Smartphone Secure Development Guidelines

DECEMBER 2016

www.enisa.europa.eu European Union Agency For Network And Information Security

About ENISA

The European Union Agency for Network and Information Security (ENISA) is a centre of network and information security expertise for the EU, its member states, the private sector and Europe's citizens. ENISA works with these groups to develop advice and recommendations on good practice in information security. It assists EU member states in implementing relevant EU legislation and works to improve the resilience of Europe's critical information infrastructure and networks. ENISA seeks to enhance existing expertise in EU member states by supporting the development of cross-border communities committed to improving network and information security throughout the EU. More information about ENISA and its work can be found at www.enisa.europa.eu.

Contact

For contacting the authors please use office@enisa.europa.eu
For media enquires about this paper, please use press@enisa.europa.eu.

Acknowledgements

We would also like to thank the experts who provided useful feedback and input during the creation of this document. Listing them in no particular order:

John Howie,

Volker Schenk,

Dominic Chell,

Loic Falleta,

Richard Landsberg,

Zach Lanier,

Pietro Ferrara,

Ethan Duffell,

David Rogers, and

Mabel Gu.

ISBN: 978-92-9204-201-1, doi: 10.2824/071102

Table of Contents

Introduction 5

1. Identify and protect sensitive data on the mobile device 6
2. Implement user authentication, authorization and session management correctly 10
3. Handle authentication and authorization factors securely on the device 12
4. Ensure sensitive data is protected in transit 14
5. Secure the backend services and the platform server and APIs 16
6. Secure data integration with third party code 17
7. Consent and privacy protection 18
8. Protect paid resources 20
9. Secure software distribution 21
10. Handle runtime code interpretation correctly 22
11. Check device and application integrity 23
12. Protect the application from client side injections 24
13. Ensure correct usage of biometric sensors and secure hardware 26

Introduction

This document is an updated version of the Smartphone Development Guidelines published by ENISA in 2011. New developments in both software and hardware have been translated into new significant threats for the mobile computing environment, highlighting the need for an update of the document.

The sections from the original document were updated and include the following:

1. Identify and protect sensitive data
2. User authentication, authorization and session management
3. Handle authentication and authorization factors securely on the device
4. Ensure sensitive data protection in transit
5. Secure the backend services and the platform server and APIs
6. Secure data integration with third party code
7. Consent and privacy protection
8. Protect paid resources
9. Secure software distribution
10. Handle runtime code interpretation

New sections which have been added in addition to the above mentioned, to cover the new developments in mobile technology are:

11. Device and application integrity
12. Protection from client side injections
13. Correct usage of biometric sensors

The objective of this document is to be technology neutral. Specific references to some technologies are made, where we were able to provide them. The references are not exhaustive, and there are potentially others which we haven't referenced.

This document is written for developers of smartphone applications as a guide for developing secure mobile applications. As such, all other guidelines regarding secure code development (e.g., Software Development Life Cycle) and guidelines for securing servers (e.g., defence in depth) are still valid and should be employed as needed. Nevertheless, mobile applications have some specific properties and functions, which we have tried to identify to help you make a secure mobile application.

Applying the measures, should always be associated with the appropriate risk assessment. It maybe that some measures are not needed in case an application is not using the resources at which the protection is aimed at.

1. Identify and protect sensitive data on the mobile device

Mobile devices due to their portable nature have a higher risk of loss or theft. Mobile applications need to take this into account and need to add the possibility of device loss into their security model. Mobile applications need to protect the data associated with the application. Specifically, sensitive data such as user personal data and business critical data has to be protected.

N	DESCRIPTION
1.	In the design phase, classify data storage according to sensitivity and apply controls accordingly (e.g. passwords, personal data, location, error logs, etc.). Process, store and use data according to its classification. Validate the security of API calls applied to sensitive data.
2.	Store and process sensitive data on the server instead of the client-end device. The relative security of client vs. server-side security also needs to be assessed on a case-by-case basis (see ENISA cloud risk assessment[1] or the OWASP Cloud top 10[2] for decision support). Highly sensitive data (e.g., biometric data, private keys) should not be transported from the component that were initially created.
3.	When storing sensitive data on the device, use a file encryption API provided by the OS or other trusted source. Some platforms (e.g., iOS and Android) provide file encryption API's which use a secret key protected by the device unlock code and deletable on remote wipe. If this is available, it should be used as it increases the security of the encryption without creating extra burden on the end-user. It also makes stored data safer in the case of loss or theft. However, it should be borne in mind that even when protected by the device unlock key, if data is stored on the device, its security is dependent on the security of the device unlock code if remote deletion of the key is for any reason not possible.
4.	Verify that OS level storage encryption is enabled and the device is protected by a PIN or passphrase.
5.	Do not store/cache sensitive data (including keys) unless they are encrypted and if possible stored in a platform supported tamper-proof area.
6.	Consider re-evaluating access authorization to sensitive data based on contextual information such as location (e.g., require further authentication if location data shows device is outside of expected region).
7.	Do not store historical location data or other sensitive information on the device beyond the period required by the application. Assume that shared storage is untrusted - information may easily leak in unexpected ways through any shared storage. In particular:

[1] Cloud Computing: Benefits, Risks and Recommendations for information security 2009 http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment.
[2] OWASP Cloud Top 10 https://www.owasp.org/index.php/Category:OWASP_Cloud_%E2%80%90_10_Project

	• Be aware of caches and temporary storage as a possible leakage channel, when shared with other apps. • Be aware of shared storage such as address book, media gallery, audio files, as a possible leakage channel. For example, storing images with location metadata in the media-gallery allows that information to be shared in unintended ways. • Do not store temporary cached data in a world readable directory
8.	For sensitive personal data, deletion should be scheduled according to a maximum retention period, (to prevent e.g. data remaining in caches indefinitely).
9.	There is currently no standard secure deletion procedure for flash memory (unless wiping the entire medium/card). Therefore, data encryption and secure key management are especially important.
10.	Consider the security of the whole data lifecycle in writing your application (collection over the wire, temporary storage, caching, backup, deletion, etc.).
11.	Ensure that during application removal (uninstall operation), any confidential user data and the corresponding app-specific credentials are deleted from the execution environment, the device, and any other storage medium.
12.	Apply the principle of minimal disclosure - only collect and disclose data which is required for business use of the application. Identify in the design phase what data is needed, its sensitivity and whether it is appropriate to collect, store and use each data type.
13.	Use non-persistent identifiers which are not shared with other apps wherever possible (e.g., do not use the device unique hardware identifiers such as IMEI or UDID as an identifier).
14.	Applications on managed devices should leverage remote wipe and kill switch APIs[3] to remove sensitive information from the device in the event of theft or loss.
15.	Application developers may want to incorporate an application-specific "data kill switch" into their products, to allow the per-app deletion of their application's sensitive data when needed (strong authentication is required to protect misuse of such a feature).
16.	Do not leak permission-protected data to other applications. This occurs when specific permissions are required to access the data, however an app that has been granted these permissions makes the data available to all other apps without restrictions (e.g., over IPC[4]).
17.	Restrict the data that is shared with other applications (e.g., by implementing an Android Content Provider). This can be accomplished using fine-grained permissions (ensure permissions are protected using signature protection level on Android).

[3] "Kill-switch" is the term used for an OS-level or purpose-built means of remotely removing applications and/or data
[4] Using Interprocess Communication https://developer.android.com/training/articles/security-tips.html#IPC

18.	Restrict broadcast messages (e.g., Android Broadcast Intents) to authorized applications and audit the application's broadcast messages for sensitive content.
19.	Do not allow third party keyboards to be used for inputs that may contain sensitive data (e.g., credentials, credit card information). Prefer a custom keyboard for such inputs instead [5] [6].
20.	Disable Auto Correction and Autosuggestion for inputs that contain sensitive data.
21.	Disable cut, copy and paste functionalities for inputs that may contain sensitive data or restrict the pasteboard to be accessible only from this application.
22.	Disable screen capture for interfaces that contain sensitive data. If the platform does not support this option (e.g., iOS), notify the user about potential security implications of storing a screenshot in unprotected storage.
23.	Disable backgrounding or use a blurry screen when the application transitions to the background in platforms that maintain a screenshot of the visible screen in the local storage (e.g., iOS).
24.	Introduce input field masking for inputs that contain sensitive data (e.g., passwords).
25.	Leverage the hardware-level encryption support for files at the highest supported security level. If possible request application's files to be protected after the device is locked.
26.	If the application while the device is locked needs to write data to a file, use temporary caches instead of weakening the encryption mode. Swap the file content when the device is unlocked and the original file is accessible again.
27.	Prefer using framework functionality (e.g., Android Content Provider) for data sharing instead of using file system permissions or a custom access scheme on platforms that support this (e.g., Android).
28.	Inspect application-initiated custom notification messages for sensitive content: • Allow the user to disable notifications • Allow the user to disable showing content in notifications
29.	Exclude sensitive application files from device backups and cloud synchronization services. If this option is not available in the in use platform (e.g., Android), exclude the whole application from device backups.
30.	If the application allows the arbitrary selection of files from the device storage, consider the use of a white-list to restrict access only to the intended (absolute) file paths.

[5] Keyboard or Keylogger?: a security analysis of third-party keyboards on Android http://seclab.skku.edu/wp-content/uploads/2015/07/mka.pdf

[6] The Samsung SwiftKey Vulnerability http://blog.trendmicro.com/trendlabs-security-intelligence/the-samsung-swiftkey-vulnerability-what-you-need-to-know-and-how-to-protect-yourself/

31.	Delete application caches on app termination.
32.	Database files that contain sensitive data (e.g., iOS WebView caches) must be manually removed from the file system. Deleting records using the database API will not necessary lead to complete data removal from database structure.
33.	Disable application logging and debug messages in production releases. All exceptions should be handled securely.
34.	In the case that the application includes embedded web browsing capabilities (e.g., WebViews), clear stored cookies on app termination or use in-memory cookie storage.

2. Implement user authentication, authorization and session management correctly

Mobile devices are often shared temporarily, lost or stolen. Mobile applications can be undermined by an insecure authentication or authorization control. Unauthorized individuals may obtain access to sensitive data or sensitive systems by circumventing authentication (logins) or by reusing valid tokens or cookies. Mobile applications must implement secure session management to prevent unauthorized access to the application and it's data.

N	DESCRIPTION
1.	Do not rely on client side security controls. Application controls can be easily tampered by an adversary. Both authentication and authorization controls should be implemented on the server side.
2.	Consider using asymmetric cryptography for authentication and authorization purposes. Generate and use the private key directly within a platform supported secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)).
3.	If a password based authentication mechanism is used, ensure that a strong a password policy is being followed. Consider enforcing restrictions about password length and formation, reuse of old user passwords, use of common passwords, password duration, etc. It may also be useful to provide feedback on the strength of the password when it is being entered for the first time. However, do not maintain any representation of the password strength in application storage or the back-end server as it may expose the password in preimage attacks.
4.	Do not reveal registered usernames and remove any fingerprint of their existence from verbose error messages.
5.	Introduce a bruteforce protection mechanism for the authentication controls (e.g., password change/reset). Consider enforcing account lockout for a specific duration, extended questions about the user, notifying the user through another channel and completely automated public Turing tests (captcha) in case of multiple failed attempts.
6.	Ensure that the session management is handled securely[7] after the initial authentication, using appropriate secure protocols.
7.	Require authentication credentials or tokens to be passed with any subsequent request (especially those granting privileged access or modification).
8.	Use unpredictable session identifiers with high entropy.

[7] Secure Session Management: Preventing Security Voids in Web Applications https://www.sans.org/reading-room/whitepapers/webservers/secure-session-management-preventing-security-voids-web-applications-1594

	Note that random number generators generally produce random but predictable output for a given seed (e.g., the same sequence of random numbers is produced for each seed). Therefore, it is important to provide an unpredictable seed for the random number generator. The standard method of using the date and time is not secure. It can be improved, for example using a combination of the date and time, the phone temperature sensor, and the data from the gyroscope sensor (x, y, and z axis). Combining multiple values and using well-tested algorithms that maximise entropy should be chosen.
9.	Use context to add security to authentication (e.g., geo location, IP location, etc). Ensure that any collected data is in compliance with the local laws and regulatory requirements.
10.	Consider using additional authentication factors for applications giving access to sensitive data or interfaces where possible: • Knowledge factors - Something you know (i.e. user's secret question) • Possession factors - Something you have (i.e. hardware token, grid, sim card) • Inherence factors - Something you are (i.e. fingerprint, voice, facial, retina recognition)
11.	Do only rely on not adequately secure channels for multi factor authentication (phone numbers and voice mails can be hijacked, see Section 4-10)
12.	Use authentication that ties back to the end user identity (rather than only to the device identity).
13.	Authentication should not be used as a replacement of authorization security controls. Authorization verifies the permissions of a user and presupposes strong authentication.
14.	Apps that support user authentication must have a logout function which terminates the authenticated session. Upon logout, session should also be invalidated on the server side.
15.	Clear any maintained sensitive data on session termination. Reset the application state and request for user re-authentication.
16.	Clear any maintained sensitive data and attempt to also terminate any server side session after application state change (e.g., termination, backgrounding). Consider a user request for application termination as a request to logout.
17.	For applications that contain sensitive data, is also recommended to request for user re-authentication when the application state changes to background or verify that the device is secured with PIN, pattern or password.
18.	For platforms that support application component history stack (e.g., Android), always clear the stack on session or app termination and user's request to logout.
19.	Ensure that the app runs with user privileges (unprivileged) on the end user device (does not require a rooted or a jailbroken device). Verify that it does not request more access authorizations to system resources and rights in the execution environment than the absolutely necessary (least privilege principle)

3. Handle authentication and authorization factors securely on the device

User account credentials, if stolen, not only provide unauthorized access to the mobile backend service but potentially to other services and accounts owned by the user. Mobile applications need to designed to protect user credentials to protect the users as well as the application's backend infrastructure.

N	DESCRIPTION
1.	Instead of passwords consider using longer term authorization tokens that can be securely stored on the device (as per the OAuth model[8]). Secure the tokens in transit (using TLS). Tokens can be issued by the backend service after verifying the user credentials initially. The tokens should be time bounded to the specific service as well as revocable (if possible server side), thereby minimizing the damage in loss scenarios. Use the latest versions of the authorization standards (such as OAuth 2.0). Make sure that these tokens expire as frequently as practicable.
2.	In the case passwords need to be stored on the device, leverage the encryption and key-store mechanisms provided by the mobile OS to securely store passwords, password equivalents and authorization tokens. Never store passwords in clear text. Do not store passwords or long term session IDs without appropriate encryption.
3.	Leverage the provided key-store mechanisms at the highest supported security level and only when a device passcode has been set. If possible request key-store items to be protected after the device is locked and to remain only in the current device (e.g., exclude these items from backups and cloud synchronization).
4.	Consider purging credentials or keys from memory after use. Avoid automatic memory managed structures (e.g., controlled by garbage collector) and immutable objects for maintaining the keys. Prefer to immediately zero out the memory containing the data after use rather than waiting for the garbage collection mechanism[9].
5.	If the credentials or keys appear in the user interface (UI) components, try to release the UI frames immediately after use.
6.	Some devices and add-ons allow developers to use a secure hardware (e.g., TEE, SE)[10] [11] - the number of devices offering this functionality is likely to increase. Developers should make use of such capabilities to store keys, credentials and other sensitive data.
7.	Provide the ability to the mobile user to change passwords or other authentication tokens.

[8] The OAuth 2.0 Authorization Framework https://tools.ietf.org/html/rfc6749
[9] Mutability http://www.oracle.com/technetwork/java/seccodeguide-139067.html#6
[10] Hardware-Backed Keystore https://source.android.com/security/keystore/
[11] iOS Secure Enclave https://www.apple.com/business/docs/iOS_Security_Guide.pdf

8.	Ensure passwords and keys are not visible in cache or logs.
9.	Do not store any passwords or secrets in the application binary. Do not use a generic shared secret for integration with the backend server (like password embedded in code). Mobile application binaries can be easily downloaded and reverse engineered.

4. Ensure sensitive data is protected in transit

Network-based attacks are one of the major threats to smartphone applications, especially since the majority of smartphones contain multiple different networking technologies. Today most smartphones contain at least WiFi and cellular networking technologies such as GPRS, UMTS, CDMA, LTE (and possible others). In addition, Bluetooth and other short distance radio interfaces such as Near Field Communication (NFC) are commonly integrated in modern smartphones. Sensitive data passing through this shared channels can be intercepted and modified[12].

N	DESCRIPTION
1.	Assume that the network layer is not secure. Specifically, WiFi networks must be considered not trustworthy. Modern network layer attacks can defeat network layer encryption.
2.	Applications should enforce the use of an end-to-end secure channel (such as TLS) when sending sensitive information over any network (e.g., using Strict Transport Security - STS[13]). This includes passing user credentials and other authentication equivalents.
3.	For sensitive data, to reduce the risk of man-in-middle attacks (like SSL proxy, SSL strip), a secure connection should only be established after verifying the identity of the remote-end-point (server). This can be achieved by ensuring that TLS is only established with end-points having the trusted certificates in the key chain.
4.	Leverage the platform specific support for enforcing additional security requirements for HTTP-based networking requests (e.g., ATS in iOS[14] and clear text traffic opt-out in Android[15]).
5.	Use strong and standardized encryption algorithms (e.g., AES) and appropriate key lengths (check recommendations for the algorithm you use e.g. for the TLS configuration). Remove support for weak ciphers.
6.	Enforce secure TLS versions. Safely abort the connection, if this is not possible[16].
7.	Use certificates signed by trusted CA providers. Do not allow self-signed certificates and do not disable or ignore certificate chain validation.
8.	Introduce certificate pinning.

[12] SSLSNIFF http://blog.thoughtcrime.org/sslsniff-anniversary-edition
[13] HTTP Strict Transport Security (HSTS) https://tools.ietf.org/html/rfc6797
[14] IOS https://developer.apple.com/library/ios/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW33
[15] Android https://developer.android.com/training/articles/security-config.html
[16] Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf

	Restrict an application's trusted certificates to a small set of known certificates that are used by the backend servers[17].
9.	Design the user interface in a way that warns the user if the peer certificate does not match the expected certificate and provide the ability to abort any further interaction.
10.	SMS and MMS should not be used to send sensitive data (e.g., two-factor authentication tokens) to or from mobile end-points as SMS and MMS can be intercepted[18][19].
11.	Always use platform supported or vetted frameworks for establishing secure communication channels. Avoid using custom solutions.
12.	Ensure adequate logs on the server are retained about established connections. In the case of multiple intermediate proxies, make sure that HTTP headers are parsed correctly (e.g., X-Forwarded-For).
13.	In the case of rooted or jailbroken devices, consider to integrate a custom or third party secure container for the transmission channel, since the platform security controls that establish the TLS connection cannot be trusted

[17] Certificate and Public Key Pinning https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning
[18] SMS-based One-Time Passwords: Attacks and Defense http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2014/tr_2014-02.pdf
[19] Security of Mobile Banking and Payments https://www.sans.org/reading-room/whitepapers/ecommerce/security-mobile-banking-payments-34062

5. Secure the backend services and the platform server and APIs

The majority of mobile applications interact with a backend using web services or proprietary protocols. Insecure implementation of backend APIs, services, and not keeping the back-end platform hardened/patched will allow attackers to compromise data on the mobile device when transferred to the back-end, or to attack the backend through the mobile application[20]. In this section we try to only provide specific measures to secure mobile application backends (appropriate literature for securing servers and web services exists and the reader should refer to those)

N	DESCRIPTION
1.	Carry out a specific check of your code for sensitive data unintentionally transferred between the mobile device and web-server back-ends and other external interfaces - (e.g., is location or other information transferred within file metadata).
2.	All back-end services (web services) for mobile apps should be tested for vulnerabilities periodically, e.g., using static code analyser tools and fuzzing tools for testing and finding security flaws. Perform abuse case testing, in addition to use case testing.
3.	Disable metadata publishing (e.g., metadata for WSDL documents and for WSDL derived objects), in order to prevent unintentional disclosure of potentially sensitive service metadata[21].
4.	Ensure that the back-end platform (server) is running with a hardened configuration with the latest security patches applied to the OS, web server and other application components.
5.	Ensure adequate logs are retained on the back-end in order to detect and respond to incidents and perform forensics (within the limits of data protection law).
6.	Protect the back-end from client initiated log injections that may corrupt or forge the history of events[22].
7.	Employ rate limiting and throttling on a per-user/IP basis (if user identification is available) to reduce the risk from denial of service (DoS) attack.
8.	Test for DoS vulnerabilities where the server may become overwhelmed by certain resource intensive application calls

[20] OWASP Web Services https://www.owasp.org/index.php/Web_Services
[21] Default Metadata Publishing Behavior in .NET Framework https://msdn.microsoft.com/en-us/library/ms751498(v=vs.110).aspx
[22] OWASP Log Injection https://www.owasp.org/index.php/Log_Injection

6. Secure data integration with third party code

Third party code can represent both a security and a privacy liability. Third party code can use the application's access to user data and leak it on purpose or by accident. Similarly, third party code can introduce security vulnerabilities into an otherwise secure application. Application developers have to invest a minimum of time to vet any third party code they include in their application.

N	DESCRIPTION
1.	Vet the security/authenticity of third party code/libraries used in the mobile application. Ensure that third party code is only taken from a reliable source that maintains their code. • Audit code for security issues. • Audit the library and inspect any transmitted data to third-party services for privacy issues (e.g., analytics or ad libraries).
2.	Track third party frameworks/APIs used in the mobile application for security patches. Integrate security updates for third party code/libraries/frameworks/APIs on a regular basis together with your own code. Ask the provider for a security report.
3.	Pay attention to validate all data received from third parties before processing them within your application. This includes local applications, OS services as well as data received over the network.
4.	Avoid using third-party libraries that contain main processor-only cryptographic implementations. Prefer using cryptographic framework provided by a platform supported secure hardware (e.g. TEE, SE).
5.	Software components that are no longer supported by the vendor or developer must not be used

7. Consent and privacy protection

Mobile applications often store and operate on personal information, therefore, they need to be designed to prevent unintentional disclosure of personal or private information. Developers have to pay specific attention to obtain consent to, any data collection, sharing and usage that takes place on the application[23][24].

N	DESCRIPTION
1.	Check whether your application is collecting personal data. It may not always be obvious - for example, do you use persistent unique identifiers linked to central data stores containing personal information?
2.	Create a privacy policy covering the usage of personal data and make it available to the user especially prior to making consent choices.
3.	Prior to using personal data consent should be obtained. When obtaining consent, explicitly notify the user with specific information such as: • what exactly personal data will be used; • what is the purpose of the processing; • who are the recipients of the data; • where is the data stored and for how long. In case that the user does not grant consent to all requested data, he/she should be informed about possible limitations of the app's functionality.
4.	Consent may be collected in 3 main ways: • At install time. • At run-time when data is sent. • Via "opt-in" mechanisms where a user has to explicitly turn on a setting.
5.	It should be possible for the user to withdraw consent at any time in the application. Notify the user how the application behaviour might change in case that consent is withdrawn.
6.	Audit communication mechanisms to check for unintended leaks (e.g., image metadata).
7.	Keep a record of user consent for the processing of different types of personal data.
8.	Check whether data collection (from the user's device) is not excessive with regard to the consent that has been granted by the user (e.g. collecting more types of data than needed - APP-native + WebKit HTML)

[23] PETs controls matrix - A systematic approach for assessing online and mobile privacy tools, ENISA, 2016
[24] ARTICLE 29, DATA PROTECTION WORKING PARTY 00461/13/EN WP 202 http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2013/wp202_en.pdf

9.	Consider taking advantage of built-in features to require access to device sensors and data (e.g., access to gps, camera, etc.). Provide clear explanation on why the access is needed.
10.	Minimize access to sensor data whenever possible (e.g., do not automatically collect geolocation data).
11.	Reduce data granularity and anonymize data on the device instead of remotely. (e.g., strip image metadata).
12.	Require consent prior to providing user data to third parties. Provide clear notice of data shared cross-application with third-parties. Never provide precise location data to third-party applications nor data stored in the secure containers of the application.
13.	Reduce retention period on the mobile or remotely to the minimum amount of time needed to provide the service. Delete data immediately after the retention period has expired. Delete data from all locations (especially remote servers) where data might be stored.
14.	Use privacy enhancing technologies, that support data minimization, anonymization and security of personal data[25]
15.	The default settings of the application should provide maximum privacy and security protection for the user.

[25] See more information and relevant ENISA's work at: https://www.enisa.europa.eu/topics/data-protection/privacy-enhancing-technologies

8. Protect paid resources

Smartphone applications give programmatic access to paid resources on mobile phones such as phone calls, SMS, phone calls and SMS to premium numbers, roaming data, NFC payments, and third party payment systems. Applications that integrate those services must take particular care to prevent abuse. Developers have to consider the financial impact of vulnerabilities in their application. Furthermore, applications that implement In-Application payment for selling services to the user must protect their payment code against abuse.

N	DESCRIPTION
1.	Maintain logs of access to paid-for resources in non-repudiable format (e.g., a signed receipt sent to a trusted server backend - with user consent) and make them available to the end-user for monitoring. Logs should be protected from unauthorized parties.
2.	Check for anomalous usage patterns in paid-for resources usage and trigger re-authentication (e.g., when significant change in location, user-language changes, significant higher paid-for service usage).
3.	Consider a white-list model by default for paid-for resources addressing e.g., address book contacts only unless specifically authorized for phone calls.
4.	Warn user and obtain consent for any cost implications for app behaviour.
5.	Applications have to take into account that different operating system versions provide different levels of access control (e.g., Android permissions) for various system resources. Specifically paid resources have different levels of access control depending on the version of the OS. Applications have to implement access control for those resources to prevent abuse of the application's access to those resources due to missing access control in older/newer versions of the OS and and/or application framework.
6.	Follow the OS/device vendor guidelines for implementing In-App payment: • Implement validation of payment receipts on the backend server not on the device[26]. • Pay specific attention when integrating payment acceptance from a third party wallet (wallet not integrated into the mobile OS).

[26] VirtualSwindle: An Automated Attack Against In-App Billing on Android
http://mulliner.org/collin/publications/asia226-mulliner.pdf

9. Secure software distribution

Overall software security on mobile devices is enforced by code signing and fast security updates. The use of secure practice for software distribution is paramount to the overall security of the application and it is fundamental to mitigate all risks described in these guidelines.

N	DESCRIPTION
1.	Applications must be designed and provisioned to allow updates for security patches, taking into account the requirements for approval by app-stores and the extra delay this may imply.
2.	Official apps stores monitor apps for insecure code and are able to remotely remove apps at short notice in case of an incident. Distributing apps through official app-stores therefore provides a safety-net in case of serious vulnerabilities in your app.
3.	Provide feedback channels for users to report security problems with apps such as a security@ email address[27].
4.	If an enterprise app store is used, protect the application signing key with the utmost care (e.g., use an HSM, air-gapped machine, etc.).
5.	Out-of-appstore security updates should be shipped using an encrypted connection and their content should be verified before applying the update.
6.	Resources used by apps that are updated outside of the app-store normal mechanism must be signed. Apps must verify the signature before accepting the updated resource.
7.	Do not deploy apps with ad-hoc signing certificates used for development and testing.
8.	Do not generate one application for multiple environments. The production app must not contain log calls, developer URLs, test methods, and settings of the development or the testing environment.

[27] Mailbox Names https://www.ietf.org/rfc/rfc2142.txt

10. Handle runtime code interpretation correctly

Runtime interpretation of code and careless treatment of information flow may give an opportunity for untrusted parties to provide unverified input which is interpreted as code or to leak sensitive information. This gives an opportunity for malware to circumvent walled garden controls provided by app-stores. It can lead to injection attacks leading to data leakage, surveillance and, spyware.

Lack of control on the information flow can lead to data leakage in the presence of a physical attacker.

Note that it is not always obvious that your code contains an interpreter. Look for any capabilities accessible via user-input data and use of third party API's which may interpret user-input - such as JavaScript interpreters.

N	DESCRIPTION
1.	Filter user data passed to interpreters.
2.	Define comprehensive escape syntax as appropriate.
3.	Do not reveal sensitive information such as usernames, personal data and others through error messages.
4.	Deny interpreted code direct access to user data and encrypted storage.
5.	Strip unused functionalities from interpreters.
6.	Limit size of input data passed to interpreters.

11. Check device and application integrity

Modified devices and/or applications undermine the security and privacy controls implemented in the mobile application. Device modification can be done through rooting/jailbreaking or by installing a custom OS image. Modified applications cannot be trusted to behave in the way the developer intended it. The same counts for modified devices. Current smartphone platforms support device and/or application integrity checking features those should be leveraged to check the integrity of the device and application.

N	DESCRIPTION
1.	Check the device/platform integrity to ensure that the device is not modified. Prefer using Platform services if available (e.g., Android SafetyNet attestation[28]). Only implement custom or use third party root/jailbreak detection, if platform does not offer a built-in solution.
2.	Check the application integrity, check that the application and its resources are not modified: • Use platform service (e.g., Android SafetyNet attestation, iOS App Store receipt[29]). • Perform in-memory code integrity checks to protect against code modification and/or runtime hooking.
3.	Disable developer features: • Disable debugging in the application settings. • Check if the device is in developer mode if supported by platform (e.g., Android). • Check if debugger is attached and/or if the process is being traced. On platforms with managed code check for managed and native code debuggers.
4.	Make reverse engineering harder: • Obfuscate code. • Encrypt data (e.g., strings) to further obfuscate application logic.

[28] Checking Device Compatibility with SafetyNet https://developer.android.com/training/safetynet/index.html
[29] IOS Receipt Validation
https://developer.apple.com/library/ios/releasenotes/General/ValidateAppStoreReceipt/Introduction.html

12. Protect the application from client side injections

Mobile apps present increased opportunities for client side injections, since they constantly interact with sensors, other installed apps and third party services. Existing mobile application flaws can be exploited in a similar way to vulnerabilities in traditional software applications. Attackers may force the application to use specially crafted data that will modify the application logic flow and lead to access control bypass or information disclosure attacks.

N	DESCRIPTION
1.	In the case that the application includes embedded web browsing capabilities (e.g., WebViews), restrict access to third party domains that do not comply with the required security standards, disable any unused platform supported functionalities, such as the plugins, local file accessibility, local content provider (content URL) accessibility and the dynamic code (e.g., JavaScript) execution support. Furthermore, avoid using full screen web interfaces since these can be abused from attackers to create fake application screens.
2.	Avoid using API calls that provide bridging of dynamic code (e.g., JavaScript) with native code (e.g., Objective-C) since an injection in the dynamic code will lead to native code execution[30].
3.	In the case that the application uses JavaScript code running in the context of a file scheme URL, it is recommended to disable any unused platform supported attributes, such as accessing content from other file scheme URL and content from any origin.
4.	Prevent interaction events when the application is obscured by another interface in the presentation layer in order to mitigate tapjacking[31] attacks. By disabling the application interaction events, the possibility of a user interacting with a hidden view is eliminated.
5.	In the case that the application requests custom permissions, and older platforms are supported (e.g., earlier than Android 5.0), always verify on the first run of the app that no other application has previously requested the same permissions[32].
6.	Always follow the domain name registration infrastructure to declare a custom permission, in order to avoid any collisions with other apps.
7.	Restrict what apps can cause an application component (e.g., Android Activity) to start or are able to interact with it (e.g., Android Service and Content Provider). This can be accomplished using strict permissions.
8.	Restrict the third party applications whose broadcast messages will be accepted by the application.

[30] A View To A Kill: WebView Exploitation https://www.usenix.org/system/files/conference/leet13/leet13-paper_neugschwandtner.pdf
[31] Tapjacking http://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/
[32] The Custom Permission Problem https://github.com/commonsguy/cwac-security/blob/master/PERMS.md

9.	In the case that the application utilizes a platform provided download manager, always verify that the received manager's notifications are related to application's initiated downloads.
10.	Always verify dynamic code downloads and application updates at the client side. Any resource that is being retrieved from an external service (e.g., compressed files, APK files) should be validated for its integrity and its signing certificate.
11.	Always validate server responses when using backend APIs. Introduce a whitelist model for accepted responses.
12.	Mitigate SQL injections, local file inclusion, JavaScript injections, XML injections. When dealing with dynamic queries (e.g., SQL queries with untrusted inputs) or Content-Providers ensure you are using parameterized queries. Always validate user provided inputs that will be used for file accessing purposes or as part of a dynamic code execution. Use a vetted framework for XML operations.
13.	Protect from memory corruptions in applications that are developed using a programming language which supports explicit memory management (e.g., Objective-C, C, C++). Perform static analysis for memory management vulnerabilities in the development process.
14.	Do not use insecure cached data in HTTP connections and in embedded web browsing capabilities (e.g., WebViews). Caches are usually located on the device file system. Many platforms allow applications to place this cached data to insecure locations (e.g., sdcard on Android) in which they can be easily tampered.
15.	In platforms that support custom applications with accessibility permissions (e.g., Android), exclude sensitive user interface elements from being accessed by accessibility applications.
16.	Avoid populating webviews loaded from the file URI scheme with user supplied DOM input.

13. Ensure correct usage of biometric sensors and secure hardware

Biometric sensors make authentication systems both easier and faster to use, however the authentication and accessibility policies must be enforced by the secure hardware in order to be protected against anything up to and including kernel compromise.

N	DESCRIPTION
1.	Always verify that there is a biometric sensor (e.g., Fingerprint reader) present and available on the device before using the API for authentication purposes. In the case that the sensor is not available, an alternative authentication control should be provided.
2.	Always verify that the biometric sensor/secure hardware authentication policy of the in-use platform complies with the application's authentication policy (passcode required after cold boot, biometric sensor authentication expiration, adding a fingerprint requires pin/passcode/biometric authentication, requirement for biometric sensor being individually paired with secure hardware).
3.	Ensure that there are enrolled data using the biometric sensor (e.g., user's fingerprints and/or user's iris are registered) before using the API for authentication purposes.
4.	Ensure that the enrolled biometric data has not been changed since the activation of the authentication control using the biometric sensor (e.g., another user added a new fingerprint/iris sample).
5.	The application should not use the biometric sensor for just verifying user presence (e.g., iOS LocalAuthentication). This control can be easily circumvented using dynamic hooking/static patching. Instead, the application should use the biometric sensor to access keys stored using a hardware backed keystore/keychain and protected with keychain access control lists (ACL).
6.	Ensure that the key material is bound to the secure hardware (e.g., TEE, SE) in platforms that this is optional (e.g., Android). When this feature is enabled for a key, its key material is never exposed outside of secure hardware.
7.	For keys whose key material is inside a secure hardware (e.g., TEE, SE), ensure that cryptographic and user authentication authorizations are also enforced by secure hardware, in platforms that this is optional (e.g., Android). Authentication control (e.g., using Biometric checks) and key decryption should be performed atomically in a TEE or on a chip with a secure channel to the TEE.
8.	The application should avoid using temporal validity interval authorizations, since they are unlikely to be enforced by the secure hardware because it normally does not have an independent secure real-time clock.
9.	Verify that the application's authentication policy complies with the possibility that different people may enroll for biometric authentication in the same device. As a result, successful biometric authentication may be possible for different device users

ENISA

European Union Agency for Network
and Information Security
Science and Technology Park of Crete (ITE)
Vassilika Vouton, 700 13, Heraklion, Greece

Athens Office

1 Vasilissis Sofias
Marousi 151 24, Attiki, Greece

TP-07-16-145-EN-N

PO Box 1309, 710 01 Heraklion, Greece
Tel: +30 28 14 40 9710
info@enisa.europa.eu
www.enisa.europa.eu

ISBN: 978-92-9204-201-1
DOI: 10.2824/071102