[MANDATORY SECURITY] GUIDELINES

GL–016 v.01

SECURE GUIDELINE ANDROID

COVER

Title	Secure Guideline Android
Classification	Mandatory Security Guidelines
Document code	GL-016 v. 01
Approved by	Nexi Group CISO
Approval date	12-06-2023
Date of entry into force	12-06-2023

UPDATES

Version	Date	Code	Updates
1	12-06-2023	GL-016 v. 01	First issue

TABLE OF CONTENTS

1 INTRODUCTION

The purpose of this document is to describe the security requirements that should be implemented in order to develop secure mobile applications for Android.

These requirements originate from the use of standard and internationally recognized methodologies such as OWASP (The Open Web Application Security Project).

The following references were used during the writing of this document:

- “OWASP Development Guide” – OWASP Foundation
- “OWASP Testing Guide” – OWASP Foundation
- “OWASP Mobile Testing Guide” – OWASP Foundation
- “OWASP Cheat Sheets Series” – OWASP Foundation
- “OWASP Secure Coding Practices” – OWASP Foundation

In particular, the security requirements to be implemented are based on the security tests described in the OWASP Mobile Testing Guide. Those will also be the reference for the subsequent security assessment phase of the software produced.

2 REQUIREMENTS DESCRIPTION

Below is a brief description of the requirements categories.

2.1 AUTHENTICATION

In the field of security, authentication is the process designed to verify the digital identity of whoever is interacting with the application. The purpose of authentication controls on application users is to uniquely associate the users with their identity on the system, in order that they can access their data. This also means preventing access to resources by users who do not have access credentials.

2.2 PROTECTION AGAINST REVERSE ENGINEERING

Protection mechanisms against Reverse Engineering are of primary importance since an attacker in possession of the application, or a device with the installed application, could carry out operations such as decompilation to reconstruct logic and information useful for further sophisticated attacks.

2.3 RUNTIME SECURITY CHECKS

Runtime security checks are usually performed to identify whether the application to be protected is running on an insecure environment.

An insecure environment could be used against the application in order to:

- Perform reverse engineering.
- Abusing internal APIs.
- Retrieve sensitive data at runtime.

There are several techniques that can be abused to achieve the aforementioned purposes such as:

- Create a root user who is able to access low-level OS functions (rooted devices).
- Use an Android emulator to install the application, in order to analyze the application from the host system (emulated devices).
- Debug the application using the device API (application debugging).
- Overwrite applications or system libraries through hooking (hooking of functions and methods).

To identify the presence of each of those techniques, one or more checks must be performed at runtime.

Also, to identify whether the application is in a malicious context, the controls should be applied as much as possible in conjunction with other controls presented in this document.

All runtime security checks must be considered as complementary to each other, and it is recommended to apply all of them to achieve a good level of security.

2.4 SENSITIVE DATA MANAGEMENT

Indicates the management of sensitive data used by the application in order to defend the application against Information Disclosure vulnerabilities.

2.5 USER INPUT MANAGEMENT

Each parameter sent or received by the application could lead to serious vulnerabilities with attacks aimed at end users or data managed by the application. It is therefore essential to design an application that implements a correct technique for validating the incoming data, encoding the outgoing data and verifying the correctness of the variables before interacting with the other layers such as DB, File System, and Operating System.

2.6 SECURE COMMUNICATION WITH THE SERVER

Encryption is the data protection process. The purpose of this process is to make any sensitive data unintelligible by a malicious user who has managed to intercept them. It is important to make sure that any sensitive data in transit between client and server is protected by encryption. Furthermore, encryption must be performed with known standard algorithms.

2.7 IPC MECHANISMS

If a mobile application exposes public interfaces via IPC (Inter Process Communication), it is important to apply countermeasures to prevent malicious applications installed on the same device from exploiting these interfaces to make the victim application perform unexpected actions.

2.8 WEBVIEW MANAGEMENT

The WebView components within the applications must be secured in order to limit exposure and entry points for attackers.

2.9 COUNTERMEASURES TO INFORMATION DISCLOSURE

The management and prevention of Information Disclosure issues for sensitive information in a mobile context is central to the security of mobile applications. In fact, mobile devices by their nature supports sensitive operations useful in daily use, such as the possibility to take screenshots of the application or copy-paste Clipboards. These functionalities could be exploited by an attacker for Information Disclosure actions.

3 REQUIREMENT SPECIFICATION

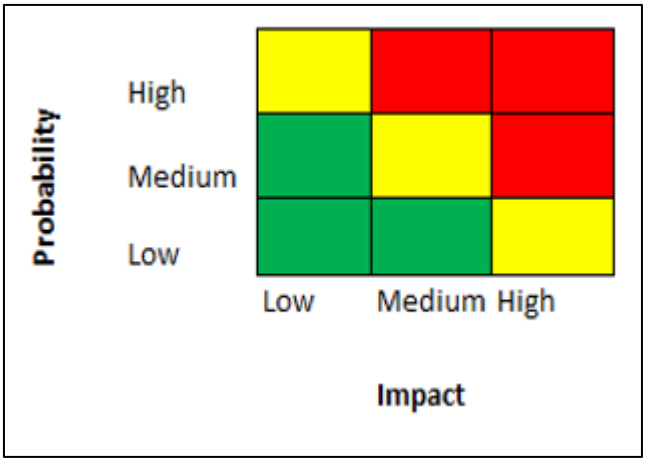
3.1 REQUIREMENT SPECIFICATION

For each requirement stated during the analysis activity, the following aspects were evaluated:

- **Difficulty of exploitation** by an attacker.
- **Technological impact (non-business)** in the event that the vulnerability is exploited by an attacker.
- **Difficulty of resolution** by applying the requirement requested by the customer.
- **Priority of intervention** in the introduction of the required safety requirement.

Based on the previous aspects, the risk in the event of a hypothetical vulnerability present in the system was taken into account for each requirement. This risk is given by the product of the **probability** of the occurrence of an attack due to the vulnerability and the **technological impact** from the exploitation of this activity.

The image below shows the **risk** calculation matrix:



Therefore, the proposed **priority of intervention** takes into account the **risk** value in case of vulnerabilities present in the system, due to the failure to adopt the required security requirements and the **difficulty** in satisfying these requirements, which is measured as the effort by the customer in applying all the countermeasures described within the proposed security requirements.

The table below shows the rules for using the terms used associated with the applicable priority values, to formalize the terminology within the security requirements:

Term	Category	Action
Is necessary Is mandatory	High/Medium	Mandatory
Is strongly suggested Is important to consider	Medium	Not mandatory but it is necessary to assess the risks in case of non-implementation
Is suggested It should be considered	Low	Implementation is not necessary except in situations of particularly stringent safety requirements

The terms: "It is strongly recommended / It is important to consider", indicate that the implementation choice is inherent to business aspects and internal risk analysis that a generic document such as this one cannot consider; therefore, the final implementation choice is left to the customer.

Finally, since the guidelines are a document that identifies and categorizes risk aspects without contextualizing specific applications, the end user can justify the failure to implement a specific control by taking the risk of this choice.

3.2 LIST OF SECURITY REQUIREMENTS

Code		Category	Name	Priority
RU1	3.3.1	Authentication	Secure authentication using Keystore and Fingerprint	High
RU2	3.3.2	Authentication	Secure logout management	Medium
RU3	3.3.3	Authentication	Usage of temporary access tokens	High
RU4	3.3.4	Authentication	Password security requirements	High
RU5	3.3.5	Authentication	PIN Security requirements	High
RU6	3.3.6	Authentication	Verify presence of local authentication	Medium
RU7	3.3.7	Authentication	Authenticate using Active Directory	Low
RU8	3.3.8	Authentication	Protect from User Enumeration	Medium
RU9	3.3.9	Authentication	Protect from bruteforcing	Medium
RU10	3.3.10	Authentication	Authentication with biometric factors	High
RU11	3.3.11	Reverse Engineering Protection	Code obfuscation	High

RU12	3.3.12	Reverse Engineering Protection	Encrypt classes with sensitive code	**High**
RU13	3.3.13	Reverse Engineering Protection	Limit the usage of whitelist –keep*	**Medium**
RU14	3.3.14	Reverse Engineering Protection	Protection against information disclosure via stacktrace	**Medium**
RU15	3.3.15	Reverse Engineering Protection	Static resources encryption	**Medium**
RU16	3.3.16	Reverse Engineering Protection	Static string encryption	**Medium**
RU17	3.3.17	Reverse Engineering Protection	Prevent tampering	**Medium**
RU18	3.3.18	Reverse Engineering Protection	Whitebox cryptography	**Low**
RU19	3.3.19	Security checks at Runtime	Anti-Root controls	**Medium**
RU20	3.3.20	Security checks at Runtime	Anti-Debugging controls	**Medium**
RU21	3.3.21	Security checks at Runtime	Anti-Hooking controls	**Medium**
RU22	3.3.22	Sensitive Data Management	Encryption of personal data	**High**
RU23	3.3.23	Sensitive Data Management	Avoid use of private embedded data	**Medium**
RU24	3.3.24	Sensitive Data Management	Secure management of files	**Medium**
RU25	3.3.25	User input management	Secure implementation of an application PIN Pad	**Medium**
RU26	3.3.26	User input management	Memory Wiping	**Medium**

RU27	3.3.27	User input management	Input validation	**High**
RU28	3.3.28	User input management	Usage of prepared statements	**High**
RU29	3.3.29	Secure communication with the Server	Communication over an encrypted channel	**High**
RU30	3.3.30	Secure communication with the Server	Implement SSL Certificate Pinning	**High**
RU31	3.3.31	IPC mechanisms	IPC interfaces secure management	**Low**
RU32	3.3.32	WebView management	Webview secure settings	**High**
RU33	3.3.33	Countermeasures to Information Disclosure	Protect against log disclosure	**High**
RU34	3.3.34	Countermeasures to Information Disclosure	Protect against screenshot leakage	**Low**
RU35	3.3.35	Countermeasures to Information Disclosure	Protect against credential theft	**Low**
RU36	3.3.36	Countermeasures to Information Disclosure	Protect against clipboard data leakage	**Low**

3.3 REQUIREMENTS DESCRIPTION

Below is a detailed description of the required security requirements listed by application categories.

3.3.1 SECURE AUTHENTICATION USING KEYSTORE AND FINGERPRINT

Requirement ID	AUT-001
Priority	**High**
Description	Mobile applications usually store authentication data on the device to authenticate users. To avoid storing credentials (username and password) on the device, it is advisable to use a random authentication token received during the login phase, with a limited duration. This can be used as an authentication parameter when communicating with remote APIs. The token must be stored on the device using cryptographic algorithms to avoid confidentiality issues. Below are guidelines for client-side implementation of a secure authentication mechanism.
Android	**Android KeyStore** Android provides KeyStore APIs for secure storage of encryption keys, which can be used to encrypt authentication data. Refer to paragraph 3.3.22, which deals with these topics such as the generation of symmetric keys and the storage of data in encrypted form. **Fingerprint** If it is necessary to unlock an encryption key via fingerprint authentication, it is possible to enable this functionality from API 23 (Android 6.0) on supported devices, i.e. devices that meet all of the following requirements: • They have the appropriate fingerprint sensor. • They have at least PIN unlock enabled. • They have at least one registered fingerprint. This function can be used to decrypt and read authentication data stored on the device. Fingerprint validation can be considered a substitute for unlocking the screen via PIN and offers the possibility of obtaining clear access to a value stored in the Android KeyStore with which authentication data can be encrypted. To use the fingerprint authentication feature, it is necessary to ask for the right permissions in AndroidManifest.xml: `<?xml version="1.0" encoding="utf-8"?>`

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.test.minded">

    <uses-permission
        android:name="android.permission.USE_FINGERPRINT" />
    <uses-permission
        android:name="android.permission.USE_BIOMETRIC" />
[ . . . ]
```

The next step will be to create a class to handle fingerprint authentication by extending FingerprintManager.AuthenticationCallback.

As shown below, the startAuth method checks for the presence of fingerprint access control and then calls manager.authenticate to request authentication of a cryptographic object.

```
public class FingerprintHandler extends
FingerprintManager.AuthenticationCallback {

    private CancellationSignal cancellationSignal;
    private Context appContext;

    public FingerprintHandler(Context context) {
        appContext = context;
    }

    public void startAuth(FingerprintManager manager,
                          FingerprintManager.CryptoObject cryptoObject) {
        cancellationSignal = new CancellationSignal();

        if (ActivityCompat.checkSelfPermission(appContext,
                Manifest.permission.USE_FINGERPRINT) !=
                PackageManager.PERMISSION_GRANTED) {
            return;
        }
        manager.authenticate(cryptoObject, cancellationSignal, 0, this, null);
    }

    @Override
    public void onAuthenticationError(int errMsgId, CharSequence errString)
{
        Toast.makeText(appContext, "Authentication error\n" + errString,
            Toast.LENGTH_LONG).show();
    }

    @Override
    public void onAuthenticationHelp(int helpMsgId, CharSequence
helpString) {
        Toast.makeText(appContext, "Authentication help\n" + helpString,
            Toast.LENGTH_LONG).show();
    }
```

```
    @Override
    public void onAuthenticationFailed() {
    Toast.makeText(appContext, "Authentication failed.",
Toast.LENGTH_LONG).show();
    }

    @Override
    public void
onAuthenticationSucceeded(FingerprintManager.AuthenticationResult
result) {
        Toast.makeText(appContext, "Authentication succeeded.",
Toast.LENGTH_LONG).show();
    }

}
```

Fingerprint authentication involves creating an encryption key in the KeyStore, which will be unlocked if fingerprint authentication is successful.

The following class shows an example of an Activity that verifies the prerequisites for implementing fingerprint authentication; it is obtained by calling isKeyguardSecure, checkSelfPermission, hasEnrolledFingerprints which respectively check:

- If the user has set an unlock PIN,
- Whether the application has permission to use the fingerprint
- If the user has registered at least one fingerprint.

Once the prerequisites have been verified, the encryption key is generated via generateKey, then the Cipher is initialized via cipherInit.

Note that the setUserAuthenticationRequired(true) method configures the key so that the user must authorize through fingerprint any access to the key.

The authentication process, and therefore the unlocking of the key present in the KeyStore, takes place by invoking startAuth.

```
public class FingerprintDemoActivity extends AppCompatActivity {

    private static final String KEY_NAME = "example_key";
    private FingerprintManager fingerprintManager;
    private KeyguardManager keyguardManager;
    private KeyStore keyStore;
    private KeyGenerator keyGenerator;
    private Cipher cipher;
    private FingerprintManager.CryptoObject cryptoObject;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fingerprint_demo);

        keyguardManager = (KeyguardManager)
getSystemService(KEYGUARD_SERVICE);
```

```
        fingerprintManager = (FingerprintManager)
getSystemService(FINGERPRINT_SERVICE);

        if (!keyguardManager.isKeyguardSecure()) {
            Toast.makeText(this, "Lock screen security not enabled in
Settings",
                    Toast.LENGTH_LONG).show();
            return;
        }

        if (ActivityCompat.checkSelfPermission(this,
                Manifest.permission.USE_FINGERPRINT) !=
                PackageManager.PERMISSION_GRANTED) {
            Toast.makeText(this, "Fingerprint authentication permission not
enabled",
                    Toast.LENGTH_LONG).show();
            return;
        }

        if (!fingerprintManager.hasEnrolledFingerprints()) {
            //In case there are no registered fingerprints.
            Toast.makeText(this, "Register at least one fingerprint in Settings",
                    Toast.LENGTH_LONG).show();
            return;
        }

        generateKey();
        if (cipherInit()) {
            cryptoObject = new FingerprintManager.CryptoObject(cipher);
            FingerprintHandler helper = new FingerprintHandler(this);
            helper.startAuth(fingerprintManager, cryptoObject);
        }
    }

    protected void generateKey() {
        try {
            keyStore = KeyStore.getInstance("AndroidKeyStore");
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            keyGenerator =
KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
                    "AndroidKeyStore");
        } catch (NoSuchAlgorithmException | NoSuchProviderException e) {
            throw new RuntimeException("Failed to get KeyGenerator
instance", e);
        }

        try {
            keyStore.load(null);
            keyGenerator.init(new
KeyGenParameterSpec.Builder(KEY_NAME,
```

```
                KeyProperties.PURPOSE_ENCRYPT
|KeyProperties.PURPOSE_DECRYPT)
                .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                .setUserAuthenticationRequired(true)

.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
                .build());
            keyGenerator.generateKey();
        } catch (NoSuchAlgorithmException |
InvalidAlgorithmParameterException
                | CertificateException | IOException e) {
            throw new RuntimeException(e);
        }
    }

    public boolean cipherInit() {
        try {
            cipher = Cipher.getInstance(
                    KeyProperties.KEY_ALGORITHM_AES + "/"
                + KeyProperties.BLOCK_MODE_CBC + "/"
                + KeyProperties.ENCRYPTION_PADDING_PKCS7);
        } catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
            throw new RuntimeException("Failed to get Cipher", e);
        }

        try {
            keyStore.load(null);
            SecretKey key = (SecretKey) keyStore.getKey(KEY_NAME, null);
            cipher.init(Cipher.ENCRYPT_MODE, key);
            return true;
        } catch (KeyPermanentlyInvalidatedException e) {
            return false;
        } catch (KeyStoreException | CertificateException
                | UnrecoverableKeyException | IOException
                | NoSuchAlgorithmException | InvalidKeyException e) {
            throw new RuntimeException("Failed to init Cipher", e);
        }
    }

}
```

It is also possible to have more complete examples of authentication via biometrics in paragraph 3.3.10.

References

- https://developer.android.com/about/versions/marshmallow/android-6.0.html#fingerprint-authentication
- http://www.techotopia.com/index.php/An_Android_Fingerprint_Authentication_Tutorial
- https://github.com/doridori/Android-Security-Reference/blob/master/framework/fingerprint.md

3.3.2 SECURE LOGOUT MANAGEMENT

Requirement ID	AUT-002
Priority	**Medium**
Description	Whenever a login functionality is present in the application, the logout feature must be present as well. In case the application uses a cookie-based session mechanism, it is necessary to make sure that, during the logout, the application invalidates the server-side session and deletes any cookie and/or client-side session data. If a persistent client-side authentication token is used, it is recommended to remove it in case of logout and to notify the backend that the token has to be invalidated.
Android	It is always mandatory to invoke the remote logout API in case of explicit logout and to set a session timeout if authentication cookies are used. In case of WebView, it is advisable to use the removeAllCookie method provided by the CookieManager class in order to delete client-side cookies: `CookieManager.getInstance().removeAllCookies(null);` It is also necessary to evaluate the use of clearHistory, clearFormData and clearCache methods that WebView class makes available to remove any data entered during the work session. `webView.clearCache(true);` `webView.clearHistory();` `webView.clearFormData();`
References	• https://developer.android.com/reference/android/webkit/CookieManager.html

3.3.3 USAGE OF TEMPORARY ACCESS TOKENS

Requirement ID	AUT-003

Priority	**High**
Description	An Access Token is an authentication parameter used by accessing an API that requires authentication. The purpose of a temporary token is to provided authentication without sending credentials each time the application needs to communicate with remote API services. Usually, it is included in the HTTP request header Authentication. Temporary tokens can be used in different contexts, the following list of best practices refers to the implementation of a client-server API communication: • Set a token expiration. It's suggested a 10 minutes expiration time; when the expiration is triggered, every API call should be refreshed and the old token invalidated. • Use a minimum token length of 2048 bytes. • The token value should be generated using cryptographically secure random algorithms, in order to be unguessable and unpredictable. • It must be uniquely associated with the device and revocable by the user through the web application. If the session token is issued as a cookie value, it must be protected by applying the HttpOnly and Secure attributes. The Secure attribute instructs the User Agent not to send the cookie through the insecure HTTP protocol to prevent the risk of session theft via network sniffing, while the HttpOnly attribute prevents access to the JavaScript code executed by the user agent. The extension of cookie protection also in the mobile context is justified by two considerations: • Applications can implement hybrid solutions that use browser instances, as such they are exposed to typical attacks from the web world (e.g., XSS); • The mobile application could use the same authentication endpoint used by the web application;
References	• https://tools.ietf.org/html/draft-ietf-oauth-v2-31#section-10.3 • https://tools.ietf.org/html/rfc7519 • https://www.oauth.com/oauth2-servers/access-tokens/access-token-lifetime/ • https://docs.aws.amazon.com/STS/latest/APIReference/API_GetSessionToken.html

3.3.4 PASSWORD SECURITY REQUIREMENTS

Requirement ID	AUT-004
Priority	**High**
Description	The password strength depends on the complexity of the password itself given by the following properties: • Predictability • Length • Entropy A good password policy that requires good constraints on these properties will strongly limit brute force attacks. A brute force attack is a technique that can be used with the aim to guess the correct value of a password by enumerating all possible values or using a dictionary of possible candidates for the searched solutions (e.g. a password value), usually these kind of attacks are automated using software tools. In order to avoid bruteforcing, it is necessary to implement a password policy mechanism to guarantee the complexity of the password chosen by the users. In particular, the password policy mechanism should evaluate the following factors: • The password length, characters complexity (upper and lowercase). • The entropy of the password characters. • The password expiration date. Regarding the best practices for a correct password validation refer to STD-006 Group Identity and Access Management Standard.
References	• https://madiba.encs.concordia.ca/~x_decarn/papers/password-meters-ndss2014.pdf • https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account • https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/04-Authentication_Testing/07-Testing_for_Weak_Password_Policy

3.3.5 PIN SECURITY REQUIREMENTS

Requirement ID	AUT-005
Priority	**High**
Description	When there are weak requirements for PINs that are used inside the application, an attacker could be able to guess their value in order to access sensitive areas. When a mobile application requires a PIN, it is important to follow a number of precautions to keep data secrecy and to mitigate sequence number prediction: • Adopt a 6-digit minimum customer PIN. • Avoid consecutive digits (e.g. 123456). • Avoid more than three equal digits (e.g. 000xxx). • Unless there is a particular requirement, PIN should never be stored on the device and the encryption key has to be stored on a different device. • Do not allow PINs to be in the clear text anywhere in the network or system. • Protect customer PINs using end-to-end application layer encryption. • Differentiate PINs for different channels of different risk levels; advise customers to use different PINs for different channels. Furthermore, to avoid brute force attacks it's recommended to set a limit to the number of attempts of PIN entering. Finally, the PIN must always be entered via an application PIN PAD as discussed in paragraph 3.3.25.
References	• https://github.com/OWASP/owasp-mstg/blob/master/Document/0x04e-Testing-Authentication-and-Session-Management.md

3.3.6 VERIFY PRESENCE OF LOCAL AUTHENTICATION

Requirement ID	AUT-006
Priority	**Medium**

Description	If there are no local authentication mechanisms set on the device (such as biometric authentication or PIN) an attacker could access the device in order to use the installed applications or access saved data. It is suggested to make the application check that the user has already set a local authentication mechanism to unlock and use the device in order to allow only the device owner to access and use the installed applications. In this way, even if a mobile app retains the authentication of the user after closing the app, malicious users that have physical access to the device won't be able to unlock it. It is suggested to check the presence of these local authentication mechanisms via the APIs offered by the platform.
Android	Starting with Android 6 (SDK 23), there is a method to check if the device is protected by a pin, password, etc, as shown in the example below: `KeyguardManager manager = (KeyguardManager) context.getSystemService(Context.KEYGUARD_SERVICE);` `if(manager.isDeviceSecure()){` `  // device is secure` `}else{` `  // device is not secure` `}`
References	• https://developer.android.com/reference/android/app/KeyguardManager#isDeviceSecure() • https://github.com/OWASP/owasp-mstg/blob/master/Document/0x04e-Testing-Authentication-and-Session-Management.md

3.3.7 AUTHENTICATE USING ACTIVE DIRECTORY

Requirement ID	AUT-007
Priority	**Low**
Description	For mobile enterprise applications that should be used by the company's employees it is suggested to use a centralized authentication via Active Directory in order to avoid ad hoc credentials that could be difficult to revoke. If the mobile application uses custom authentication mechanisms based on ad hoc credentials, in case of necessity it could be difficult to reset an account

	and, in addition, there would be an additional cost in securely managing another database with all the credentials. If the application uses Active Directory authentication instead, it is easier to manage company accounts in order to revoke or upgrade access for the users of the application.
Android	For enterprise applications, if possible, it is suggested to use Active Directory authentication and make sure that LDAP data are synchronized in order to be sure that only valid company accounts can access the applications.
References	• https://docs.microsoft.com/it-it/azure/app-service-mobile/app-service-mobile-android-get-started-users • https://msdn.microsoft.com/en-us/library/hh871909.aspx

3.3.8 PROTECT FROM USER ENUMERATION

Requirement ID	AUT-008
Priority	**Medium**
Description	An attacker could be able to identify if a username is valid or not for the application by trying to interact with the authentication system. This happens if the application replies in two different ways if a username exists or not, regardless of the password. Therefore, an application is affected by this vulnerability if, given a valid username and a wrong password, the system replies with a message like the following: `Login failed for User foo: invalid password` Otherwise, it replies with the following message when the user does not exist on the system: `Login failed for User foo: invalid Account`
Android	Be sure that the application does not provide too many details during the authentication phase and always provide the same generic error message. Also, in case of a non-existing user on platform, always display a generic error message like the following:

	Wrong Credentials Make sure that the server does not use different response times depending on whether the user exists or not in authentication process, to prevent an attacker from inferring through this mechanism, even if unchanged error message is provided, and to make user enumeration. Finally, it is worth noting that the login functionality is not the only one that can be abused to enumerate users of the platform. Another functionality is, for example, password recovery.
References	• https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account • https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html#Authentication_and_Error_Messages

3.3.9 PROTECT FROM BRUTEFORCING

Requirement ID	AUT-009
Priority	**Medium**
Description	Brute forcing of access credentials consists in trying to guess the password of a user for which the username is known. This kind of attack is performed by using automatic tools that, given a username, try to guess the password making several tries. A particular case of bruteforcing is the dictionary attack in which the passwords are retrieved starting from a list of words instead of all possible sequences of characters. The second method obviously would allow an attacker to reach the result in much less time. The most common protection against these attacks is to implement account lockout, which prevents any more login attempts for a period after a certain number of failed logins (for more information refer to STD-006 Group Identity and Access Management Standard.) The counter of failed logins should be associated with the account itself, rather than the source IP address, in order to prevent an attacker making login attempts from a large number of different IP addresses. When designing an account lockout system, care must be taken to prevent it being used to cause a denial of service (DoS) by locking out other users' accounts. For this reason, rather than implementing a fixed lockout duration

	it is suggested to use an exponential lockout, where the lockout duration starts as a very short period (e.g., one second), but doubles after each failed login attempt. Adding the use of an effective CAPTCHA can help to prevent automated login attempts against accounts.
References	• https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/04-Authentication_Testing/03-Testing_for_Weak_Lock_Out_Mechanism • https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/03-Identity_Management_Testing/02-Test_User_Registration_Process

3.3.10 AUTHENTICATION WITH BIOMETRIC FACTORS

Requirement ID	AUT-010
Priority	**High**
Description	The usage of biometric authentication in mobile applications it's often implemented to facilitate users during the login phase. After executing the first login using standard credentials (e.g. username and password) it is possible to configure the application to use biometric access as primary factor and pin or password as fallback. In order to allow biometric access, the device must be secured with a pin or password. For more details, refer to the requirement 3.3.6. For biometric access to the application to be implemented correctly, a suitably configured key must be generated within the system Keystore which must be used to encrypt and decrypt the secret necessary to authenticate the user on the system, such as a token login or the credentials themselves. Once encrypted, this secret must be saved and managed securely as described in paragraph 3.3.22. In order to properly implement biometric access to the application, it is necessary to generate a key properly configured inside the system Keystore. That key must be used to encrypt and decrypt the secret needed to authenticate the user on the system, such as an access token or the credentials themselves. Once encrypted, this secret must be saved and securely managed as described in paragraph 3.3.22. For further details on the generation of the key in the system Keystore and for its correct configuration, refer to the paragraph 3.3.1. Furthermore, starting from Android 11 it is possible to define the minimum security class required to enable biometric authentication.

Android	Below is an example of code that allows access to an application through the Android BiometricPrompt class. The example below is valid both for use with the native Android APIs, and with the Jetpack / AndroidX libraries which will provide backward compatibility with previous versions of the operating system. It should be noted that this flow at the end uses the encryption, since at the first access the biometry is used to encrypt the user credentials just entered. In the following executions, it will be necessary to use decryption in order to recover the encrypted credentials and log in to the application. Key generation: `private fun generateSecretKey(keyGenParameterSpec: KeyGenParameterSpec) {` `  val keyGenerator = KeyGenerator.getInstance(` `  KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore")` `  keyGenerator.init(keyGenParameterSpec)` `  keyGenerator.generateKey()` `}` `private fun getSecretKey(): SecretKey {` `  val keyStore = KeyStore.getInstance("AndroidKeyStore")` `  keyStore.load(null)` `  return keyStore.getKey(KEY_NAME, null) as SecretKey` `}` `private fun getCipher(): Cipher {` `  return Cipher.getInstance(KeyProperties.KEY_ALGORITHM_AES + "/"` `  + KeyProperties.BLOCK_MODE_CBC + "/"` `  + KeyProperties.ENCRYPTION_PADDING_PKCS7)` `}` `…` `generateSecretKey(KeyGenParameterSpec.Builder(` `  KEY_NAME,` `  KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT)` `  .setBlockModes(KeyProperties.BLOCK_MODE_CBC)` `.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)` `  .setUserAuthenticationRequired(true)` `  .setInvalidatedByBiometricEnrollment(true)` `  .build())` `…` Starting the authentication flow with the push of a button: `biometricLoginButton.setOnClickListener {` `  val cipher = getCipher()` `  val secretKey = getSecretKey()` `  cipher.init(Cipher.ENCRYPT_MODE, secretKey)`

	promptInfo = BiometricPrompt.PromptInfo.Builder() .setTitle("Login tramite accesso biometrico") .setSubtitle("Accedi con le tue credenziali biometriche") .setAllowedAuthenticators(BIOMETRIC_STRONG or DEVICE_CREDENTIAL) .build() biometricPrompt.authenticate(promptInfo, BiometricPrompt.CryptoObject(cipher)) } The operating system will then show a login prompt that will ask the user to enter the biometric identity (fingerprint, iris, etc.). Once the authentication is complete, the following method will be invoked by the operating system. In particular, the result.cryptoObject.cipher parameter will contain the key that must be used to carry out the operations necessary to save and retrieve the access credentials: override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) { val encryptedInfo: ByteArray = result.cryptoObject.cipher?.doFinal(plaintext-string.toByteArray(Charset.defaultCharset())) // Save the credentials or retrieve them in subsequent accesses }
References	• https://developer.android.com/training/sign-in/biometric-auth • https://source.android.com/compatibility/android-cdd#7_3_10_biometric_sensors • https://www.raywenderlich.com/18782293-android-biometric-api-getting-started#toc-anchor-003 • https://developer.android.com/codelabs/biometric-login#0

3.3.11 CODE OBFUSCATION

Requirement ID	RE-001
Priority	**High**
Description	The code obfuscation technique is implemented in ProGuard and improved in DexGuard. • https://www.guardsquare.com/blog/dexguard-vs.-proguard

	The default configuration dexguard-release.pro already covers all obfuscation settings to a good extent, with the default obfuscation level set to low. In order to have a reasonable level of obfuscation it is important to adopt the following techniques: • Perform code obfuscation through code rewrite, accomplished by adding dead code and increasing code complexity to make automatic decompilation difficult. • Renaming of packages, classes and methods using random names or names with no semantic meaning. • Automatic transformation of sensitive methods direct call to indirect call via reflection and encryption of classes and method names used for reflection transformation. In particular, the following settings should be considered as important: **-obfuscatecode**,[low, medium, high] [class_filter] # Obfuscation is a default option. This directive can be used to provide a specific obfuscation level on specific classes. Default: obfuscatecode,low * -flattenpackagehierarchy [package_name] # Flattens the package hierarchy for the given package_name. Developers have to pay attention while using this directive and the original code uses reflection API applied to `package_name`. DexGuard, in that case will not be able to rewrite the reflected call with the new package name. **-accessthroughreflection**[,encryptstrings] class_specification # Forces the rewriting of direct method calls matching class_specification filter to indirect call via reflection. This settings makes static analisys more difficult. It's recommended, unless there are particular performance requirements, to use `encryptstrings`. it encrypts classes and methods names used in the transformation to reflection call.
Android	Obfuscatecode usage -obfuscatecode [,strength] class_specification This is the directive used to obfuscate specific methods. If it is not specified a class or a method, all method and nested classes in the class will be obfuscated. The default obfuscation strength is Low. After identifying the directive, Dexguard will rewrite the class and methods code in order to increase reverse engineering complexity.

WARNING

Since this approach is performed by static code analysis, it works well when "reflection" is not used.

Flattenpackagehierarchy usage:

Assuming code contains the following classes and packages:

```
mycompany.myapplication.MyMain
mycompany.myapplication.Foo
mycompany.myapplication.Bar
mycompany.myapplication.extra.FirstExtra
mycompany.myapplication.extra.SecondExtra
mycompany.util.FirstUtil
mycompany.util.SecondUtil
```

The default package name obfuscation setting will produce the following result:

```
mycompany.myapplication.MyMain
mycompany.myapplication.a
mycompany.myapplication.b
mycompany.myapplication.a.a
mycompany.myapplication.a.b
mycompany.a.a
mycompany.a.b
```

If the following directive is added to the custom configuration:

```
-flattenpackagehierarchy 'fakepackagename'
```

The result will be:

```
mycompany.myapplication.MyMain
mycompany.myapplication.a
mycompany.myapplication.b
fakepackagename.a.a
fakepackagename.a.b
fakepackagename.b.a
fakepackagename.b.b
```

WARNING:

	Since this approach is performed by static code analysis, it works well when "reflection" is not used. Accessthroughreflection usage: With the class and method of the previous example and the following settings: `-accessthroughreflection[,encryptstrings] class` It specifies to substite the direct access of class and members with reflection. When obfuscation and other renaming directives are applied, names will be rewritten according to DexGuard internal mapping. Finally, when developers explicitly use the reflection approach, it is recommended to use “encryptstrings” directive to obfuscate methods and class names used as reflection arguments.

3.3.12 ENCRYPT CLASSES WITH SENSITIVE CODE

Requirement ID	RE-002
Priority	**High**
Description	Class encryption is another technique that can be used to obfuscate code in order to increase the difficulty for reverse engineering performed through static code analysis. We recommend using this directive for classes with highly sensitive content. -encryptclasses [class_filter] -encryptclasses class_specification Specifies that classes whose names either match the given filter or class specification should be encrypted. When using a class specification, specified fields and methods are ignored as classes will always be encrypted as a whole. This makes it more difficult to disassemble or decompile them. "Obfuscated" would be a better word, since the processed code necessarily has to be able to reverse the encryption. It therefore increases the code size and introduces processing overhead at runtime, whenever the class is loaded or accessed. However, it raises the bar for any reverse engineering attempts. For example, if you have some sensitive license checking class, you may want to protect it by encrypting it. Only applicable when obfuscating. You can additionally protect encrypted class by obfuscating their code with obfuscation strength medium or high. This will render the classes unusable if they are manually extracted from the DEX file. Code obfuscation should

	not be applied however on performance-sensitive code that is executed very frequently (e.g., tight code loops). Counter-indications: It is not possible to encrypt classes that are explicitly preserved from obfuscation (in your configuration), extended by non-encrypted classes, or created by reflection (for instance because they are referenced from XML files). Note: When encrypting classes in library projects, the code must set a temporary directory: System.setProperty("java.io.tmpdir", getDir("files", Context.MODE_PRIVATE).getPath()); Caveat: When encrypting classes in library projects, the encrypted classes must not contain references to classes or class members that are later on obfuscated in the final application projects. Once encrypted, classes can no longer be changed, so their references would become invalid. If encrypted classes do contain references to other non-encrypted classes in their library projects, these referenced classes and class members must be preserved from obfuscation in the application projects. If your requirements allow it, it is easier to encrypt classes when processing the final application. Performance tip: Every access from an external class to an encrypted class carries some overhead, due to reflection. If performance is important in this part of your code, you can reduce the overhead by accessing the class through an interface that is not encrypted.
Android	The following settings instruct DexGuard to perform the following operations: • Encrypt the bytecode of “TestLicense”. • Generate a class that uses “ClassLoader” and calls “Class.forName” that loads the class at runtime. -encryptclasses com.application.TestLicense

3.3.13 LIMIT THE USAGE OF WHITELIST –KEEP*

Requirement ID	RE-003
Priority	**Medium**
Description	Sometimes code obfuscation can result in application runtime errors due to special implementations such as reflection or RPC or in case DexGuard performs file deletions or inserts dead code for incorrect automatic assumptions.

	The command that asks DexGuard to not obfuscate specific classes are identified by the prefix "**-keep**". -keep [,modifier,...] class_specification Specifies classes and class members (fields and methods) to be preserved as entry points to your code. For example, in order to keep an application, you can specify the main class along with its main method. In order to process a library, you should specify all publicly accessible elements. -keepclassmembers [,modifier,...] class_specification Specifies class members to be preserved, if their classes are preserved as well. For example, you may want to keep all serialization fields and methods of classes that implement the Serializable interface. -keepclasseswithmembers [,modifier,...] class_specification Specifies classes and class members to be preserved, on the condition that all of the specified class members are present. For example, you may want to keep all applications that have a main method, without having to list them explicitly. Although these directives are very useful for solving post-obfuscation problems, they must be used with attention as they instruct DexGuard to explicitly do not obfuscate specific parts of the code. To analyze and debug such situations it is suggested to use the "**printusage**" directive: -printusage [filename] Specifies to list dead code of the input class files. The list is printed to the standard output or to the given file. For example, you can list the unused code of an application. Only applicable when shrinking.

3.3.14 PROTECTION AGAINST INFORMATION DISCLOSURE VIA STACKTRACE

Requirement ID	RE-004
Priority	**Medium**
Description	During the application execution errors might be triggered and managed through software exceptions. These exceptions contain the stacktrace of the errors, such as: Caused by: java.lang.ArithmeticException: divide by zero at com.example.stacktraceeample.MainActivity.onCreate(MainActivity.kt:13) at android.app.Activity.performCreate(Activity.java:7009) at android.app.Activity.performCreate(Activity.java:7000)

	at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1214) The stacktrace is an important source of information for developers in order analyze errors and fix them. On the other hand, it may contain lines and methods names or filenames, so it can disclose useful information also to an attacker who want to perform a reverse engineering analysis. Code obfuscation allows to minimize the information contained in a stacktrace. The settings shown in the following examples are present in the DexGuard default configuration.
Android	To get the stacktrace with obfuscated information, which can be remapped on the original code, developers can use the following settings: -printmapping FILEMAPPING.map # saves a map between original names and obfuscated ones -renamesourcefileattribute SourceFile # resolves the leakage of filename in each .class by overwriting it to a given string -keepattributes SourceFile,LineNumberTable # keeps the lineNumber Table to give obfuscated stack traces that can be mapped back to original code using `retrace` utility and FILEMAPPING.map file. These settings keep all source file attributes but replace their values with the string "SourceFile". Any strings could be used. This string is already present in all class files, so it doesn't take up any extra space. It also keeps the line number tables of all methods. Whenever both of these attributes are present, the Java run-time environment will include line number information when printing out exception stacktraces. The information will only be useful if it can map the obfuscated names back to their original names, so it saves the mapping to a file “out.map”. The information can then be used by the ReTrace tool to restore the original stack trace.

3.3.15 STATIC RESOURCES ENCRYPTION

Requirement ID	RE-005
Priority	**Medium**
Description	Static resources encryption increases the difficulty for an attacker to find secrets in decompiled or disassembled code. For sake of completeness, the correct term of this technique is resources obfuscation since resources are encrypted at obfuscation time and decrypted and kept in memory during application execution by internal methods. DexGuard settings for resources encryption are: `-encryptassetfiles` [file_filter] Specifies the Android asset files that should be encrypted. Asset files are stored in the assets directory and can contain any data. The obfuscation step can automatically encrypt them and make sure they are decrypted on the fly at run-time. In order for this to work, the assets must be loaded using one of the AssetManager.open methods. Only applicable when obfuscating Android code. Note: If the processed application contains at least one call to AssetManager.open(String) with a non-constant string argument, DexGuard will encrypt all assets that match the specified file filter. If there are assets that are loaded via other mechanisms, make sure they do not match the specified file filter. `-encryptresourcefiles` [file_filter] Specifies to encrypt Android resource files. Resource files are stored in the res directory and can contain application resources such as layout XML files. The obfuscation step can automatically encrypt them and make sure they are decrypted on the fly at run-time. Supported resources: res/layout, res/menu, and res/xml files. Counter-indication: app widgets can't decrypt resources. Don't encrypt resource files that are accessed by app widgets. It is possible to get a list of all the resources encrypted by DexGuard by using the “-print*” directives. This directive can be very useful in case of execution issues. `-printclassencryption` [filename] `-printstringencryption` [filename] `-printassetencryption` [filename] `-printresourceencryption` [filename] `-printnativelibraryencryption` [filename]

Android	The following example can be used to encrypt assets. `fun displayOneImage(imageViewbyCode: ImageView) {` `  if (imageViewbyCode != null)` `  {` `  imageViewbyCode.setVisibility(View.GONE)` `  }` `  imageViewbyCode.setVisibility(View.VISIBLE)` `  val assetManager = getAssets()` `  try` `  {` `  val `is` = assetManager.open("img/image1.png")` `  val bitmap = BitmapFactory.decodeStream(`is`)` `  imageViewbyCode.setImageBitmap(bitmap)` `  }` `  catch (e: IOException) {` `  Log.e(TAG, e.message)` `  }` `  }` The following setting asks DexGuard to look for the “AssetManager.open” method call, and only in that case the asset will be encrypted: `-encryptassetfiles assets/img/image1.png` As before, resource encryption, it is automatically performed (without the requirement of particular code paradigms) as follows: `-encryptresourcefiles res/**/*` WARNING: Pay attention to the resources used by app widgets, in fact DexGuard is not able to decrypt these resources at runtime, it is therefore recommended to use the -keepresources directive for these exceptions.

3.3.16 STATIC STRING ENCRYPTION

Requirement ID	RE-006
Priority	**Medium**

Description	Static strings encryption increases the difficulty for an attacker to find secrets in decompiled or disassembled code. The correct term of this technique is strings obfuscation, since strings are encrypted at obfuscation time and decrypted and kept in memory during application execution by methods internal to the application itself. DexGuard settings for string encryption are: -encryptstrings [string_filter] #All strings matching the filter will be encrypted (wildcard * matches all) -encryptstrings class_specification # all strings in the matching class or classes are encrypted With specified fields, the matching final String constants are encrypted. With specified methods, all strings in the matching methods are encrypted. -encryptresources [name_filter] # Specifies the Android resources to be encrypted. The filter is applied to strings of the form "type/name", for example "string/apiKey". Currently only String resources are supported. If a resource is also excluded from obfuscation using -keepresources, this takes precendence over encryption. Counter-indication: resource Strings referenced from other XML files can't be encrypted. Besides performing encryption on particular static secrets, it's suggested, in case of access through reflection, to do the same with methods and classes names to make more difficult a reverse engineering analysis. **WARNING:** To increase decryption performance in methods and classes where performance is a mandatory requirement, it's suggested to define strings as: private static String This way decryption takes place only once the class it belongs is initialized. Be careful to not declare it as final because the compiler could change it to inline in the belonging class and perform decryption every time an object is created. **WARNING:** Please note that DexGuard does not support encryption of static strings in the "Interface classes". Though it supports "final strings" in the "Interface classes". Below few drawbacks related to this feature: • It increments the software size.

	• It introduces an overhead due to runtime decryption.
Android	The following directive obfuscate “clicker.jar” into “clicker_ob.jar” and encrypts all strings in all classes. -injars Compiled/clicker.jar -outjars Compiled/clicker_ob.jar -libraryjars <java.home>/lib/rt.jar -printmapping clicker.map -keep public class com.amazon.sample.buttonclicker.Clicker { public static void main(java.lang.String[]); } -renamesourcefileattribute SourceFile -keepattributes SourceFile,LineNumberTable -encryptstrings *

3.3.17 PREVENT TAMPERING

Requirement ID	RE-007
Priority	**Medium**
Description	It is possible to perform a tamper detection in order to check if the application apk has been changed. DexGuard can perform this check by verifying the integrity of the apk and the signing certificate. To implement this control, it is possible to call the following APIs: com.guardsquare.dexguard.runtime.detector.TamperDetector.checkApk(context,OK) # Check if the apk has been modified com.guardsquare.dexguard.runtime.detector.CertificateChecker.checkCertificate(context, OK) # Check if the application has been signed with the compilation certificate. Since the implementation is very specific to the application that we need to protect, it is necessary to obfuscate or encrypt the methods that perform these checks, otherwise they could be easily spotted by an attacker and removed. In addition, when a tampering attempt is detected, it is suggested to make the application work with limited capabilities and to not inform the user with

	any kind of error messages. In this way it is more difficult for an attacker to understand when this check is actually performed.
Android	Since the apk modification is done on the static data, tampering can be verified when starting the application. It is necessary to put this control in a class of its own so that it can be encrypted with DexGuard: /** This utility class performs tamper detection and creates the View. This functionality can be placed in a separate class so that it can be encrypted, as an additional layer of protection when tampering is detected. The Activity itself cannot be encrypted, for technical reasons, but an inner class or another class is fine anyway. */ private class Delegate { fun checkAndInitialize() { // Context for many methods is needed. val context = this@HelloWorldActivity /* A developer can choose their own value (or values) for OK, to make the code less predictable. */ val OK = 1 /* Through the DexGuard library at runtime it detects if the apk has been modified or repackaged in any way (with jar, zip, jarsigner, zipalign or any other tool), after DexGuard has packed it. The return value is the value of the optional integer argument OK (default = 0) if the apk is unchanged. */ val apkChanged = TamperDetector.checkApk(context, OK) /* Using the DexGuard library at runtime it detects if the apk has been rewritten with a different certificate, after DexGuard has packaged it. The return value is the value of the optional integer argument OK (default = 0) if the certificate is always the same. */ val certificateChanged = CertificateChecker.checkCertificate(context, OK) /* The developer can also explicitly pass the MD5 hash of a certificate, if the application is signed only after DexGuard has packaged it. You can print the MD5 hash of your keystore certificate with: keytool -list -keystore my.keystore If the developer wants to extract the MD5 hash from the latest version of the application, he can print it as follows: keytool -printcert -v -jarfile my.apk If the developer is posting on the Amazon Store, they can find the MD5 hash at: Amazon Apps & Games Developer Portal> Binary File (s) Tab> Settings> My Account. With the MD5 hash, you can use one of these controls:*/ // CertificateChecker.checkCertificate(context, // "FA:F8:0A:CB:26:C9:08:DD:3F:E4:A4:76:1B:37:3E:C1", OK); // CertificateChecker.checkCertificate(context, // "FAF80ACB26C908DD3FE4A4761B373EC1", OK); // CertificateChecker.checkCertificate(context,

```
    // 0xFAF80ACB, 0x26C908DD, 0x3FE4A476, 0x1B373EC1, OK);
    // CertificateChecker.checkCertificate(context,
    // 0xFAF80ACB26C908DDL, 0x3FE4A4761B373EC1L, OK);
    //
    /* If the developer specifies a string, they should make sure it is largely
encrypted. Show a message. */
    val view = TextView(context)
    view.setText(if ((apkChanged == OK && certificateChanged == OK))
                "Hello world!"
                else
                "Tamper alert!")
    view.setGravity(Gravity.CENTER)
    // Change the background color if someone has changed the archive.
    if ((apkChanged != OK || certificateChanged != OK))
    {
      view.setBackgroundColor(Color.RED)
    }
    setContentView(view)
    // Briefly show a comment.
    val comment = if (certificateChanged != OK)
    "The certificate is not the expected certificate"
    else if (apkChanged != OK)
    "The application archive has been modified"
    else
    "The application has not been modified"
    Toast.makeText(context, comment, Toast.LENGTH_LONG).show()
  }
}
```

3.3.18 WHITEBOX CRYPTOGRAPHY

Requirement ID	RE-008
Priority	**Low**
Description	The challenge that whitebox cryptography aims to address is to implement a cryptographic algorithm in such a way that cryptographic assets remain secure even when subject to whitebox attacks. Whitebox cryptography can be useful in those cases where an encryption key or other sensitive data needs to be hardcoded into the application. It works by encrypting and decrypting dynamic application data with a whitebox implementation of a cryptographic algorithm such as AES. The whitebox encryption key is weaved in the decryption algorithm in such a way that it can't be extracted because it becomes part of the algorithm itself.

	The advantages of this approach consist in the fact that some secrets and some sensitive parts of the mobile application are encrypted making the effort of reverse engineering quite considerable. Be aware that the attacks an adversary can perform are very dependent on the construction of the whitebox implementation and the properties of the underlying cipher. Hence, widely applicable attacks are difficult to deploy, and there do not exist any automatic tools to break them. There are, on the other hand, disadvantages such as the facts that: • It is computationally more expensive resulting in slowing down the application, which could also reduce device battery life significantly. • It has a large code footprint. Given the previous drawbacks, it should be considered to use whitebox cryptography. For example, applying the asset whitebox plugin to a large number of assets would probably expand the application code footprint too much. On the other hand, the overhead of protecting a single sensitive class or native library with a whitebox encryption plugin is generally acceptable in most cases.

3.3.19 ANTI-ROOT CONTROLS

Requirement ID	RT-001
Priority	**Medium**
Description	Anti-rooting check can be performed at runtime to identify if the application is running on a rooted device. Anti-root control is available from DexGuard via API in the package: com.guardsquare.dexguard.runtime.detection.RootDetector.isDeviceRooted(context, OK); # Check if the application is running on a rooted device Since custom code is needed to implement these checks, it is necessary to hide obfuscating the code, failing to do so could allow an attacker to find it and remove it modifying the pseudocode. When the detection found an issue, it is important to implement a behavior that cannot be easily detected by an attacker. For instance, the application could limit the functionalities provided to the user. This helps to mitigate the possibility that an attacker bypasses the detection mechanism.

Android	An attacker can switch the environment at any time while the application is running, so it is advisable to implement controls in an asynchronous class via a background service. It is necessary to put this control in a class of its own so that it can be encrypted with DexGuard:

```
/ *
This utility performs debug detection, emulator detection and root privilege detection and displays the View. If the environment is okay, the application works normally and displays "Environment is ok". Otherwise it displays information about the environment.

This functionality can be placed in a separate class so that they can be encrypted, as an additional layer of protection along with tampering detection and some essential codes. The Activity itself cannot be encrypted, for technical reasons, but an inner class or another class is fine.

The Delegate class is implemented as an asynchronous Activity. This ensures that the small overhead introduced by environment controls does not affect the main application thread.
*/

private class Delegate : AsyncTask<Void, Int, Boolean>() {
    override fun onPreExecute() {
        super.onPreExecute()

        for (imageView in envCheckImageViewList) {
            imageView.setImageDrawable(null)
        }
    }
    protected override fun onProgressUpdate(vararg values: Int) {
        super.onProgressUpdate(values)

        val imageView = envCheckImageViewList.get(values[0])
        imageView.setImageDrawable(if (values[1] == 0) okIcon else
detectedIcon)
    }

    /**
     * This method will be performed in a separate thread.
     */
    override fun doInBackground(vararg voids: Void): Boolean? {
        // Context for many methods is needed.
        val context = this@HelloWorldActivity
        /* A developer can choose their own value (or values) for OK, to make
the code less predictable. */
        val OK = 1

        /* Through the DexGuard library at runtime it is detected if the device
is rooted. The return code is OK if not. */
        val isDeviceRooted = RootDetector. isDeviceRooted(OK)
        publishProgress(5, if (isDeviceRooted == OK) 0 else 1)
```

```
        return true
    }
}
```

3.3.20 ANTI-DEBUGGING CONTROLS

Requirement ID	RT-002
Priority	**Medium**
Description	The anti-debugging check can be run at runtime to identify if the application is being analyzed while it is running using either of these techniques. These controls are available from DexGuard via API in the package: `com.guardsquare.dexguard.runtime.detector.DebugDetector.isDebuggable (context, OK);` # Check if the application has the debug flag set to true `com.guardsquare.dexguard.runtime.detector.DebugDetector.isDebuggerConnected(OK);` # Check if the application is debugging `com.guardsquare.dexguard.runtime.detector.DebugDetector.isSignedWithDebugKey(context, OK);` # Check if the application is signed with a debug key `com.guardsquare.dexguard.runtime.detector.EmulatorDetector.isRunningInEmulator(context, OK);` # Check if the application is running on an emulator Since custom code is needed to implement these checks, it is necessary to hide obfuscating the code, failing to do so could allow an attacker to find it and remove it modifying the pseudocode. When the detection found an issue, it is important to implement a behavior that cannot be easily detected by an attacker. For instance, the application could limit the functionalities provided to the user. This helps to mitigate the possibility that an attacker bypasses the detection mechanism.
Android	An attacker can switch the environment at any time while the application is running, so it is advisable to implement controls in an asynchronous class via a background service.

It is necessary to include this control in a class of its own so that it can be encrypted with DexGuard:

```
/ *
This utility performs debug detection, emulator detection and root privilege detection and displays the View. If the environment is okay, the application works normally and displays "Environment is ok". Otherwise it displays information about the environment.

This functionality can be placed in a separate class so that they can be encrypted, as an additional layer of protection along with tampering detection and some essential codes. The Activity itself cannot be encrypted, for technical reasons, but an inner class or another class is fine.

The Delegate class is implemented as an asynchronous Activity. This ensures that the small overhead introduced by environment controls does not affect the main application thread.
*/
private class Delegate:AsyncTask<Void, Int, Boolean>() {
  protected override fun onPreExecute() {
    super.onPreExecute()
    for (imageView in envCheckImageViewList)
    {
      imageView.setImageDrawable(null)
    }
  }
  protected fun onProgressUpdate(vararg values:Int) {
    super.onProgressUpdate(values)
    val imageView = envCheckImageViewList.get(values[0])
    imageView.setImageDrawable(if (values[1] == 0) okIcon else
detectedIcon)
  }
  /**
 * This method will run in a separate class.
 */
  protected fun override doInBackground(vararg voids:Void):Boolean {
    // Context for many methods is needed.
    val context = this@HelloWorldActivity
    /* A developer can choose their own value (or values) for OK, to make
the code less predictable. */
    val OK = 1
    // Using the DexGuard library at runtime it detects if the application is
debuggable. The return code is OK if it isn't.
    val isDebuggable = DebugDetector.isDebuggable(context, OK)
    publishProgress(0, if (isDebuggable == OK) 0 else 1)
    // Using the DexGuard library at runtime it detects if a debugger is
connected to the application. The return code is OK if not.
    val isDebuggerConnected = DebugDetector.isDebuggerConnected(OK)
    publishProgress(1, if (isDebuggerConnected == OK) 0 else 1)
    // Using the DexGuard library at runtime, it detects whether the
application is signed with a debug key. The return code is OK if not.
    val isSignedWithDebugKey =
DebugDetector.isSignedWithDebugKey(context, OK)
```

	publishProgress(2, if (isSignedWithDebugKey == OK) 0 else 1) // Using the DexGuard library at runtime, it detects whether the application is running on an emulator. The return code is OK if not. val isRunningInEmulator = EmulatorDetector.isRunningInEmulator(context, OK) publishProgress(3, if (isRunningInEmulator == OK) 0 else 1) return true } }

3.3.21 ANTI-HOOKING CONTROLS

Requirement ID	RT-003
Priority	**Medium**
Description	Anti-hooking checks can be performed at runtime to identify if the application is being scanned while it is running using either of these techniques. This control is available from DexGuard via API in the package: com.guardsquare.dexguard.runtime.detector.HookDetector.isApplicationHooked(OK) Since custom code is needed to implement this checks, it is necessary to hide obfuscating the code, failing to do so could allow an attacker to find it and remove it modifying the pseudocode. When the detection found an issue, it is important to implement a behavior that cannot be easily detected by an attacker. For instance, the application could limit the functionalities provided to the user. This helps to mitigate the possibility that an attacker bypasses the detection mechanism.
Android	An attacker can switch the environment at any time while the application is running, so it is advisable to implement controls in an asynchronous class via a background service. It is necessary to put this control in a class of its own so that it can be encrypted with DexGuard: / * This utility performs debug detection, emulator detection and root privilege detection and displays the View. If the environment is okay, the application

works normally and displays "Environment is ok". Otherwise it displays information about the environment.

This functionality can be placed in a separate class so that they can be encrypted, as an additional layer of protection along with tampering detection and some essential codes. The Activity itself cannot be encrypted, for technical reasons, but an inner class or another class is fine.

The Delegate class is implemented as an asynchronous Activity. This ensures that the small overhead introduced by environment controls does not affect the main application thread.
*/

```
private class Delegate : AsyncTask<Void, Int, Boolean>() {
    override fun onPreExecute() {
        super.onPreExecute()

        for (imageView in envCheckImageViewList) {
            imageView.setImageDrawable(null)
        }
    }
    protected override fun onProgressUpdate(vararg values: Int) {
        super.onProgressUpdate(values)

        val imageView = envCheckImageViewList.get(values[0])
        imageView.setImageDrawable(if (values[1] == 0) okIcon else
detectedIcon)
    }

    /**
     * This method will be performed in a separate thread.
     */
    override fun doInBackground(vararg voids: Void): Boolean? {
        // Context for many methods is needed.
        val context = this@HelloWorldActivity
        /* A developer can choose their own value (or values) for OK, to make
the code less predictable. */
        val OK = 1

        // Using the DexGuard library at runtime it detects if the application is
subject to hooking. The return code is OK if not.        val
isApplicationHooked = HookDetector.isApplicationHooked(OK)
        publishProgress(5, if (isApplicationHooked == OK) 0 else 1)
        return true
    }
}
```

3.3.22 ENCRYPTION OF PERSONAL DATA

<table>
<tr><td>Requirement ID</td><td>DS-001</td></tr>
<tr><td>Priority</td><td>High</td></tr>
<tr><td>Description</td><td>If the application saves on the device sensitive information in clear or using insecure encryption algorithms, an attacker could easily access that information.

Usually, application saves sensitive information locally on the same device on which they are running.

That sensitive information must be encrypted using a strong encryption algorithm and encryption key, in order to be protected in case of unauthorized access or rooted devices. This implies the implementation of secure strategies in order to save the secret on the client in a secure manner.

Furthermore, the application must implement modern and up-to-date encryption algorithms that are known for their robustness and security. The key length should be appropriate for the chosen algorithm.

These are the current standard for encryption algorithms.

<table>
<tr><td>Asymmetric encryption</td><td>RSA with minimum key length of 2048 bits</td></tr>
<tr><td>Symmetric encryption</td><td>AES with minimum key length of 256 bit</td></tr>
<tr><td>Hashing algorithms</td><td>SHA512</td></tr>
</table>

Finally, the application must limit the amount of local data saved. For instance, it’s better to save only the following sensitive information on the device:

Authentication token
Name and surname
Last 4 digits of identification codes
Seed or other cryptographic information

</td></tr>
<tr><td>Android</td><td>Starting with Android version 4.3 (API 18), Android allows you to securely store encryption keys via KeyStore; these keys can then be used for encryption and decryption of local data belonging to the mobile application.

Below is an example of creating the AES symmetric encryption key inserted in the KeyStore; you can see that a custom alias is associated with the encryption key:

KeyGenerator keyGenerator = KeyGenerator.getInstance(</td></tr>
</table>

```
  KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
keyGenerator.init(
  new KeyGenParameterSpec.Builder("CUSTOM_KEY_ALIAS",
    KeyProperties.PURPOSE_ENCRYPT |
KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)

.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .build());
SecretKey secretKey = keyGenerator.generateKey();
```

The key can then be read using the following code, providing its alias:

```
        val keyStore = KeyStore.getInstance("AndroidKeyStore")
        keyStore.load(null)
        val keyStoreKey = keyStore.getKey("CUSTOM_KEY_ALIAS", null) as
SecretKey

        val cipher = Cipher.getInstance("AES/GCM/NoPadding")
        cipher.init(Cipher.ENCRYPT_MODE, keyStoreKey)
```

Then the key recovered from the KeyStore, can be used to encrypt private data, as shown below; keep in mind that the generation of the initialization vector is delegated to Android:

```
       val textToEncrypt = "..."
        val TRANSFORMATION = "AES/GCM/NoPadding"
        val cipher = Cipher.getInstance(TRANSFORMATION)
        cipher.init(Cipher.ENCRYPT_MODE, getSecretKey())
        val iv = cipher.getIV()
        val encrypted =
cipher.doFinal(textToEncrypt.toByteArray(charset("UTF-8")))
```

Once the encrypted data has been received, it can be stored in the private memory areas of the mobile application, using SharedPreferences in MODE_PRIVATE mode, together with the relative IV:

```
val settings = getSharedPreferences(PREFS_NAME,
Context.MODE_PRIVATE)
val editor = settings.edit()
editor.putString("encryptedData", base64encoded_encryptedPayload)
editor.putString("iv", base64encoded_iv)
```

WARNING:

When using SharedPreferences even in MODE_PRIVATE, if a user were to move the application to the external memory, these data would still be readable in clear text.

Furthermore, it must be noted that the use of ExternalStorage for sensitive information is strongly discouraged.

For completeness, it is necessary to consider that KeyGenParameterSpec is available for Android API <= 23, otherwise you will need to generate keys with KeyPairGeneratorSpec.

Enabling access to KeyStore keys may depend on the security status of the device. In particular, a key can only be accessed if the user authenticates with a PIN or fingerprint.

The use of the encryption key is recommended, although it strongly depends on the requirements of the device and there are devices without an unlocking mechanism.

Refer to the setUserAuthenticationRequired and setUserAuthenticationValidityDurationSeconds function, as shown in the following code:

```
    val keyGenerator = KeyGenerator.getInstance(
        KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore"
    )
    keyGenerator.init(
        KeyGenParameterSpec.Builder(
            "CUSTOM_KEY_ALIAS",
            KeyProperties.PURPOSE_ENCRYPT or
KeyProperties.PURPOSE_DECRYPT
        )
            .setBlockModes(KeyProperties.BLOCK_MODE_GCM)

.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
            .setUserAuthenticationRequired(true)
            .setUserAuthenticationValidityDurationSeconds(120)
            .build()
    )
    val secretKey = keyGenerator.generateKey()
```

Obviously, the use of this additional security measure requires the implementation of a PIN sending mechanism to unlock the device with KeyGuardManager.createConfirmDeviceCredentialIntent.

Android Jetpack Security

Within the AndroidX libraries, is offered a set of APIs dedicated to security built as a wrapper around the operating system primitives whose task is to improve and standardize the encryption procedures for files and sensitive data contained in the system SharedPreferences.

The library offers two levels of protection:

- **Strong:** Offers a balance between performance and robust encryption levels. Suitable for consumer and banking applications.
- **Maximum:** Suitable for solutions that require hardware-based keystores and user interaction to obtain encryption keys.

These libraries deal with data encryption "at rest", i.e. when they are stored in non-volatile memory portions of the devices. Encryption takes place using two different types of keys:

- **key-set**: it consists of one or more keys that are used to actually encrypt the data. The key-set is in encrypted itself and is saved within the SharedPreferences.
- **Primary master key**: is the key that is used to encrypt all key-sets, and is stored within the system KeyStore.

The following example shows how an encrypted file can be created and read using the security module of the Jetpack library.

```
String masterKeyAlias =
MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

File file = new File(context.getFilesDir(), "secret_data");
EncryptedFile encryptedFile = EncryptedFile.Builder(
    file,
    context,
    masterKeyAlias,
    EncryptedFile.FileEncryptionScheme.AES256_GCM_HKDF_4KB
).build();

// opening an output stream on an encrypted file
FileOutputStream encryptedOutputStream =
encryptedFile.openFileOutput();


// opening an input stream on an encrypted file
FileInputStream encryptedInputStream = encryptedFile.openFileInput();
```

The following example shows how it is possible to insert data into the system SharedPreferences in an encrypted manner, using the EncryptedSharedPreferences class belonging to the Jetpack libraries. As a result, both the keys and the values saved within the related XML files will be encrypted.

```
String masterKeyAlias =
MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

SharedPreferences sharedPreferences =
EncryptedSharedPreferences.create(
```

```
    "secret_shared_prefs",
    masterKeyAlias,
    context,

EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,

EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
 );

 // SharedPreferences can now be used as usual
 SharedPreferences.Editor editor = sharedPreferences.edit();
```

Finally, it should be noted that the Jetpack libraries offer an additional level of security when generating the MasterKey. Since this key is based on the keyGenParameterSpec pattern, it is possible to configure the parameters passed to the MasterKeys.getOrCreate() method so that the MasterKey requires, for example, a strong biometric authentication to be used.

SQLite

It is available an extension of the open source and multiplatform SQLCipher for the secure storage of sensitive information using encryption best practices.

The implementation of SQLCipher requires the inclusion of additional libraries and the management of the encryption key in a secure manner. The key must not be hardcoded inside the source code or any application files. If an attacker could access the encryption key, it could decrypt the database contents.

The following are the changes to implement in the Kotlin source code in order to use SQLCipher.

1. Importing the appropriate classes from the net.sqlcipher library:

```
import net.sqlcipher.Cursor;
import net.sqlcipher.database.SQLiteDatabase;
import net.sqlcipher.database.SQLiteOpenHelper;
```

2. Loading the native libraries required by SQLCipher:

```
   constructor(context: Context) : super(context, DATABASE_NAME, null,
DATABASE_VERSION) {
       SQLiteDatabase.loadLibs(context)
   }
```

3. Passing the encryption secret as a parameter:

	var db = this.getWritableDatabase(this.secure_key)
References	• https://developer.android.com/training/articles/keystore.html • https://developer.android.com/topic/security/data#kotlin • https://developer.android.com/reference/android/security/keystore/KeyProtection.html • https://medium.com/@ericfu/securely-storing-secrets-in-an-android-application-501f030ae5a3 • https://medium.com/@josiassena/using-the-android-keystore-system-to-store-sensitive-information-3a56175a454b • https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder.html#setUserAuthenticationRequired(boolean) • https://github.com/doridori/Android-Security-Reference/blob/master/framework/keystore.md#user-authenticating-key-use • https://github.com/doridori/Android-Security-Reference/blob/master/framework/keystore.md#more-on-setuserauthenticationrequiredtrue • https://www.zetetic.net/sqlcipher/ • https://developer.android.com/topic/security/data • https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder

3.3.23 AVOID USE OF PRIVATE EMBEDDED DATA

Requirement ID	DS-002
Priority	**Medium**
Description	It is a bad practice to embed sensitive data (like testing credentials, testing environments URLs, etc.) inside the source code of the application because they could be used by an attacker in order to perform further attacks. Analyzing the source code with an automatic scanner or via a code review process can help in identifying this information lowering the risk of leaking potentially sensitive piece of information.
References	• https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/01-Information_Gathering/05-Review_Webpage_Comments_and_Metadata_for_Information_Leakage

3.3.24 SECURE MANAGEMENT OF FILES

Requirement ID	DS-003
Priority	**Medium**
Description	The most common security problem for an Android application is whether the data saved on the device is accessible to other apps. There are three basic ways to save data on a device: • Internal storage: by default, files created by an app in the internal memory are accessible only by the app that created them. • External storage: files created on external memory, such as SD cards, are globally readable and writable. It is not recommend using this type of memory for saving unencrypted sensitive files. In addition, it is advisable to carefully validate the input on files read from external memory before using them or opening them in the application context. • Content providers: they offer a structured archiving mechanism that can be restricted to your own application or exported for access by other applications. In order to not give other applications access to the ContentProvider, it is necessary to add the string android: exported = false in the manifest. Viceversa, setting this attribute to true will allow access to files by other apps.
References	• https://developer.android.com/training/articles/security-tips#StoringData

3.3.25 SECURE IMPLEMENTATION OF AN APPLICATION PIN PAD

Requirement ID	IN-001
Priority	**Medium**
Description	Applications developed for mobile devices may require the use of a PIN input functionality. This technique must be implemented in a secure way in order to avoid sensitive information disclosure. It is worth considering that PIN Pad must also be used if a credit card number is manually typed.

Android	It's recommended to implement an application PIN Pad for features that need it (such as when applications require to enter PAN numbers), rather than using the system keyboard. These pads will require the following characteristics: • The numeric keys in the keypad must always be randomly arranged. • No visual feedback should be provided on key pressing. In such a way, a user will be protected from: • Any "shoulder surfing" attacks • Malware that screenshots the screen • Any malicious software that replaces system keyboards with a key logger. Furthermore, in order to mitigate brute force attacks, it is suggested to limit the number of PIN input attempts.
References	• https://www.kaspersky.it/blog/cloak-and-dagger-attack/13155/ • https://researchcenter.paloaltonetworks.com/2017/09/unit42-android-toast-overlay-attack-cloak-and-dagger-with-no-permissions/

3.3.26 MEMORY WIPING

Requirement ID	IN-002
Priority	**Medium**
Description	Sometimes applications might require users to enter private data, such as PANs or passwords, which could be leaked in case of main memory persistence. It's recommended to wipe the portion of the memory used to store sensitive data as soon as they have been used, to reduce the time duration those values are stored in RAM.
Android	Due to the behavior of the Java Virtual Machine which manages the memory allocation, it is not possible to determine the moment a variable is dismissed in Java. Also, since String objects are immutable, it's recommended the use of char arrays for the management of private data. Char arrays used to store sensitive information must be overwritten with zeros once the data has been used: `Arrays.fill(chars_array, '\u0000')`

	Keep in mind that the JVM could save in RAM multiple copies of a char array, and in case of overwriting, only the last version of this copy could be deleted. Nevertheless, the proposed strategy turns out to be the most appropriate known way of wiping memory portions in Java.
References	• https://developer.android.com/reference/kotlin/javax/crypto/package-summary • https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#PBEEx • https://www.sjoerdlangkemper.nl/2016/05/22/should-passwords-be-cleared-from-memory/ • https://security.stackexchange.com/questions/6753/deleting-a-java-object-securely

3.3.27 INPUT VALIDATION

Requirement ID	IN-003
Priority	**High**
Description	If the application does not perform any validation on user supplied inputs an attacker could send a malicious character sequence to try and exploit a security weakness. In this scenario it is worth noting that the attack surface can be very extensive because there can be injection attempts on the client side (ex. template injection, XSS, etc.), on the database or even on the server side (ex. deserialization injection, XXE, etc.).
Android	It's a good practice to use a centralized validation system for the application, so that, in case of data validation errors, the input will be rejected. Every user input must always be validated before the application uses it. It is recommended to perform the following actions: • Type checking and type casting • Data structure checking • Characters subsets checks • Maximum length controls Be sure that input is validated according to the application requirements by checking the compliance of its value range and meaning.

	If possible, the best approach would be to use a whitelist of allowed values and reject any other inputs. If it's not possible to use this approach, use regular expressions to make sure that only allowed characters can be entered and that its length is within the expected range.
References	• https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html • https://owasp.org/www-community/vulnerabilities/Improper_Data_Validation

3.3.28 USAGE OF PREPARED STATEMENTS

Requirement ID	IN-004
Priority	**High**
Description	The application is vulnerable to client-side SQL Injection when it fails to validate and encode user-supplied input to create queries against the database. When database queries have to be performed using user input data, the correct approach is to use prepared statements. String concatenation mixing input data and SQL code should never be used. Prepared statements allow developers to have a separation between code and data, blocking, therefore, any SQL injection attacks.
Android	Prepared statements allow developers to have a separation between code and data, blocking, therefore, any SQL injection attacks. It's possible to use prepared statements through the correct usage of query method of android.database.sqlite, which allows parametric values to the selection parameter, and it's immune to SQL Injection assuming, obviously, that selection is not dynamically built with trusted and untrusted values. The following example shows the case of correct and secure usage of the query method: `SQLiteDatabase db = mDbHelper.getReadableDatabase();` `String[] projection = {` `  BaseColumns._ID,` `  FeedEntry.COLUMN_NAME_TITLE,` `  FeedEntry.COLUMN_NAME_SUBTITLE` `  };`

	String selection = FeedEntry.COLUMN_NAME_TITLE + " = ?"; String[] selectionArgs = { "My Title" }; String sortOrder = FeedEntry.COLUMN_NAME_SUBTITLE + " DESC"; Cursor cursor = db.query(FeedEntry.TABLE_NAME, // The table to query projection, // The array of columns to return (pass null to get everything) selection, // The column for the WHERE clause selectionArgs, // The value for the WHERE clause null, // do not group rows null, // do not filter by row groups sortOrder // sorting);
References	• https://developer.android.com/training/data-storage/sqlite.html#kotlin • https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html • https://developer.android.com/training/articles/security-tips.html#InputValidation

3.3.29 COMMUNICATION OVER AN ENCRYPTED CHANNEL

Requirement ID	SC-001
Priority	**High**
Description	Sensitive information that is exchanged between client and server must always pass through encrypted channels (e.g. HTTPS). It's important to be aware that the use of HTTPS is always recommended for any connection, as mobile devices frequently connect to unsecure networks, such as public Wi-Fi hotspots, thus exposing themselves to Man-in-the-Middle (MitM) attacks. It's always necessary to verify that server and client negotiate the use of robust ciphers, and that the server supports only TLS protocols, and not SSL.
Android	It's very important to ensure that the app correctly validates the TLS certificate returned by servers in order to determine and block any Man-in-the-Middle attempt.

	The adoption of Certificate Pinning is also recommended (see paragraph 3.3.30). To establish a secure connection, it is required a certificate exchange. Usually, an application uses a set of trusted CAs pre-installed on the operating system. However, this behavior exposes the application to the risk of Man-in-the-Middle attacks in the potential case any of these CAs wrongly issue a fraudulent certificate or if a malicious CA is installed.
References	• https://books.nowsecure.com/secure-mobile-development/en/sensitive-data/fully-validate-ssl-tls.html • https://developer.android.com/training/articles/security-ssl.html

3.3.30 IMPLEMENT SSL CERTIFICATE PINNING

Requirement ID	SC-002
Priority	**High**
Description	It's advisable to use SSL Certificate Pinning techniques in order to ensure secure client-server communications. Through this technique the mobile application can use a whitelist of certificates or expected public keys, so it can compare the remote certificate to the expected ones. The whitelist is usually called "pinset" and is chosen during the development phase to ensure that it's not possible for an attacker to modify the pins in Main-in-the-Middle scenarios. Therefore, certificate pinning mitigates the problem of compromised CAs, malicious CA cases and Main-in-the-Middle scenarios.
Android	Application should implement SSL Pinning protections. DexGuard offers SSL Pinning support by providing specific APIs. It is common practice to perform pinning on public certificates since, in case of frequent certificate changes, it's usual practice to keep the public key static, and this reduces any updating problems in case of certificate renewal. In order to get the PINs from public keys from a given host, the following command can be used on Unix systems: `echo | openssl s_client -connect HOST:443 2>/dev/null | openssl x509 -pubkey -noout | openssl enc -base64 -d | md5sum` The previous command will extract the md5 digest.

Pinning through HttpURLConnection

The example application below creates an HTTPS connection by performing the pinning on the public key present in the leaf certificate.

First, DexGuard PublicKeyTrustManager class is imported, then getConnection method is implemented, returning a HttpURLConnection object, which performs the pinning process independently, since it uses the PublicKeyTrustManager.

PublicKeyTrustManager is a DexGuard class that accepts in its constructor a series of public keys MD5 hashes of the certificates in order to perform pinning.

```
import com.guardsquare.dexguard.runtime.net.PublicKeyTrustManager

object PinnedPubKeyConnection {
    @Throws(Throwable::class)
    fun getConnection(endpoint: String): HttpURLConnection {
        val url = URL(endpoint)
        val urlConnection: HttpURLConnection

        if (url.getProtocol().toLowerCase().equals("https")) {
            val trustManager =
PublicKeyTrustManager(arrayOf("8FCF8FD90E88D6E35BA8CB6D8836A
2BF"))

            val trustManagers = arrayOf<TrustManager>(trustManager)
            val sslContext = SSLContext.getInstance("TLS")
            sslContext.init(null, trustManagers, null)
            urlConnection = url.openConnection() as HttpsURLConnection

urlConnection.setSSLSocketFactory(sslContext.getSocketFactory())
        } else {
            urlConnection = url.openConnection() as HttpURLConnection
        }

        return urlConnection
    }
}
```

Whenever the developer wants to start a connection via HttpURLConnection, he will have to use the getConnection method as in the example shown below.

```
. . .
val url = "https://sec.ure.com";
try
{
  val pinConnection = PinnedPubKeyConnection.getConnection(url)
  InputStreamReader(pinConnection.getInputStream())
```

```
  (Exception)
  ex
  run({
    // Exception with connection or SSL Pin not valid
  })
}
    . . .
```

Pinning with Network Security Configuration

Starting from Android N it is possible to implement certificate pinning through an XML configuration file which will then be referenced within the AndroidManifest. This type of implementation has the peculiarity of being effective both in the network calls generated by the Kotlin code, and by those generated by the WebView.

The following example file shows how it is possible to specify the desired certificates:

File: network_security_config.xml

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">domain.com</domain>
        <pin-set>
            <pin digest="SHA-
256">4hw5tz+scE+TW+mfFWn1dqvfLG+nU7tq1V8=</pin>
            <pin digest="SHA-
256">YLh1dUR9y6KnbQG/uEtLMkBgFF2Fuihg=</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

Then, this file needs to be referenced inside the AndroidManifest.xml with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application
android:networkSecurityConfig="@xml/network_security_config"
                    ... >
        ...
    </application>
</manifest>
```

Pinning with Retrofit

It is possible to implement certificate pinning through the Retrofit library, which being implemented on top of OkHttp requires the configuration of both components.

The following code shows an example of implementing certificate pinning using Retrofit.

```
CertificatePinner certPinner = new CertificatePinner.Builder()
        .add("domain.com",
             "sha256/4hw5tz+scE+…+nU7tq1V8=")
        .build();
OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .certificatePinner(certPinner)
        .build();
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://domain.com")
        .addConverterFactory(GsonConverterFactory.create())
        .client(okHttpClient)
        .build();
```

Pinning in WebViews

DexGuard class used to perform pinning in a WebView scenario is SSLPinningWebViewClient, which requires an array of the certificate public keys belonging to the server he wants to connect to.

Follows a code example of an activity that instantiate a new activity that contains the application WebView. The PIN and the URL to connect are used as arguments for PinningWebViewActivity.createPinningWebViewIntent.

```
class SampleStartActivity:Activity() {
  private val url = "https://sec.ure.com/ "
  private val publicKeyHashes =
arrayOf<String>("8FCF8FD90E88D6E35BA8CB6D8836A2BF")
  protected fun onCreate(savedInstanceState:Bundle) {
    super.onCreate(savedInstanceState)
    …
    val btn = Button(this)
    btn.setText("SSL check before handshake.")
    btn.setOnClickListener(object:View.OnClickListener() {
      fun onClick(v:View) {
        startActivity(PinningWebViewActivity.createPinningWebViewIntent(
          this@SampleStartActivity, url, publicKeyHashes
        ))
      }
    })
  }
}
```

Then, the WebView is initialized with the DexGuard SSLPinningWebViewClient client, which will transparently perform the pinning action.

If the certificate validation is not successful such as in case of a MitM attack, an error message will be displayed.

```
import com.guardsquare.dexguard.runtime.net.SSLPinningWebViewClient
import
com.guardsquare.dexguard.runtime.net.SimpleSSLPinningWebViewClient
import
com.guardsquare.dexguard.runtime.net.WrongSSLCertificateListener
class PinningWebViewActivity: Activity() {
   protected override fun onCreate(savedInstanceState: Bundle) {
      super.onCreate(savedInstanceState)
      val url = getIntent().getStringExtra(EXTRA_URL)
      val keyHashes = getIntent().getStringArrayExtra(EXTRA_HASHES)
      val webView = WebView(this)
      val client = createPinningClient(webView, keyHashes)
      webView.setWebViewClient(client)
      webView.loadUrl(url)
      setContentView(webView)
   }
   private fun createPinningClient(view:WebView,
publicKeyHashes:Array<String>): WebViewClient {
      val webClient = SSLPinningWebViewClient(publicKeyHashes)

webClient.addWrongCertificateListener(object:WrongSSLCertificateListene
r() {
         fun onWrongCertificate() {
            runOnUiThread(object:Runnable {
               public override fun run() {
                  Toast.makeText(getApplicationContext(),
                     "Wrong certificate, page will not load.",
                     Toast.LENGTH_LONG).show()
               }
            })
         }
      })
      return webClient
   }
   companion object {
      val EXTRA_URL = "extraUrl"
      val EXTRA_HASHES = "extraHashes"
      fun createPinningWebViewIntent(context: Context, url:String,
                              keyHashes:Array<String>): Intent {
         return createGenericPinningWebviewIntent(context, url,
keyHashes)
      }
      private fun createGenericPinningWebviewIntent(context:Context,
url:String,
                                          keyHashes:Array<String>):Intent {
         val intent = Intent(context, PinningWebViewActivity::class.java)
         intent.putExtra(EXTRA_URL, url)
```

	intent.putExtra(EXTRA_HASHES, keyHashes) return intent } } }
References	• https://developer.android.com/training/articles/security-ssl.html • https://books.nowsecure.com/secure-mobile-development/en/sensitive-data/fully-validate-ssl-tls.html • https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning • https://medium.com/@appmattus/android-security-ssl-pinning-1db8acb6621e • https://developer.android.com/training/articles/security-config

3.3.31 IPC INTERFACES SECURE MANAGEMENT

Requirement ID	IPC-001
Priority	**Low**
Description	If the application exposes public interfaces accessible from other applications, it's mandatory to verify that it's not possible to maliciously abuse them by third-party applications installed on the same device. A malicious application may perform actions to check if the targeted application does not verify the trustworthiness of the invoker. URL schemes offer a potential attack vector into the application, so it is necessary to validate all URL parameters and discard any malformed URLs. In addition, it is suggested to limit the available actions to those that do not risk the user's data.
Android	If it's not required to let external applications to use the components of our application, it's advisable to define them as non-exportable, by configuring the android:exported attribute to false in the AndroidManifest.xml manifest file. On the contrary, if it's required to expose those components, it's a good practice to protect them by defining "custom" permissions that must be requested by third-party applications to access those components. Specifically, if the mechanisms of IPC (e.g. Intent, Binder, Service or BroadcastReceiver) do not require to be accessible from external applications, it's advisable to configure the android:exported=false attribute

in the relative element of the component inside the manifest, as shown in the following example:

```
<receiver android:name=“my.special.receiver”
        android:exported=false>
[ . . . ]
</receiver>
```

Also, it's advisable to always specify the android:exported value in order to not introduce vulnerabilities because of an unexpected default behavior.

In fact, default services are not exported by default, but if intent filters are added to their declaration, they are automatically exported and available as IPC invocation to third party apps.

In case access from other applications is required, developers must define a custom permission using the android:permission attribute, as shown in the example below:

```
<permission android:name="my.own.mypermission"
          android:label="my_permission"
          android:protectionLevel="signature" />
[ . . . ]
<receiver android:name=“my.special.receiver”
          android:exported=true
          android:permission=“my.own.permission”
          android:protectionLevel=”signature”>
[ . . . ]
</receiver>
```

Keep in mind that setting the parameter android:protectionLevel to signature allows to guarantee a correct access control, as it ensures that the external app which intends to use the service, has been developed by the same entity (publisher with same certificate) that implemented the app that provide it.

The following Kotlin code snippet shows an implementation example of a runtime check on contact reading permissions.

```
if ((ContextCompat.checkSelfPermission(this,
                          Manifest.permission.READ_CONTACTS) !==
PackageManager.PERMISSION_GRANTED))
{
  if (ActivityCompat.shouldShowRequestPermissionRationale(this,

Manifest.permission.READ_CONTACTS))
  {
    // Visualizzazione di una spiegazione all'utente
  }
  else
  {
    ActivityCompat.requestPermissions(this,
```

```
arrayOf<String>(Manifest.permission.READ_CONTACTS),

MY_PERMISSIONS_REQUEST_READ_CONTACTS)
   // MY_PERMISSIONS_REQUEST_READ_CONTACTS è una costante
intera
  }
}
```

A snippet of code for managing the user response is shown below.

```
override fun onRequestPermissionsResult(requestCode:Int,
permissions:Array<String>, grantResults:IntArray) {
       when (requestCode) {
          MY_PERMISSIONS_REQUEST_READ_CONTACTS -> {
             if ((grantResults.size > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED))
             {
                // permission granted
             }
             else
             {
                // permission  not granted
             }
             return
          }
       }
   }
```

Since, as mentioned, Android activities are considered private by default (android:exported=false) unless they contain an <intent-filter>, it's advisable to continue the interaction only if the received intent is valid.

An Intent is an abstract representation of an action to be performed containing the Description of the operation and the data to be passed to the recipient. Android defines a form of messaging for communication between components of the same app, or different apps, via binder calls.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Intent intent = getIntent();
    Bundle extras = intent.getExtras();
    // Check if intents and extras match the whitelist
```

If the component to be protected is a content provider, developers can use two different permissions type: one for reading and another one for writing.

```
<provider android.name="com.example.testapps.test1.MailProvider"
android.authorities="com.example.testapps.test1.mailprovider"
android.readPermission="com.example.testapps.test1.permission.DB_RE
AD"
android.writePermission="com.example.testapps.test1.permission.DB_WR
ITE">
</provider>
</service>
```

Furthermore, another possible solution is to implement an authentication mechanism based on verifying the signature of the calling application against a set of authorized signatures.

The following Kotlin code example implements the caller validation of the received intent by verifying the signature which, however, suffers from the limitation given by the fact that it only works correctly in a ContentProvider.

```
    fun onIntentReceived(context: Context, intent: Intent) {
        val pid = Binder.getCallingPid()
        val uid = Binder.getCallingUid()
        val pm = context.getPackageManager()
        val packages = pm.getPackagesForUid(uid) ?: throw
IntentValidationException("Invalid UID $uid")
        for (packageName in packages) {
            val packageInfo = pm.getPackageInfo(packageName,
GET_SIGNATURES)
            val appStore = pm.getInstallerPackageName(packageName)
            verifySender(packageName, appStore, packageInfo.signatures)
        }
    }
```

The Activity component offers the getCallingPackage method to extract the caller information, but it must be invoked via startActivityForResult. The Service component allows the invocation of Binder.getCallingUid (), but it is necessary to extend the class and implement a custom handler, as shown in the following code example.

```
class TestRawService : Service() {
    internal val mMessenger = Messenger(IncomingHandler())
    override fun onBind(intent: Intent): IBinder {
        return mMessenger.getBinder()
    }

    internal inner class IncomingHandler : Handler() {
        override fun sendMessageAtTime(msg: Message, uptimeMillis: Long):
Boolean {
            // Here the Binder.getCallingUid () works!
            return super.sendMessageAtTime(msg, uptimeMillis)
        }
```

```
        override fun handleMessage(msg: Message) {
            // Do something
        }
    }
}
```

In addition, to mitigate the risk of compromising the integrity of the received message, it is possible to digitally sign its content, as shown by the following sender-side Kotlin code fragment.

```
        val parcel = Parcel.obtain()
        var bundle = intent.extras
        if (bundle == null) {
            bundle = Bundle()
            intent.replaceExtras(bundle)
        }
        assert(!bundle.containsKey("publicKey") &&
!bundle.containsKey("signature"))
        bundle.writeToParcel(parcel, 0)
        val bytes = parcel.marshall()
        parcel.recycle()
        val signature = createSignature(bytes)
        bundle.putByteArray("publicKey", getPublicKey())
        bundle.putByteArray("signature", signature)
```

The receiving side code for verifying the intent received by validating the signature is instead the following.

```
Bundle bundle = intent.getExtras();
assert( bundle != null );
assert( bundle.containsKey("publicKey") &&
bundle.containsKey("signature"));
byte[] signature = bundle.getByteArray("signature");
byte[] publicKey = bundle.getByteArray("publicKey");
bundle.remove("signature");
bundle.remove("publicKey");
Parcel parcel = Parcel.obtain();
bundle.writeToParcel(parcel, 0);
byte[] bytes = parcel.marshall();
parcel.recycle();
verifySignature( bytes, signature, publicKey );
```

Also, in order to mitigate the risk of replay attacks, it is possible to add the current timestamp to the message.

```
Bundle bundle = intent.getExtras();
if( bundle == null ){
bundle = new Bundle();
```

	} bundle.writeToParcel(parcel, 0); byte[] bytes = encrypt(parcel.marshall()); parcel.recycle(); bundle = new Bundle(); bundle.putByteArray("payload", bytes); bundle.putLong("timestamp", SystemClock.elapsedRealtimeNanos()); bundle.putByteArray("publicKey", getPublicKey()); byte[] signature = createSignature(bytes); bundle.putByteArray("signature",signature); intent.replaceExtras(bundle); Below is the code fragment for the validation of the received intent, including the timestamp of the instant of creation. var bundle = intent.extras!! assert(bundle.containsKey("publicKey") && bundle.containsKey("timestamp")) val timestamp = bundle.getByteArray("timestamp") bundle.remove("timestamp") val publicKey = bundle.getByteArray("publicKey") bundle !!. remove "publicKey" verifyAgainstReplay(timestamp, publicKey) assert(bundle.containsKey("payload") && bundle.containsKey("signature")) var payload = bundle.getByteArray("payload") bundle !!. remove "payload" val signature = bundle.getByteArray("signature") bundle.remove("signature") verifySignature(payload, signature, publicKey) payload = decrypt(payload) val parcel = Parcel.obtain() parcel.unmarshall(payload, 0, payload!!.size) bundle = Bundle.CREATOR.createFromParcel(parcel) intent.replaceExtras(bundle)
References	• https://developer.android.com/training/articles/security-tips.html#Permissions • https://developer.android.com/guide/topics/manifest/permission-element.html#plevel • https://books.nowsecure.com/secure-mobile-development/en/android/implement-intents-carefully.html • https://books.nowsecure.com/secure-mobile-development/en/android/check-activities.html • https://books.nowsecure.com/secure-mobile-development/en/android/implement-content-providers-carefully.html • https://books.nowsecure.com/secure-mobile-development/en/android/use-broadcasts-carefully.html

	• https://books.nowsecure.com/secure-mobile-development/en/android/protect-application-services.html

3.3.32 WEBVIEW SECURE SETTINGS

Requirement ID	WV-001
Priority	**High**
Description	Incorrect use of the WebView can expose applications to attacks in the WEB context in the event that the HTML and/or JavaScript code can be manipulated by a malicious user.
Android	It is recommended to not invoke setJavaScriptEnabled method and to not enable JavaScript interfaces with addJavaScriptInterface. In case it is required to enable the execution of JavaScript code within a WebView or to use the JavaScript interfaces through the addJavaScriptInterface method, it is necessary to make sure that the HTML and JavaScript sources interpreted within the WebView themselves come from a reliable source that implements appropriate user input sanitization. In the event that these measures are not properly implemented, it could be possible to inject malicious code into the WebView and that could lead to the arbitrary execution of Java / Kotlin methods on the device of the victim user. The following code example shows how it is possible to improve the security of a WebView by disabling the execution of JavaScript code, prohibiting access to files and disabling plugins within it. `object Util {` `  fun disableRiskySettings(webView: WebView) {` `  webView.settings.javaScriptEnabled = false` `  webView.settings.pluginState = WebSettings.PluginState.OFF` `  webView.settings.allowFileAccess = false` `  }` It is also recommended blocking any HTTP requests to unexpected domains using shouldOverrideUrlLoading and shouldInterceptRequest and using a whitelist of legitimate domains: `class SaferWebViewClient(hostsWhitelsit: String) : WebViewClient() {` `  private val hostsWhitelist: Array<String>` `  private val webResourceResponseFromString: WebResourceResponse`

```
        get() =
getUtf8EncodedWebResourceResponse(StringBufferInputStream("alert('!NO!')"))

    init {
        this.hostsWhitelist = this.hostsWhitelist
    }

    override fun shouldInterceptRequest(view: WebView, url: String):
WebResourceResponse? {
        return if (isValidHost(url)) {
            super.shouldInterceptRequest(view, url)
        } else {
            webResourceResponseFromString
        }
    }

    private fun getUtf8EncodedWebResourceResponse(data: InputStream):
WebResourceResponse {
        return WebResourceResponse("text/css", "UTF-8", data)
    }


    override fun shouldOverrideUrlLoading(view: WebView, url: String):
Boolean {
        return isValidHost(url)
    }

    private fun isValidHost(url: String): Boolean {
        if (!TextUtils.isEmpty(url)) {
            val host = Uri.parse(url).getHost()
            for (whitelistedHost in hostsWhitelist) {
                if (whitelistedHost.equals(host, ignoreCase = true)) {
                    return true
                }
            }
        }
        return false
    }

}
```

Encoding

If it is necessary to enable the execution of JavaScript code within the WebView, it is necessary to manage any variable controlled by the user that will compose the source code displayed within the WebView.

For example, in the event that the user can control a portion of data similar to the following, which will be inserted in the HTML context:

```
“host.it?user_controlled=<script>// JavaScript code</script>”
```

It will be necessary to apply an encoding function so that the representation of the data is modified as follows:

```
“host.it?user_controlled=&lt;script&gt;//codiceJavaScript&lt;/script&gt;”
```

In this way, in the event that this value is inserted within an HTML page, the structure of the DOM will not be changed and the WebView will display the characters encoded in their equivalent (e.g., & lt; in <).

This can be done by applying the TextUtils.htmlEncode() method to the string received in input by the user, as shown below.

```
var customHtml = getFromTemplate(TextUtils.htmlEncode(receivedInput))
```

It should also be noted that this encoding method will be effective if the user input is inserted within an HTML context, while it will be totally ineffective if the insertion takes place in the JavaScript context.

In the event that the insertion takes place in the JavaScript context, an input validation phase must be applied to verify that the value received at the input conforms to the expected format and an escaping dedicated to the context in question must subsequently be applied.

In the event that the HTML and JavaScript code rendered by the WebView comes from a web server, and in the event that this code is dynamically generated on the basis of parameters that can be controlled by a user, it is recommended, similarly to what described above, to carry out the HTML encoding of all the data received in input that will compose the final HTML code.

This encoding can be done through the features offered by the server framework used, or open-source libraries can be used.

Hardening WebView

It is advisable to configure the WebView used in order to expose the minimum set of features necessary for the proper functioning of the same in relation to the business of the application.

The controllable WebSettings and their purpose are listed below.

- setAllowContentAccess: allows the uploading of content through content providers installed on the device
- setAllowFileAccess: allows access to the file system
- setAllowFileAccessFromFileURLs: allows cross-origin access to local resources through the file:// scheme by other files uploaded through the same scheme

	• setAllowUniversalAccessFromFileURLs: allows access to local resources through the file:// scheme from any source • setBlockNetworkLoads: allows access to remote resources • setJavaScriptCanOpenWindowsAutomatically: allows the execution of the window.open() function • setMixedContentMode: it allows the loading of resources on an unencrypted channel by pages served on an unencrypted channel • setSafeBrowsingEnabled: enables the native SafeBrowsing engine Since the default behavior of the above settings is heterogeneous, it is recommended to set these values to false (with the exception of setSafeBrowsingEnabled which must be set to true) and to subsequently enable only the features strictly necessary for the correct operation of the application. This can be done with the following code: `webView.settings.allowContentAccess = false` `webView.settings.allowFileAccess = false` `webView.settings.allowFileAccessFromFileURLs= false` `webView.settings.allowUniversalAccessFromFileURLs= false` `// In case the contents to be displayed are generated or saved locally` `webView.settings.blockNetworkLoads = true` `webView.settings.javaScriptCanOpenWindowsAutomatically= false` `webView.settings.mixedContentMode = false` `webView.settings.safeBrowsingEnabled = true` It is also recommended to make sure that debugging is disabled on the WebViews using the following code: `webView.webContentsDebuggingEnabled = false` Finally, using third-party code such as dynamic JavaScript code within a WebView could allow malicious users to execute arbitrary code. Therefore, it is recommended to be sure to check of all external scripts that are loaded by the WebView.
References	• https://labs.mwrinfosecurity.com/blog/adventures-with-android-webviews/ • https://developer.android.com/training/articles/security-tips.html#WebView • https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringEscapeUtils.html • https://owasp.org/www-project-enterprise-security-api/

	• https://developer.android.com/reference/kotlin/android/webkit/WebSettings

3.3.33 PROTECT AGAINST LOG DISCLOSURE

Requirement ID	ID-001
Priority	**High**
Description	It's advisable not to use log functions in production. In fact, logs can be easily accessed by malicious users, who could then get sensitive information from them. The information considered critical or that could allow access to additional personal data are the following: • Username • Authentication token o password • Application logs or debugging information • Personal and confidential information (e.g., personal data, payment data) • Device identification data (e.g., IMEI, UDID)
Android	Minimize the usage of standard Android features provided by the Log class and check that confidential information is not leaked in log files. Log files, can be accessible from other apps in case of devices with root privileges, and also via USB debug on non-rooted devices using the "adb" tool with a smartphone linked to a PC: `adb logcat` Avoid invoking methods that print the stacktrace in case of an exception. Another way to mitigate the problem of logging sensitive information without affecting the ability to distinguish the identifiers is to use a different representation of the username, such as a secure hash (e.g., SHA-256). This way, even if attackers are able to access the logs, they will not get any useful information on the actual username.

3.3.34 PROTECT AGAINST SCREENSHOT LEAKAGE

Requirement ID	ID-002
Priority	**Low**
Description	Private data could be subject to disclosure in case of screenshots taken by malicious applications or by the operating system itself. Mobile applications should implement countermeasures to prevent screenshots that could expose sensitive data when shown in the task manager.
Android	Activities that display private data should be protected from capturing screenshots. This can be done using the FLAG_SECURE flag, as shown in the following example: `class FlagSecureActivity : Activity() {` `  public override fun onCreate(savedInstanceState: Bundle?) {` `  super.onCreate(savedInstanceState)` `  window.setFlags(WindowManager.LayoutParams.FLAG_SECURE,` `  WindowManager.LayoutParams.FLAG_SECURE` `  )` `  setContentView(R.layout.main)` `  }` `}`
References	• https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE • https://blog.mindedsecurity.com/2021/05/mobile-screenshot-prevention-cheatsheet.html

3.3.35 PROTECT AGAINST CREDENTIAL THEFT

Requirement ID	ID-003
Priority	**Low**
Description	In order to learn how user digits, mobile operating systems use the Auto Correction feature to populate local cache files.

	Private data such as usernames and passwords could be cached in those files and therefore may be accessed by malicious users who have access to the mobile phone.
Android	It is recommended to disable the auto-complete features on text fields that contain sensitive data, in order to prevent the entered data from being saved in the system caches. This can be done in two ways. Via Kotlin code: `edittext.setInputType(InputType.TYPE_TEXT_FLAG_NO_SUGGESTIONS)` Or via XML: `android:inputType="textNoSuggestions|textVisiblePassword"`
References	• https://developer.android.com/reference/android/text/InputType • https://developer.android.com/reference/android/widget/TextView#setInputType(int)

3.3.36 PROTECT AGAINST CLIPBOARD DATA LEAKAGE

Requirement ID	ID-004
Priority	**Low**
Description	In the event that the application uses the system clipboard in an unsecured manner, a potential attacker could, through access to the device or simply through a malicious application without any particular privilege, access all the data entered in the clipboard from the application, including sensitive data and passwords. Furthermore, note that the malicious application would not need root privileges or special environments to be able to access the system clipboard. The mobile operating system clipboard is globally accessible from all applications. It is not necessary to request any permission or request the user to access the data within it. If the clipboard is used to copy private data, it may be accessible to other applications.
Android	It is recommended to disable the copy/paste functionality on text fields that may contain sensitive data, as the system clipboard could be accessible to

any application installed on the device, making it possible to access the sensitive data stored.

Starting with Android SDK 11 it is possible to overwrite the copy/paste events on a text field:

```
class NoMenuEditText : EditText() {
   private val context: Context? = null
   /**
    * This is a replacement method for the base TextView class method of the same name.
    * This method is used in the android.widget.Editor hidden class to determine whether to make the PASTE / REPLACE popup appear when triggered by the text insertion handler. By returning false, we prevent this window from opening.
    * @return false
    */
   internal fun canPaste(): Boolean {
      return false
   }

   /**
    * This is a replacement method for the base TextView class method of the same name.
    * This method is used in the android.widget.Editor hidden class to determine whether to make the PASTE / REPLACE popup appear when triggered by the text insertion handler. By returning false, we prevent this window from opening.
    * @return false
    */
   override fun isSuggestionsEnabled(): Boolean {
      return false
   }
   private fun init() {

this.setCustomSelectionActionModeCallback(ActionModeCallbackInterceptor())
      // disable long clicks on the field.
      this.isLongClickable = false
   }
   /**
    * Prevents the action bar (the top horizontal bar with the copy, cut, paste, etc .. functions) from appearing, by intercepting (and interrupting) the callback that would create it.
    */
   private inner class ActionModeCallbackInterceptor : ActionMode.Callback {
      private val TAG = NoMenuEditText::class.java.simpleName
      override fun onCreateActionMode(mode: ActionMode, menu: Menu): Boolean {
         return false
      }
      override fun onPrepareActionMode(mode: ActionMode, menu: Menu): Boolean {
```

	`` return false } override fun onActionItemClicked(mode: ActionMode, item: MenuItem): Boolean { return false } override fun onDestroyActionMode(mode: ActionMode) {} } } ```
References	• https://android--examples.blogspot.com/2016/10/android-disable-copy-and-paste-in.html • https://developer.android.com/reference/android/view/View • https://developer.android.com/reference/android/view/ActionMode.Callback • https://developer.android.com/reference/android/widget/TextView#isSuggestionsEnabled()

4 CHECKLIST FOR REQUIREMENT ACCEPTANCE

For the acceptance of the requirements, the following criteria are required:

- Have the requirements been clearly explained in chapter 3?
- Have the requirements been numbered and prioritized?
- Does each requirement meet these characteristics?
    - Complete: necessary information must not be left out.
    - Correct: each requirement must accurately describe the functionality to be implemented.
    - Feasible: it must be possible to implement the requirement with the known possibilities and limitations of the system and the environment.
    - Necessary: the requirement must document something that is actually needed by the customer or by an external requirement, an external interface, or a standard.
    - Prioritized: a Priority must be assigned to the requirement in order to indicate the importance of including it in a specific release of the product.
    - Unambiguous: the requirement must be written in a concise, simple manner, in the language of the user's domain, so that anyone who reads the requirement can give a single interpretation and different readers reach the same conclusion.
    - Verifiable: It must be possible to carry out tests for the requirement in order to verify correct implementation.

4.1 CHECKLIST

The following table summarizes the set of security requirements to be implemented in the development of secure software for Android mobile applications:

Category	Check to implement	Requirement Implemented
Authentication	Secure authentication using Keystore and Fingerprint	
	Secure logout management	
	Usage of temporary access tokens	
	Password security requirements	
	PIN Security requirements	
	Verify presence of local authentication	
	Authenticate using Active Directory	
	Protect from User Enumeration	
	Protect from bruteforcing	

Category	Check to implement	Requirement Implemented
	Authentication with biometric factors	
Reverse engineering protection	Code obfuscation	
	Encrypt classes with sensitive code	
	Limit the usage of whitelist –keep*	
	Protection against information disclosure via stacktrace	
	Static resources encryption	
	Static string encryption	
	Prevent tampering	
	Whitebox cryptography	
Security checks at runtime	Anti-Root controls	
	Anti-Debugging controls	
	Anti-Hooking controls	
Sensitive data management	Encryption of personal data	
	Avoid use of private embedded data	
	Secure management of files	
User input management	Secure implementation of an application PIN Pad	
	Memory Wiping	
	Input validation	
	Usage of prepared statements	

Category	Check to implement	Requirement Implemented
Secure communication with the server	Communication over an encrypted channel	
	Implement SSL Certificate Pinning	
IPC mechanisms	IPC interfaces secure management	
Webview management	Webview secure settings	
Countermeasures to information disclosure	Protect against log disclosure	
	Protect against screenshot leakage	
	Protect against credential theft	
	Protect against clipboard data leakage	