. One-tap SMS verification with the SMS User Consent API. SMS Verification APIs. https://developers.google.com/identity/sms-retriever/user-consent/overview
8. Mayrhofer, R., Stoep, J. V., Brubaker, C., & Kralevich, N. (2021). The android platform security model. *ACM Transactions on Privacy Security*. https://doi.org/10.1145/3448609
9. Bojjagani, S., & Sastry, V. N. (2017). Vaptai: A threat model for vulnerability assessment and penetration testing of android and IOS mobile banking apps. In *2017 IEEE 3rd international conference on collaboration and internet computing (CIC)*, pp. 77–86 (2017). https://doi.org/10.1109/CIC.2017.00022
10. Kazi, M. A., Woodhead, S., & Gan, D. (2023). An investigation to detect banking malware network communication traffic using machine learning techniques. *Journal of Cybersecurity and Privacy, 3*(1), 1–23. https://doi.org/10.3390/jcp3010001
11. Zimba, A., Chen, H., & Wang, Z. (2019). Bayesian network based weighted apt attack paths modeling in cloud computing. *Future Generation Computer Systems, 96*, 525–537. https://doi.org/10.1016/j.future.2019.02.045
12. Ma, S., Feng, R., Li, J., Liu, Y., Nepal, S., Diethelm, Bertino, E., Deng, R.H., Ma, Z., & Jha, S. (2019). An empirical study of SMS one-time password authentication in android apps. In *Proceedings of the 35th annual computer security applications conference*. ACSAC '19, pp. 339– 354. Association for Computing Machinery. https://doi.org/10.1145/3359789.3359828
13. Aparicio, A., Martínez, M. M., & Cardeñoso, V. (2023). Vulnerabilities of the SMS retriever API for the automatic verification of SMS OTP codes in the banking sector. In *Proceedings of the international conference on ubiquitous computing & ambient intelligence (UCAmI 2022)*, pp. 983– 994. Springer. https://doi.org/10.1007/978-3-031-21333-5_99
14. Developers, A. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission
15. Muthumanickam, K., & Senthil Mahesh, P. (2020). A collaborative policy-based security scheme to enforce resource access controlling mechanism. *Wireless Networks, 26*(4), 2537–2547. https://doi.org/10.1007/s11276-019-01984-x
16. Li, Z., & Feng, G. (2020). Inter-language static analysis for android application security. In *2020 IEEE 3rd international conference on information systems and computer aided education (ICISCAE)*, pp. 647– 650. https://doi.org/10.1109/ICISCAE51034.2020.9236807 . IEEE
17. Dmitrienko, A., Liebchen, C., Rossow, C., & Sadeghi, A.-R. (2014). On the (in) security of mobile two-factor authentication. In Financial cryptography and data security: 18th international conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18, pp. 365– 383. https://doi.org/10.1007/978-3-662-45472-5_24 . Springer
18. Peeters, C., Patton, C., Munyaka, I.N., Olszewski, D., Shrimpton, T., & Traynor, P. (2022). SMS OTP security (SOS) hardening

SMS-based two factor authentication. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pp. 2– 16. https://doi.org/10.1145/3488932.3497756
19. Varghese, A., & Mathews, D. (2014) Securing SMS-based approach for two factor authentication. International Journal of Research in Computer and Communication Technology, 3(3)
20. Zhou, Y., Hu, L., & CHu, J. (2017). An enhanced sms-based otp scheme. In *2017 2nd international conference on automation, mechanical control and computational engineering (AMCCE 2017)*, pp. 1091– 1094. https://doi.org/10.2991/amcce-17.2017.196 . Atlantis Press
21. Kurniawan, D. E., Iqbal, M., Friadi, J., Hidayat, F., & Permatasari, R. D. (2021). Login security using one time password (OTP) application with encryption algorithm performance. *Journal of Physics Conference Series, 1783*, 012041. https://doi.org/10.1088/1742-6596/1783/1/012041
22. Shesashaayee, A., & Sumathy, D. (2014). Otp encryption techniques in mobiles for authentication and transaction security. *International Journal of Innovative Research in Computer and Communication Engineering, 2*(10), 6192–6201.
23. Bojjagani, S., & Sastry, V. (2017). A secure end-to-end SMS-based mobile banking protocol. *International Journal of Communication Systems, 30*(15), 3302. https://doi.org/10.1002/dac.3302
24. Luo, H., Wen, G., & Su, J. (2020). Lightweight three factor scheme for real-time data access in wireless sensor networks. *Wireless Networks, 26*, 955–970. https://doi.org/10.1007/s11276-018-1841-x
25. Chen, J., Guo, L., Shi, Y., Shi, Y., & Ruan, Y. (2021). An edge computing oriented unified cryptographic key management service for financial context. *Wireless Networks*. https://doi.org/10.1007/s11276-021-02831-8
26. Gosavi, S., & Shyam, G. K. (2020). A novel approach of OTP generation using time-based OTP and randomization techniques. In *Data Science and Security: Proceedings of IDSCS 2020* (pp. 159-167). Springer Singapore. https://doi.org/10.1007/978-981-15-5309-7_16
27. Aloul, F.A., Zahidi, S., & El-Hajj, W. (2009). Two factor authentication using mobile phones. In *2009 IEEE/ACS international conference on computer systems and applications*, pp. 641–644. https://doi.org/10.1109/AICCSA.2009.5069395



Amador Aparicio is a member of the Department of Computer Science at the University ofValladolid, Spain. His research areas are Security and Privacy. He is professor in several Masters andCybersecurity subjects.https://www.mypublicinbox.com/AmadorAparicio.

M. Mercedes Martínez-González is an Associate Professor in the Department of Computer Science of the University of Valladolid, Spain. She obtained her Ph.D. in Computing Science from theUniversity of Valladolid andthe INRIA institute (France). Her research areas include Semantic Web, Data Engineering, Law andTechnology, and Privacy Engineering. She is a teacher of data engineering and Semantic Web topics.She has participated, and coordinated, several publications and symposiums about Law andTechnology. She coordinates LexDatum, an annual meeting about Law and Information Technology.http://www.infor.uva.es/~mercedes.

Valentín Cardeñoso-Payo Ph.D. in physics in 1984 and 1988, both from the University of Valladolid, Valladolid, Spain. Since 1998, he has been the ECA-SIMM Group Director with the University of Valladolid. His researchinterests include machine learning techniques applied to human language technologies,cybersecurity and human–computer interaction and biometric person recognition. He has been theadvisor of ten Ph.D. works in speech synthesis and recognition, online signature verification, andstructured parallelism for high-performance computing.