

Vulnerabilities of the *SMS Retriever* API for the automatic verification of SMS OTP codes in the banking sector

Amador Aparicio¹, M. Mercedes Martínez-González¹, Valentín Cardeñoso¹
¹ Departamento de Informática, Universidad de Valladolid, 47071, Valladolid, Spain.
{amador,mercedes,valen}@infor.uva.es

The final, published version of this article is available online. Please check the final publication record for the latest revisions to this article.

[Aparicio, A., Martínez-González, M.M., Cardeñoso, V. (2023). Vulnerabilities of the SMS Retriever API for the Automatic Verification of SMS OTP Codes in the Banking Sector. In: Bravo, J., Ochoa, S., Favela, J. (eds) Proceedings of the International Conference on Ubiquitous Computing & Ambient Intelligence (UCAmI 2022). UCAmI 2022. Lecture Notes in Networks and Systems, vol 594. Springer, Cham.

https://doi.org/10.1007/978-3-031-21333-5_99

Abstract. One of the ways to authenticate users of mobile devices is by sending One Time Password (OTP) codes via SMS messages. In order to facilitate the use of these codes by customers, Google has proposed APIs that allow the automatic verification of the SMS messages without the intervention of the users themselves. One of these APIs is the *SMS Retriever* API for Android devices. This article presents a study of this API. Different scenarios of interaction between mobile apps and SMS OTP servers are posed to determine which implementations of the *SMS Retriever* API are vulnerable. The study presented here focuses on Spain's banking sector. The results show that there are vulnerable implementations which would allow cybercriminals to steal the users' SMS OTP codes. The desirable equilibrium between ease of use and security needs to be improved in order to maintain the high level of security which has traditionally characterized this sector. The proposed methodology, applied here to this particular sector (banking), is nevertheless simple enough to be applied to any other sector. One of its advantages is that it proposes a method for detecting bad implementations of the *SMS Retriever* API on the server side, based analyses of the apps, which would make it easily applicable.

Keywords: app, SMS, OTP, 2FA, Android, malware, security, privacy.

1 Introduction

The use of double factor authentication (2FA) mechanisms gives users greater security and thus greater confidence when accessing a system or authorizing a transaction.

This is an authentication method that has become extremely widespread since apps have become one of the commonest ways for users to interact with services in the internet. Mobile devices are nowadays a basic tool that practically every person uses on a daily basis. When they use applications for mobile devices (apps), the 2FA systems permission the user's identity to be verified by sending an OTP code to his/her device. Only the user who receives the OTP code can verify her/his identity. Double factor authentication relies on SMS to handle the OTP codes. 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. However, a bad implementation of the 2FA in the apps means that this authentication scheme becomes vulnerable [1,6].

In this article, a study is presented that analyzes the handling of the SMS OTP codes by the APIs in the banking sector apps. The APIs and the research method used (the latter being applicable to any sector) are described. This method includes the selection of the most commonly used banking apps in Spain, an analysis of the most commonly used APIs that handle the SMS OTP codes and the study of the vulnerabilities present in the chosen API.

Following the analysis of a bank app installed in an Android device infected by banking *malware*¹, we discovered that the app used the *SMS Retriever* API [2]. Due to the bad implementation of the *SMS Retriever* API in the app, there appeared vulnerabilities, present in [1], that allowed the theft of OTP codes through the SMS service. For that reason, we decided to investigate how bank apps manage the OTP codes received via SMS in Android mobile devices.

The rest of the article is organized as follows: Section 2 shows the APIs that handle the SMS messages used in the banking sector; Section 3 presents a general methodology, applicable to any sector, for detecting bad implementations of the *SMS Retriever* API; Section 4 presents the results obtained after applying the methodology to the most commonly used banking sector apps in Spain; and Section 5 sets out the conclusions.

2 The use of APIs for managing SMS OTP codes in Spain's banking sector apps

The APIs that handle SMS OTP codes appeared so that the user did not have to manually introduce the OTP code in the app. Google offers various APIs to manage SMS OTP codes in Android devices [2,9]: one-touch verification by SMS (*One-tap SMS verification*) and automatic verification by SMS (*SMS Retriever*). Bank apps for Android devices use these APIs. The *One-tap SMS verification* API needs the user to authorize the recuperation by the app of the OTP code [9]. There is a risk here that the users will not understand that they need to agree to authorizing the OTP recuperation code. On the other hand, the *SMS Retriever* API automatically recuperates the SMS

¹ Malicious app whose objective is to remotely control the device in order to steal such banking information as: bank account credentials, authorization codes for electronic transactions, etc.

OTP codes destined for the apps without having to request any interaction with the app on the user's part [2].

Section 2.1 shows a generic process for exchanging SMS OTP codes for Android devices. Section 2.2 sets out in detail the exchange of SMS OTP codes in apps belonging to the finance sector for Android devices; while Section 4 presents the study of the APIs for the automatic verification of the SMS OTP codes used in Spain's banking sector.

2.1 Exchange of SMS OTPs in Android devices

In Android, applications can request permission to access SMS messages. The necessary permissions are `READ_SMS` [4,5] and `RECEIVE_SMS` [4,5]. The former allows an app to read the SMS in-tray at any time. The latter allows the app to read the new messages that enter just before they appear in the in-tray. Google classifies both as *dangerous* [4,5,8], as they allow SMS messages to be read in an arbitrary manner, even when the messages are not relevant for the functioning of the app. As they are *dangerous* permissions, they require user consent at time of execution and this may not be granted.

To facilitate their management and to free the user from having to handle the OTP codes or the SMS permissions, Android introduced the APIs *SMS Retriever* [2] and *SMS User Consent* API [9] for the automatic verification of SMS OTP codes [1,2]. From here on, we focus on the *SMS Retriever* API, which requires no intervention on the part of the user. For the API 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. The server of the application sends the SMS OTP code to the device that made the request together with 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 [1,2].

The paper [1] demonstrates the vulnerabilities present in the *SMS Retriever* API. There are three possibilities. First of all, the SMS OTP is not eliminated from the in-tray, which would allow other applications to access it, for instance, malware installed in the device itself. The second possibility is that the *hash* which identifies the receiving app's SMS OTP is not generated or not registered in the SMS OTP server, while it is the app which sends it. This may be due to the fact that the app has it in its source code, or that the app can generate it dynamically. Finally, an attacker could generate an SMS OTP petition to a remotely controlled malicious app installed in the victim's device in order to steal the SMS OTP code.

The vulnerabilities are taken advantage of by the *malware* to steal the SMS OTP codes, which are then sent to the attacker in order to complete the authorization for the fraudulent bank transaction. To do so, the malware needs to have the permission `android.permission.INTERNET`. This permission has a normal protection level [4,5], which means that it is granted at the moment the app (the *malware* in this case) is installed in the device and cannot be managed by the user

while the app is being executed [4,5,8]. The participants in this interaction are summarized in the diagram in Figure 1.

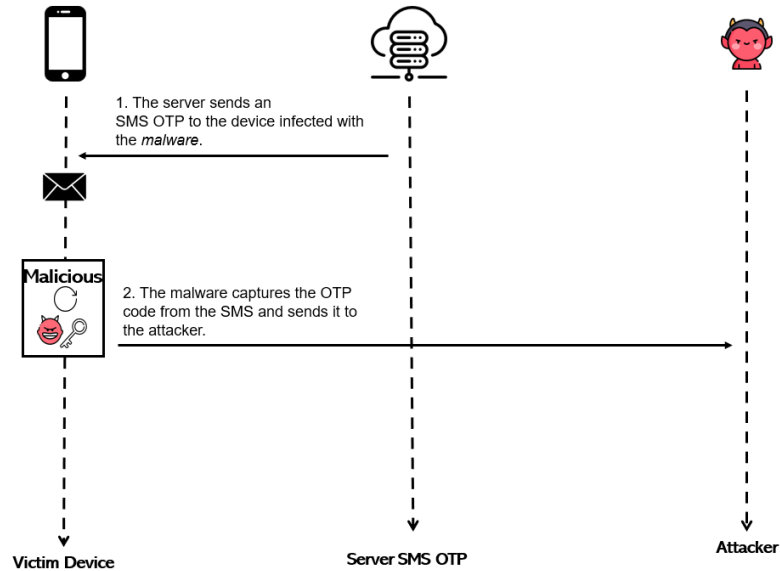


Fig. 1. The *malware* in the device captures the SMS OTP and sends it to the attacker.

2.2 The apps of Spain's banking sector

The apps used in digital banking use SMS OTP codes to identify users through mobile devices and thus authorize bank transactions. We carried out a study which discovered that the apps of Spanish banks use the APIs for the automatic verification of SMS OTPs proposed by Google and presented in Section 2.1.

When clients of digital banking use these services, the OTP codes that authorize their transactions are sent via SMS to a telephone number that the clients have already provided when registering for the service. This is the case whether the clients use a banking app to make transactions or the bank's website. In our study, we focus on situations in which banking apps are used.

3 Study of the vulnerabilities in the APIs for the automatic verification of SMS OTPs

The work consists of the study of a sufficiently representative set of apps from the banking sector to know how the SMS OTPs are handled.

3.1 Methodology

The methodology presented here starts from the supposition that the sector whose apps are to be studied is already known. As indicated above, the methodology is applicable to any sector. The steps are as follows:

1. The most used apps from the chosen category are selected. In the case of this study, the category is “finance” from *Google Play*². The selection criteria to be used are: largest number of downloads, largest number of evaluations, and largest number of stars.
2. A static analysis of the selected apps is carried out to know which automatic verification method of SMS messages is used.
3. The most commonly used automatic verification method of SMS messages in the analyzed apps is selected. The study focuses on the most used API.
4. A search is carried out to identify the bad implementations faults when using the SMS automatic verification API selected in the point above. The bad implementations correspond to the scenarios presented in [1]. 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 receiving the *hash* of the app, which indicates that the server did not generate it or did not have it registered. The correct implementation of the *SMS Retriever* API requires the server to know or to calculate the *hash* with respect to the package name of the app and
5. Given that it is not usual to have access to the SMS OTP servers used by the selected apps, we focus the study 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.
6. We then check whether, in the set of apps, the implementation faults of the API for the automatic verification of SMS messages actually occur.
7. The results obtained are analyzed.

3.2 Static analysis

Static analysis allows us to search the source code of the apps for code chains without the apps being executed in the device [7]. Having found the code chains described in Table 1 in the source code of the app [1], we can know which API for managing SMS OTP codes is used by the app.

Table 1. Code chains present in the APIs that manage SMS codes.

API de verificación SMS	Code chains
<i>SMS Retriever</i>	SmsRetrieverClient SmsRetriever.getClient SmsRetriever.API

² <https://play.google.com/store/apps/category/FINANCE?hl=es&gl=ES>

SMS Token	createAppspecificToken() WithPackageInfo()
SMS Token+	createAppspecificSmsTokenWithPackageInfo()
One-tap SMS verification	sendSMS()

The first step is to have the apps³. Then, the app to be statically analyzed is selected and is decompiled using the tool `jadx-gui` [10]. Finally, using the same tool, a search is carried out in the source code of the app for the code chains of Table 1. Should the search be successful, we then know the API used to manage the SMS OTP codes.

Given that the banking apps analyzed were downloaded from *Google Play* before being stored, we shall focus on the APIs *SMS Retriever* and *One-tap SMS verification*, which are the APIs for managing SMS OTP codes proposed by Google [2,9].

4 Results obtained

After applying step 1 of the methodology proposed in the Spanish banking sector, the most commonly used apps in Spain for Android devices in the finance category are those shown in Table 2.

Table 2. Most popular online banking apps in Spain present in Google Play.

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

In step 2, after carrying out the static analysis of the apps mentioned above, we detected that the most used APIs for managing SMS OTP codes are *SMS Retriever* (see Figures 2, 3, 4 and 5) and *SMS OTP IFA* (see Figures 5 and 6). Column 2 in Table 3 shows the code chain used by the *SMS Retriever* API.

Table 3. Banking apps that use the *SMS Retriever* API.

APP	Chain of the <i>SMS Retriever</i> API
caixabankknow.apk	SmsRetriever.API
bankia.apk	SmsRetriever.API SmsRetrieverClient
Nueva Santander_8.1.1_apkcombo.com.apk	SmsRetriever.API

³ The apps are available for download and analysis in the URL <https://cutt.ly/UI91Eso> [password: UCAMI2022]

	SmsRetrieverClient
ruralvia.apk	SmsRetriever.API SmsRetrieverClient

We detected that two of the apps analyzed use the *One-Tap SMS verification* API (see Figures 5 and 6). Column 2 of Table 4 shows the code chain used by the *One-Tap SMS verification* API.

Table 4. Banking apps that use the API *One-Tap SMS verification*.

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

Step 3 determines that the method for managing SMS messages most commonly used in the analyzed apps is *SMS Retriever* (see Figures 3, 4, 5 and 6, and Table 3). In addition, this is the verification method found in the banking app present in the Android device infected by banking *malware*. So we decided to study this API for automatically verifying SMS messages.

In steps 4, 5 and 6, knowing how the *hash* that identifies the apps (see Figure 2) is generated, we searched the source code of the apps for the methods and packages shown in [1] that are used to generate the *hash* of the apps (see Figure 3 and the Appendices).

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

Fig. 2. Generating the hash of an app.

```
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. 3. Packages and methods used to calculate the *hash* of the apps.

5 Conclusions

We have carried out a study of the diverse ways of using the automatic verification of OTP codes in the development of apps for Spain's banking sector. The most important conclusions obtained are as follows:

- 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.
- We have proposed a method to find out if an SMS OTP server does not correctly implement the *SMS Retriever* API. The originality of the proposal is that the method is based on an analysis of the apps, which brings a generality that

methods based on analyses of the servers lacks. Gaining access to the apps is much easier for anyone interested in this research than attempting to access the servers. This method analyzes the way in which the apps handle the *hash*.

- We have seen that the banking sector does not implement correctly the API to automatically verify SMS messages in the apps and in the servers of SMS OTP codes.
- The balance between the ease of using apps and their security is not the adequate one in the banking sector. The banks are encouraging their clients to use electronic banking, but technological barriers mean that the apps have to be easy to use. This means that clients with scarce digital skills will be able to use these applications, thus reaching an older sector of the population. However, this unquestionable advantage comes with security risks; which, in our opinion, are excessive, given the consequences they may have for people when they see funds disappearing from their bank accounts. We therefore believe that security in the technological solutions for this sector is particularly important; a quality that has historically been a hallmark of the banking sector. The balance between ease of use and security for banking apps should be better. It is possible to use safer solutions, where the bank servers of SMS OTP codes do not accept that the *hash* codes that identify their apps should be sent by the apps, but that they should be stored as Google recommends in [2]. It would thus be unnecessary for banking apps to send the *hash* that identifies them to the server of the SMS OTP codes. Consequently, neither the *hash* that identifies the app nor the dynamic generation of *hash* patterns should appear in the source code of the apps.
- Finally, a methodology has been proposed that has the qualities of simplicity and generality. The former facilitates its application and getting results from diverse profiles; while the latter means it is applicable to any sector.

As future work, we are working on the development of a *framework* to generate test scenarios for the *SMS Retriever* API. Our objective is to discover vulnerabilities caused by poor implementation by developers in the use of the SMS Retriever API in apps and in the servers used by the apps to request and receive OTP codes. ~~Our objective is to know more and to understand better what the vulnerabilities are due to the bad implementation of this API in the apps and in the SMS OTP server.~~

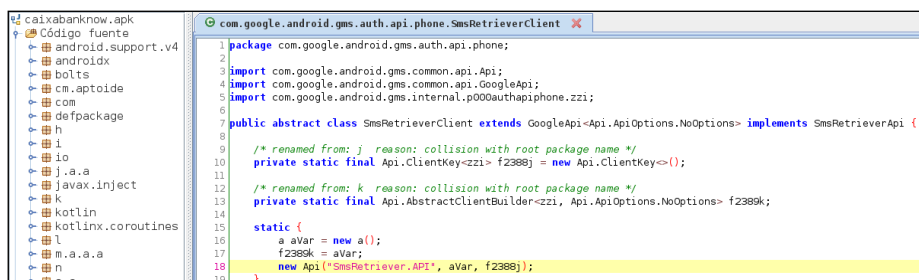
References

1. Lei, Z., Nan, Y., Fratantonio, Y., Bianchi, A. (2021). *On the Insecurity of SMS One-Time Password Messages against Local Attackers in Modern Mobile Devices – NDSS Symposium*. NDSS Symposium. <https://www.ndss-symposium.org/ndss-paper/on-the-insecurity-of-sms-one-time-password-messages-against-local-attackers-in-modern-mobile-devices/>
2. *Automatic SMS Verification with the SMS Retriever API*. (s. f.). Google Developers. Recuperado 16 de septiembre de 2021, de <https://developers.google.com/identity/sms-retriever/overview>.

3. Subramaniam, M. (2019, 16 mayo). *No more SMS permission required for SMS verification in Android O+(8+)*. Medium. <https://s-muthukhumar.medium.com/no-more-sms-permission-required-for-sms-verification-in-android-o-3e9389a1e43e>
4. *Manifest.permission*. (s. f.). Android Developers. Recuperado 3 de octubre de 2021, de <https://developer.android.com/reference/android/Manifest.permission>
5. *Manifest.protectionLevel*. (s. f.). Android Developers. Recuperado 3 de octubre de 2021, de <https://developer.android.com/guide/topics/manifest/permission-element>
6. M4: Insecure Authentication | OWASP. (2016). OWASP Mobile Top 10. Recuperado 21 de diciembre de 2021, de <https://owasp.org/www-project-mobile-top-10/2016-risks/m4-insecure-authentication>
7. Z. Li and G. Feng, "Inter-Language Static Analysis for Android Application Security," 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE), 2020, pp. 647-650, doi: 10.1109/ICISCAE51034.2020.9236807.
8. Mayrhofer, R., Stoep, J.V., Brubaker, C., & Kravovich, N. (2021). The Android Platform Security Model. *ACM Transactions on Privacy and Security (TOPS)*, 24, 1 - 35.
9. *One-tap SMS verification with the SMS User Consent API | SMS Verification APIs |*. (s. f.). Google Developers. Recuperado 17 de enero de 2022, de <https://developers.google.com/identity/sms-retriever/user-consent/overview>
10. GitHub - skylot/jadx: Dex to Java decompiler. (s. f.). GitHub. Recuperado 24 de enero de 2022, de <https://github.com/skylot/jadx>

Appendix A. Evidence of the use of the APIs for OTP code verification in the banking sector

This appendix provides evidence in the form of screen captures extracted from the static analysis carried out concerning the banking applications mentioned in the article. Figures 4, 5, 6 and 7 show evidence of the use of the *SMS Retriever* API in the bank apps “caixabanknow”, “bankia”, “Nueva Santander” and “ruralvia”. In the first, “caixabanknow”, the use of the chain `SmsRetriever.API` in line 18 is evidence of the use of this API. Analogously, the appearance of the chain `SmsRetrieverClient` in the remaining apps is evidence that they all use this API. We have also detected the use of the One-Tap SMS Verification API. The presence of the chain `sendsms` is evidence that both the bank app “BBVA Más Azul”, in Figure 8, and the app “ING” (Figure 9) use this API to manage the OTP codes. The appearance of these chains in all of them has been highlighted in yellow.



```

1 package com.google.android.gms.auth.api.phone;
2
3 import com.google.android.gms.common.api.Api;
4 import com.google.android.gms.common.api.GoogleApi;
5 import com.google.android.gms.internal.p000authapiphone.zzi;
6
7 public abstract class SmsRetrieverClient extends GoogleApi<Api.ApiOptions.NoOptions> implements SmsRetrieverApi {
8
9     /* renamed from: j reason: collision with root package name */
10    private static final Api.ClientKey<?> f2388j = new Api.ClientKey<>();
11
12    /* renamed from: k reason: collision with root package name */
13    private static final Api.AbstractClientBuilder<?> f2388k = new Api.AbstractClientBuilder<?>();
14
15    static {
16        a aVar = new a();
17        f2388k = aVar;
18        new Api("SmsRetriever.API", aVar, f2388j);
19    }

```

Fig. 4. Chain `SmsRetriever.API` present in the bank app “caixabanknow”.

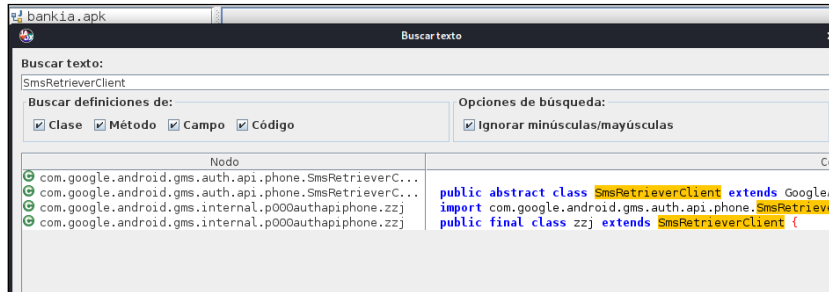


Fig. 5. Chain SmsRetrieverClient present in the bank app “bankia”.

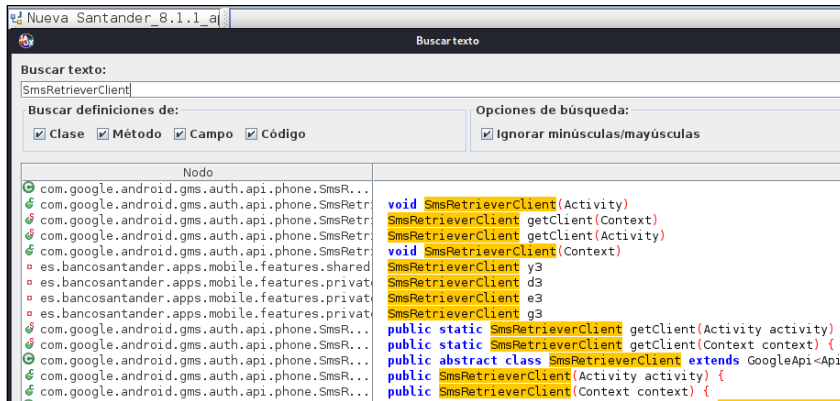


Fig. 6. Chain SmsRetrieverClient present in the bank app “Nueva Santander”.

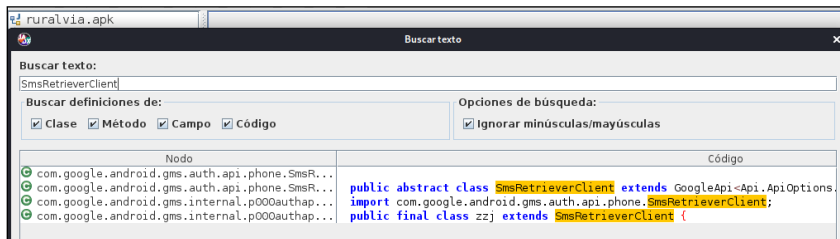


Fig. 7. Chain SmsRetrieverClient present in the bank “app ruralvia”.

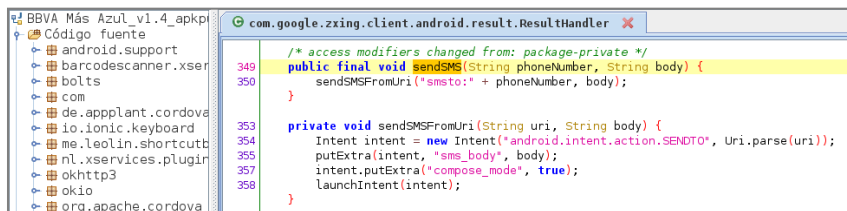


Fig. 8. Chain sendSMS present in the bank app “BBVA Más azul”.

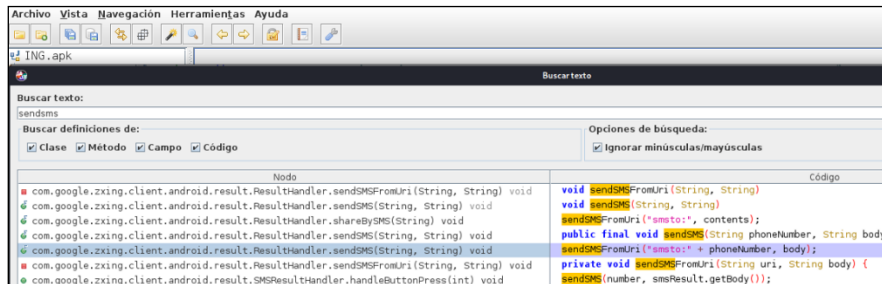


Fig. 9. Chain sendSMS present in the bank app ING.

Appendix B. Evidence of hash generation in the apps

This appendix provides evidence of the use of methods and functions to generate the hash in the studied apps belonging to the banking sector. They are all invocations to methods that allow the package name of the app to be signed to be obtained, or they allow those that permission the signing to be obtained. Figure 10 shows the use of the method `getPackageName` to obtain the name of the app's package. Figure 11 shows the use of the library `android.content.pm.Signature` used to sign the app's package. On the other hand, Figure 12 detects the library `java.security.MessageDigest` used to create summaries of messages in MD5, SHA-1 or SHA-256 format. Figure 13 shows the use of the method `encodeToString` used to represent a chain of characters in base64.

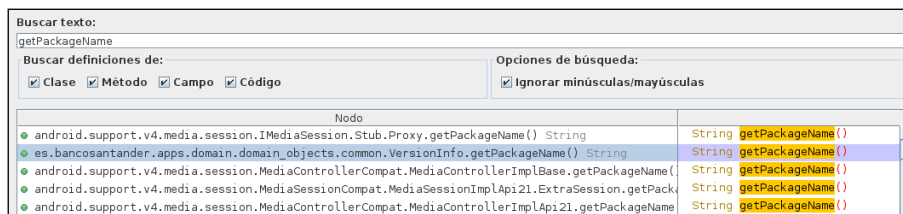


Fig. 10. Chain getPackageName present in the bank app "Nueva Santander".

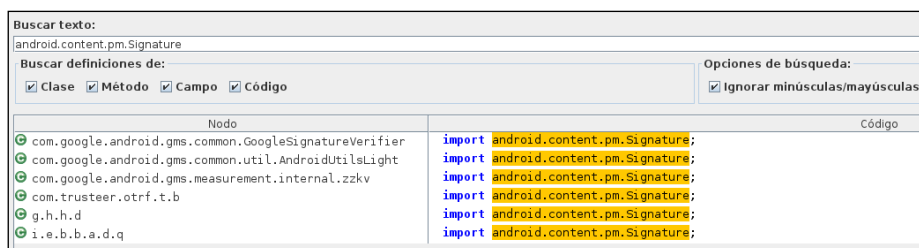


Fig. 11. Chain getPackageName present in the bank app "ING".

Buscar texto:	
MessageDigest.getInstance	
Buscar definiciones de:	
<input checked="" type="checkbox"/> Clase	<input checked="" type="checkbox"/> Método
<input checked="" type="checkbox"/> Campo	<input checked="" type="checkbox"/> Código
Opciones de búsqueda:	
<input checked="" type="checkbox"/> Ignorar minúsculas/mayúsculas	
Nodo	Código
e.i.a.c(String) String	MessageDigest instance = MessageDigest.getInstance("SHA-256");
g.a.a.f.d.c.a.C0383a.d() String	MessageDigest instance = MessageDigest.getInstance("SHA-256");
io.emma.android.controllers.ENMADeviceController.md5(String,...	MessageDigest = MessageDigest.getInstance("MD5");
m.f.d(String) f	return a(MessageDigest.getInstance(str)).digest(this.a);
org.apache.http.impl.auth.DigestScheme.createMessageDigest(...	return MessageDigest.getInstance(str);
org.apache.http.impl.auth.NTLMEngineImpl.f.a(byte[], Certif...	byte[] digest = MessageDigest.getInstance("SHA-256").digest(certificate.getEncoded());
org.apache.http.impl.auth.NTLMEngineImpl.e() MessageDigest	return MessageDigest.getInstance("MD5");
org.spongeycastle.cert.jcajce.JcaX509ExtensionUtils.jcaX509E...	super(new SHA1DigestCalculator(MessageDigest.getInstance("SHA1")));
org.spongeycastle.jcajce.util.DefaultJcaSchemeHelper.b(String) ...	return MessageDigest.getInstance(str);
org.spongeycastle.jcajce.util.NamedJcaSchemeHelper.b(String) Me...	return MessageDigest.getInstance(str, this.a);

Fig. 12. Chain java.security.MessageDigest present in the bank app “Nueva Santander”.

Buscar texto:	
encodeToString	
Buscar definiciones de:	
<input checked="" type="checkbox"/> Clase	<input checked="" type="checkbox"/> Método
<input checked="" type="checkbox"/> Campo	<input checked="" type="checkbox"/> Código
Opciones de búsqueda:	
<input checked="" type="checkbox"/> Ignorar minúsculas/mayúsculas	
Nodo	Código
com.google.android.gms.common.util.Base64Utils.encode(byte{...	return Base64.encodeToString(bArr, 0);
com.google.android.gms.common.util.Base64Utils.encodeUrlSaf...	return Base64.encodeToString(bArr, 10);
com.google.android.gms.common.util.Base64Utils.encodeUrlSaf...	return Base64.encodeToString(bArr, 11);
com.google.android.gms.tagmanager.zzbk.zzv(zzbs>) zzbs	str2 = Base64.encodeToString(bArr, 12);
com.google.android.gms.tagmanager.zzbk.zzv(zzbs>) zzbs	str2 = Base64.encodeToString(bArr, 12 8);
com.google.firebase.iid.p.a(KeyPair) String	return Base64.encodeToString(digest, 0, 8, 11);
com.google.firebase.iid.y0.e(Context, String) File	String encodeToString = Base64.encodeToString(str.getBytes(HTTP
com.google.firebase.iid.y0.e(Context, String) File	StringBuilder sb = new StringBuilder(String.valueOf(encodeToStr
com.google.firebase.iid.y0.e(Context, String) File	sb.append(encodeToString);
com.google.firebase.iid.z0.b() String	return Base64.encodeToString(this.a.getPublic().getEncoded(), 1
com.google.firebase.iid.z0.c() String	return Base64.encodeToString(this.a.getPrivate().getEncoded(), 1

Fig. 13. Chain encodeToString present in the bank app “Nueva Santander”.