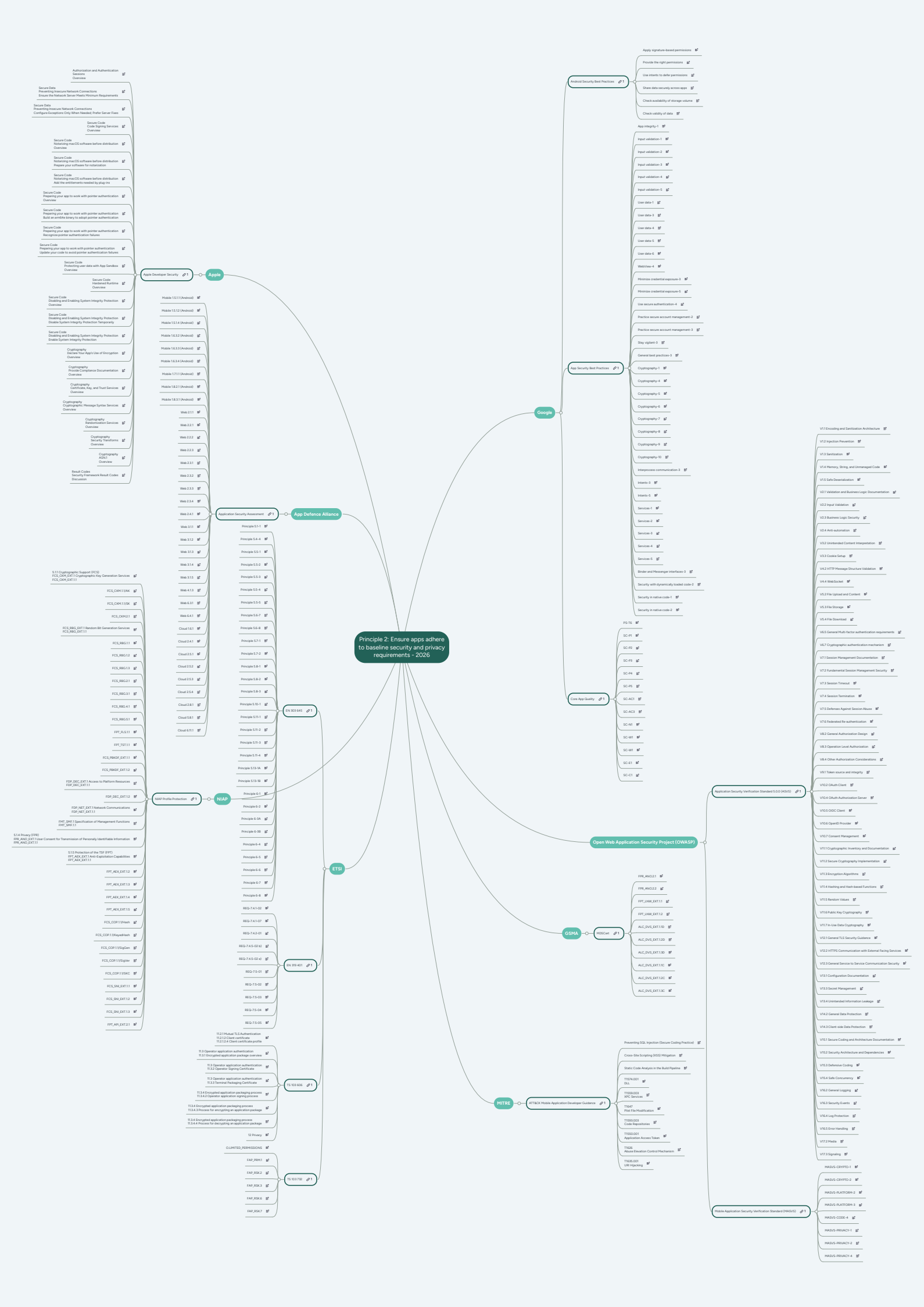
Principle 2: Ensure apps adhere to baseline security and privacy requirements - 2026
Google
Android Security Best Practices 1
Apply signature-based permissions
Provide the right permissions
Use intents to defer permissions
Share data securely across apps
Check availability of storage volume
Check validity of data
App Security Best Practices 1
App integrity-1
Input validation-1
Input validation-2
Input validation-3
Input validation-4
Input validation-5
User data-1
User data-3
User data-4
User data-5
User data-6
WebView-4
Minimize credential exposure-3
Minimize credential exposure-5
Use secure authentication-4
Practice secure account management-2
Practice secure account management-3
Stay vigilant-3
General best practices-3
Cryptography-1
Cryptography-4
Cryptography-5
Cryptography-6
Cryptography-7
Cryptography-8
Cryptography-9
Cryptography-10
Interprocess communication-3
Intents-3
Intents-5
Services-1
Services-2
Services-3
Services-4
Services-5
Binder and Messenger interfaces-3
Security with dynamically loaded code-2
Security in native code-1
Security in native code-2
Core App Quality 1
PS-T6
SC-P1
SC-P2
SC-P3
SC-P4
SC-P5
SC-AC1
SC-AC3
SC-N1
SC-W1
SC-W1
SC-E1
SC-C1
Apple
Apple Developer Security 1
Authorization and Authentication Sessions Overview
Secure Data Preventing Insecure Network Connections Ensure the Network Server Meets Minimum Requirements
Secure Data Preventing Insecure Network Connections Configure Exceptions Only When Needed; Prefer Server Fixes
Secure Code Code Signing Services Overview
Secure Code Notarizing macOS software before distribution Overview
Secure Code Notarizing macOS software before distribution Prepare your software for notarization
Secure Code Notarizing macOS software before distribution Add the entitlements needed by plug-ins
Secure Code Preparing your app to work with pointer authentication Overview
Secure Code Preparing your app to work with pointer authentication Build an arm64e binary to adopt pointer authentication
Secure Code Preparing your app to work with pointer authentication Recognize pointer authentication failures
Secure Code Preparing your app to work with pointer authentication Update your code to avoid pointer authentication failures
Secure Code Protecting user data with App Sandbox Overview
Secure Code Hardened Runtime Overview
Secure Code Disabling and Enabling System Integrity Protection Overview
Secure Code Disabling and Enabling System Integrity Protection Disable System Integrity Protection Temporarily
Secure Code Disabling and Enabling System Integrity Protection Enable System Integrity Protection
Cryptography Declare Your App's Use of Encryption Overview
Cryptography Provide Compliance Documentation Overview
Cryptography Certificate, Key, and Trust Services Overview
Cryptography Cryptographic Message Syntax Services Overview
Cryptography Randomization Services Overview
Cryptography Security Transforms Overview
Cryptography ASN.1 Overview
Result Codes Security Framework Result Codes Discussion
App Defence Alliance
Application Security Assessment 1
Mobile 1.5.1.1 (Android)
Mobile 1.5.1.2 (Android)
Mobile 1.5.1.4 (Android)
Mobile 1.6.3.2 (Android)
Mobile 1.6.3.3 (Android)
Mobile 1.6.3.4 (Android)
Mobile 1.7.1.1 (Android)
Mobile 1.8.2.1 (Android)
Mobile 1.8.3.1 (Android)
Web 2.1.1
Web 2.2.1
Web 2.2.2
Web 2.2.3
Web 2.3.1
Web 2.3.2
Web 2.3.3
Web 2.3.4
Web 2.4.1
Web 3.1.1
Web 3.1.2
Web 3.1.3
Web 3.1.4
Web 3.1.5
Web 4.1.3
Web 6.3.1
Web 6.4.1
Cloud 1.6.1
Cloud 2.4.1
Cloud 2.5.1
Cloud 2.5.2
Cloud 2.5.3
Cloud 2.5.4
Cloud 2.8.1
Cloud 5.8.1
Cloud 6.11.1
NIAP
NIAP Profile Protection 1
5.1.1 Cryptographic Support (FCS) FCS_CKM_EXT.1 Cryptographic Key Generation Services FCS_CKM_EXT.1.1
FCS_CKM.1.1/AK
FCS_CKM.1.1/SK
FCS_CKM.2.1
FCS_RBG_EXT.1 Random Bit Generation Services FCS_RBG_EXT.1.1
FCS_RBG.1.1
FCS_RBG.1.2
FCS_RBG.1.3
FCS_RBG.2.1
FCS_RBG.3.1
FCS_RBG.4.1
FCS_RBG.5.1
FPT_FLS.1.1
FPT_TST.1.1
FCS_PBKDF_EXT.1.1
FCS_PBKDF_EXT.1.2
FDP_DEC_EXT.1 Access to Platform Resources FDP_DEC_EXT.1.1
FDP_DEC_EXT.1.2
FDP_NET_EXT.1 Network Communications FDP_NET_EXT.1.1
FMT_SMF.1 Specification of Management Functions FMT_SMF.1.1
5.1.4 Privacy (FPR) FPR_ANO_EXT.1 User Consent for Transmission of Personally Identifiable Information FPR_ANO_EXT.1.1
5.1.5 Protection of the TSF (FPT) FPT_AEX_EXT.1 Anti-Exploitation Capabilities FPT_AEX_EXT.1.1
FPT_AEX_EXT.1.2
FPT_AEX_EXT.1.3
FPT_AEX_EXT.1.4
FPT_AEX_EXT.1.5
FCS_COP.1.1/Hash
FCS_COP.1.1/KeyedHash
FCS_COP.1.1/SigGen
FCS_COP.1.1/SigVer
FCS_COP.1.1/SKC
FCS_SNI_EXT.1.1
FCS_SNI_EXT.1.2
FCS_SNI_EXT.1.3
FPT_API_EXT.2.1
ETSI
EN 303 645 1
Principle 5.1-1
Principle 5.4-4
Principle 5.5-1
Principle 5.5-2
Principle 5.5-3
Principle 5.5-4
Principle 5.5-5
Principle 5.6-7
Principle 5.6-8
Principle 5.7-1
Principle 5.7-2
Principle 5.8-1
Principle 5.8-2
Principle 5.8-3
Principle 5.10-1
Principle 5.11-1
Principle 5.11-2
Principle 5.11-3
Principle 5.11-4
Principle 5.13-1A
Principle 5.13-1B
Principle 6-1
Principle 6-2
Principle 6-3A
Principle 6-3B
Principle 6-4
Principle 6-5
Principle 6-6
Principle 6-7
Principle 6-8
EN 319 401 1
REQ-7.4.1-02
REQ-7.4.1-07
REQ-7.4.2-01
REQ-7.4.5-02 b)
REQ-7.4.5-02 e)
REQ-7.5-01
REQ-7.5-02
REQ-7.5-03
REQ-7.5-04
REQ-7.5-05
TS 103 606 1
11.2.1 Mutual TLS Authentication 11.2.1.2 Client certificate 11.2.1.2.4 Client certificate profile
11.3 Operator application authentication 11.3.1 Encrypted application package overview
11.3 Operator application authentication 11.3.2 Operator Signing Certificate
11.3 Operator application authentication 11.3.3 Terminal Packaging Certificate
11.3.4 Encrypted application packaging process 11.3.4.2 Operator application signing process
11.3.4 Encrypted application packaging process 11.3.4.3 Process for encrypting an application package
11.3.4 Encrypted application packaging process 11.3.4.4 Process for decrypting an application package
12 Privacy
TS 103 732 1
CLIMITED_PERMISSIONS
FAP_PRM.1
FAP_RSK.2
FAP_RSK.3
FAP_RSK.6
FAP_RSK.7
GSMA
MOSCart 1
FPR_ANO.2.1
FPR_ANO.2.2
FPT_LNK_EXT.1.1
FPT_LNK_EXT.1.2
ALC_DVS_EXT.1.1D
ALC_DVS_EXT.1.2D
ALC_DVS_EXT.1.3D
ALC_DVS_EXT.1.1C
ALC_DVS_EXT.1.2C
ALC_DVS_EXT.1.3C
MITRE
ATT&CK Mobile Application Developer Guidance 1
Preventing SQL Injection (Secure Coding Practice)
Cross-Site Scripting (XSS) Mitigation
Static Code Analysis in the Build Pipeline
T1574.001 DLL
T1059.003 XPC Services
T1647 Plist File Modification
T1593.003 Code Repositories
T1550.001 Application Access Token
T1626 Abuse Elevation Control Mechanism
T1635.001 URI Hijacking
Open Web Application Security Project (OWASP)
Application Security Verification Standard 5.0.0 (ASVS) 1
V1.1 Encoding and Sanitization Architecture
V1.2 Injection Prevention
V1.3 Sanitization
V1.4 Memory, String, and Unmanaged Code
V1.5 Safe Deserialization
V2.1 Validation and Business Logic Documentation
V2.2 Input Validation
V2.3 Business Logic Security
V2.4 Anti-automation
V3.2 Unintended Content Interpretation
V3.3 Cookie Setup
V4.2 HTTP Message Structure Validation
V4.4 WebSocket
V5.2 File Upload and Content
V5.3 File Storage
V5.4 File Download
V6.5 General Multi-factor authentication requirements
V6.7 Cryptographic authentication mechanism
V7.1 Session Management Documentation
V7.2 Fundamental Session Management Security
V7.3 Session Timeout
V7.4 Session Termination
V7.5 Defenses Against Session Abuse
V7.6 Federated Re-authentication
V8.2 General Authorization Design
V8.3 Operation Level Authorization
V8.4 Other Authorization Considerations
V9.1 Token source and integrity
V10.2 OAuth Client
V10.4 OAuth Authorization Server
V10.5 OIDC Client
V10.6 OpenID Provider
V10.7 Consent Management
V11.1 Cryptographic Inventory and Documentation
V11.2 Secure Cryptography Implementation
V11.3 Encryption Algorithms
V11.4 Hashing and Hash-based Functions
V11.5 Random Values
V11.6 Public Key Cryptography
V11.7 In-Use Data Cryptography
V12.1 General TLS Security Guidance
V12.2 HTTPS Communication with External Facing Services
V12.3 General Service to Service Communication Security
V13.1 Configuration Documentation
V13.3 Secret Management
V13.4 Unintended Information Leakage
V14.2 General Data Protection
V14.3 Client-side Data Protection
V15.1 Secure Coding and Architecture Documentation
V15.2 Security Architecture and Dependencies
V15.3 Defensive Coding
V15.4 Safe Concurrency
V16.2 General Logging
V16.3 Security Events
V16.4 Log Protection
V16.5 Error Handling
V17.2 Media
V17.3 Signaling
Mobile Application Security Verification Standard (MASVS) 1
MASVS-CRYPTO-1
MASVS-CRYPTO-2
MASVS-PLATFORM-2
MASVS-PLATFORM-3
MASVS-CODE-4
MASVS-PRIVACY-1
MASVS-PRIVACY-2
MASVS-PRIVACY-4

Principle 2: Ensure apps adhere to baseline security and privacy requirements - 2026

1. Google

1.1. Android Security Best Practices

Link: https://developer.android.com/privacy-and-security/security-best-practices

Link: https://developer.android.com/privacy-and-security/security-best-practices

1.1.1. Apply signature-based permissions

> When sharing data between two apps that you control or own, use signature-based permissions. These permissions don't require user confirmation and instead check that the apps accessing the data are signed using the same signing key. Therefore, these permissions offer a more streamlined, secure user experience.

1.1.2. Provide the right permissions

> Request only the minimum number of permissions necessary for your app to function properly. When possible, relinquish permissions when your app no longer needs them.

1.1.3. Use intents to defer permissions

> Whenever possible, don't add a permission to your app to complete an action that can be completed in another app. Instead, use an intent to defer the request to a different app that already has the necessary permission.
>
> In addition, if your app needs to perform file-based I/O—such as accessing storage or choosing a file—it doesn't need special permissions because the system can complete the operations on your app's behalf. Better still, after a user selects content at a particular URI, the calling app gets granted permission to the selected resource.

1.1.4. Share data securely across apps

> Follow these best practices to share your app's content with other apps in a more secure manner:
>
> Enforce read-only or write-only permissions as needed.
>
> Provide clients one-time access to data by using the FLAG_GRANT_READ_URI_PERMISSION and FLAG_GRANT_WRITE_URI_PERMISSION flags.
>
> When sharing data, use content:// URIs, not file:// URIs. Instances of FileProvider do this for you.

1.1.5. Check availability of storage volume

> If your app interacts with a removable external storage device, keep in mind that the user might remove the storage device while your app is trying to access it. Include logic to verify that the storage device is available.

1.1.6. Check validity of data

> If your app uses data from external storage, make sure that the contents of the data haven't been corrupted or modified. Include logic to handle files that are no longer in a stable format.

1.2. App Security Best Practices

Link: https://developer.android.com/privacy-and-security/security-tips

Link: https://developer.android.com/privacy-and-security/security-tips

1.2.1. App integrity-1

The Play Integrity API helps you check that interactions and server requests are coming from your genuine app binary running on a genuine Android-powered device. By detecting potentially risky and fraudulent interactions, such as from tampered app versions and untrustworthy environments, your app's backend server can respond with appropriate actions to prevent attacks and reduce abuse.

1.2.2. Input validation-1

Insufficient input validation is one of the most common security problems affecting applications, regardless of what platform they run on. Android has platform-level countermeasures that reduce the exposure of applications to input validation issues, and we recommend that you use those features where possible. Also, we recommend using type-safe languages to reduce the likelihood of input validation issues.

1.2.3. Input validation-2

If you are using native code, any data read from files, received over the network, or received from an IPC has the potential to introduce a security issue. The most common problems are buffer overflows, use after free, and off-by-one errors. Android provides a number of technologies, like ASLR and Data Execution Prevention (DEP), that reduce the exploitability of these errors, but they don't solve the underlying problem. You can prevent these vulnerabilities by carefully handling pointers and managing buffers.

1.2.4. Input validation-3

Dynamic, string-based languages such as JavaScript and SQL are also subject to input validation problems due to escape characters and script injection.

1.2.5. Input validation-4

If you are using data within queries that are submitted to an SQL database or a content provider, SQL injection can be an issue. The best defense is to use parameterized queries, as discussed in the section about content providers. Limiting permissions to read-only or write-only can also reduce the potential for harm related to SQL injection.

1.2.6. Input validation-5

If you can't use the security features discussed in this section, make sure to use well-structured data formats and verify that the data conforms to the expected format. While blocking specific characters or performing character replacement can be an effective strategy, these techniques are error prone in practice, and we recommend avoiding them when possible.

1.2.7. User data-1

The best approach for user data security is to minimize the use of APIs that access sensitive or personal information. If you have access to user data, avoid storing or transmitting it if you can. Consider whether your application logic can be implemented using a hash or non-reversible form of the data. For example, your app might use the hash of an email address as a primary key to avoid transmitting or storing the email address. This reduces the chances of inadvertently exposing data, and it also reduces the chance of attackers attempting to exploit your app.

1.2.8. User data-3

Also, consider whether your application could inadvertently expose personal information to other parties, such as third-party components for advertising or third-party services used by your application. If you don't know why a component or service requires personal information, don't provide it. In general, reducing the access to personal information by your application reduces the potential for problems in this area.

1.2.9. User data-4

If your app requires access to sensitive data, evaluate whether you need to transmit it to a server or if you can run the operation on the client. Consider running any code using sensitive data on the client to avoid transmitting user data. Also, make sure that you don't inadvertently expose user data to other applications on the device through overly permissive IPC, world-writable files, or network sockets. Overly permissive IPC is a special case of leaking permission-protected data, discussed in the Permission requests section.

1.2.10. User data-5

If a Globally Unique Identifier (GUID) is required, create a large, unique number and store it. Don't use phone identifiers such as the phone number or IMEI, which might be associated with personal information. This topic is discussed in more detail in the page about best practices for unique identifiers.

1.2.11. User data-6

Be careful when writing to on-device logs. On Android, logs are a shared resource and are available to an application with the READ_LOGS permission. Even though the phone log data is temporary and erased on reboot, inappropriate logging of user information could inadvertently leak user data to other applications. In addition to not logging PII, limit log usage in production apps. To easily implement this, use debug flags and custom Log classes with easily configurable logging levels.

1.2.12. WebView-4

If your application accesses sensitive data with a WebView, consider using the clearCache() method to delete any files stored locally. You can also use server-side headers, such as no-store, to indicate that an application should not cache particular content.

1.2.13. Minimize credential exposure-3

Limit the scope of permissions. Don't request broad permissions for a task that only requires a more narrow scope.

1.2.14. Minimize credential exposure-5

Limit authentication rates. Rapid, successive authentication or authorization requests can be a sign of a brute-force attack. Limit these rates to a reasonable frequency while still allowing for a functional and user-friendly app experience.

1.2.15. Use secure authentication-4

Encrypt communication Use HTTPS and similar technologies to ensure the data your app sends over a network is protected.

1.2.16. Practice secure account management-2

After using AccountManager to retrieve an Account, use CREATOR before passing in any credentials so that you don't inadvertently pass credentials to the wrong application.

1.2.17. Practice secure account management-3

If credentials are used only by applications that you create, you can verify the application that accesses the AccountManager using checkSignatures. Alternatively, if only one application uses the credential, you might use a KeyStore for storage.

1.2.18. Stay vigilant-3

Audit your code. Perform regular security checks against your codebase to look for potential credential request issues.

1.2.19. General best practices-3

Validate and sanitize user input: Validate and sanitize user input to prevent injection attacks that could expose API keys.

1.2.20. Cryptography-1

In addition to providing data isolation, supporting full-filesystem encryption, and providing secure communications channels, Android provides a wide array of algorithms for protecting data using cryptography.

1.2.21. Cryptography-4

If you find that you need to implement your own protocol, don't implement your own cryptographic algorithms. Use existing cryptographic algorithms, such as the implementations of AES and RSA provided in the Cipher class. Additionally, follow these best practices:

1.2.22. Cryptography-5

Use 256-bit AES for commercial purposes. (If unavailable, use 128-bit AES.)

1.2.23. Cryptography-6

Use either 224- or 256-bit public key sizes for elliptic curve (EC) cryptography.

1.2.24. Cryptography-7

Know when to use CBC, CTR, or GCM block modes.

1.2.25. Cryptography-8

Avoid IV/counter reuse in CTR mode. Ensure that they're cryptographically random.

1.2.26. Cryptography-9

When using encryption, implement integrity using the CBC or CTR mode with one of the following functions:

HMAC-SHA1
HMAC-SHA-256
HMAC-SHA-512
GCM mode

1.2.27. Cryptography-10

> Use a secure random number generator, SecureRandom, to initialize any cryptographic keys generated by KeyGenerator. Use of a key that is not generated with a secure random number generator significantly weakens the strength of the algorithm and may allow offline attacks.

1.2.28. Interprocess communication-3

> If your IPC is accessible to other applications, you can apply a security policy by using the element. If the IPC is between apps that are your own and are signed with the same key, use a signature-level permission in the android:protectionLevel.

1.2.29. Intents-3

> Note that ordered broadcasts can be consumed by a recipient, so they might not be delivered to all applications. If you are sending an intent that must be delivered to a specific receiver, you must use an explicit intent that declares the receiver by name.

1.2.30. Intents-5

> Note: Intent filters aren't security features. Components can be invoked with explicit intents and might not have data that would conform to the intent filter. To confirm that it is properly formatted for the invoked receiver, service, or activity, perform input validation within your intent receiver.

1.2.31. Services-1

> A Service is often used to supply functionality for other applications to use. Each service class must have a corresponding declaration in its manifest file.

1.2.32. Services-2

> By default, services aren't exported and can't be invoked by any other application. However, if you add any intent filters to the service declaration, it is exported by default. It's best if you explicitly declare the android:exported attribute to be sure it behaves the way you intend it to. Services can also be protected using the android:permission attribute. By doing so, other applications need to declare a corresponding element in their own manifest to be able to start, stop, or bind to the service.

1.2.33. Services-3

> Note: If your app targets Android 5.0 (API level 21) or higher, use the **JobScheduler** to execute background services.

1.2.34. Services-4

> A service can protect individual IPC calls that are made into it with permissions. This is done by calling checkCallingPermission() before executing the implementation of the call. We recommend using the declarative permissions in the manifest, since those are less prone to oversight.

1.2.35. Services-5

> Caution: Don't confuse client and server permissions; ensure that the called app has appropriate permissions and verify that you grant the same permissions to the calling app.

1.2.36. Binder and Messenger interfaces-3

If you are providing an interface that does require access controls, use checkCallingPermission() to verify whether the caller has a required permission. This is especially important before accessing a service on behalf of the caller, as the identity of your application is passed to other interfaces. If you are invoking an interface provided by a Service, the bindService() invocation can fail if you don't have permission to access the given service. If you need to allow an external process to interact with your app but it doesn't have the necessary permissions to do so, you can use the clearCallingIdentity() method. This method performs the call to your app's interface as though your app were making the call itself, rather than the external caller. You can restore the caller permissions later with the restoreCallingIdentity() method.

1.2.37. Security with dynamically loaded code-2

If your application does dynamically load code, the most important thing to keep in mind is that the dynamically loaded code runs with the same security permissions as the application APK. The user makes a decision to install your application based on your identity, and the user expects that you provide any code run within the application, including code that is dynamically loaded.

1.2.38. Security in native code-1

In general, we recommend using the Android SDK for application development, rather than using native code with the Android NDK. Applications built with native code are more complex, less portable, and more likely to include common memory-corruption errors such as buffer overflows.

1.2.39. Security in native code-2

Android is built using the Linux kernel, and being familiar with Linux development security best practices is especially useful if you are using native code. Linux security practices are beyond the scope of this document, but one of the most popular resources is Secure Programming HOWTO - Creating Secure Software.

1.3. Core App Quality

Link: https://developer.android.com/docs/quality-guidelines/core-app-quality

Link: https://developer.android.com/docs/quality-guidelines/core-app-quality

1.3.1. PS-T6

No debug libraries are included in the production app. This can cause performance as well as security issues.

1.3.2. SC-P1

The app requests only the absolute minimum number of permissions that it needs to support its use case at hand. For some permissions such as location, use coarse location in place of fine location if possible.

1.3.3. SC-P2

The app requests permission to access sensitive data (such as SMS, Call Log, or Location) or services that cost money (such as Dialer or SMS) only when directly related to the core use cases of the apps. Implications related to these permissions should be prominently disclosed to the user.

Depending on how you are using the permissions, there might be an alternative way to fulfill your app's use case without relying on access to sensitive information. For example, instead of requesting permissions related to a user’s contacts, it may be more appropriate to request access by using an implicit intent.

1.3.4. SC-P3

The app requests runtime permissions in context, when the functionality is requested, rather than upfront during app startup.

1.3.5. SC-P4

The app clearly conveys why certain permissions are needed or follow the recommended flow to explain why it needs a permission.

1.3.6. SC-P5

The app should gracefully degrade when users deny or revoke a permission. The app should not prevent the user from accessing the app altogether.

1.3.7. SC-AC1

The app sets the android:exported attribute explicitly for all activities, services, broadcast receivers, and especially content providers.

Only application components that share data with other apps, or components that should be invoked by other apps, are exported.

1.3.8. SC-AC3

All components that share content between your apps use android:protectionLevel="signature" for custom permissions. This includes activities, services, broadcast receivers, and especially content providers.
Apps should not rely on accessing a list of installed packages. The access has been restricted beginning in Android 11.

1.3.9. SC-N1

All network traffic is sent over SSL.

1.3.10. SC-W1

Do not use setAllowUniversalAccessFromFileURLs() for accessing local content. Instead, use WebViewAssetLoader.

1.3.11. SC-W1

WebViews should not use addJavaScriptInterface() with untrusted content.
On Android 6.0 and above, use HTML message channels instead.

1.3.12. SC-E1

The app does not dynamically load code from outside the app's APK. Developers should use Android App Bundles, which includes Play Feature Delivery and Play Asset Delivery.
Starting August 2021, the use of Android App Bundles will become mandatory for all new apps in the Google Play store.

1.3.13. SC-C1

The app uses strong, platform-provided cryptographic algorithms and a random number generator. Also, the app does not implement custom algorithms.

2. Open Web Application Security Project (OWASP)

2.1. Application Security Verification Standard 5.0.0 (ASVS)

Link:
https://scotthelme.co.uk/content/files/2025/06/OWASP_Application_Security_Verification_Standard_5.0.0_en.pdf

Link:
https://scotthelme.co.uk/content/files/2025/06/OWASP_Application_Security_Verification_Standard_5.0.0_en.pdf

2.1.1. V1.1 Encoding and Sanitization Architecture

1.1.1
Verify that input is decoded or unescaped into a canonical form only once, it is only decoded when encoded data in that form is expected, and that this is done before processing the input further, for example it is not performed after input validation or sanitization.

1.1.2
Verify that the application performs output encoding and escaping either as a final step before being used by the interpreter for which it is intended or by the interpreter itself.

2.1.2. V1.2 Injection Prevention

1.2.1
Verify that output encoding for an HTTP response, HTML document, or XML document is relevant for the context required, such as encoding the relevant characters for HTML elements, HTML attributes, HTML comments, CSS, or HTTP header fields, to avoid changing the message or document structure.

1.2.10
Verify that the application is protected against CSV and Formula Injection. The application must follow the escaping rules defined in RFC 4180 sections 2.6 and 2.7 when exporting CSV content. Additionally, when exporting to CSV or other spreadsheet formats (such as XLS, XLSX, or ODF), special characters (including '=', '+', '-', '@', '\t'(tab), and '\0'(null character)) must be escaped with a single quote if they appear as the first character in a field value.

1.2.2
Verify that when dynamically building URLs, untrusted data is encoded according to its context (e.g., URL encoding or base64url encoding for query or path parameters). Ensure that only safe URL protocols are permitted (e.g., disallow javascript: or data:).

1.2.3
Verify that output encoding or escaping is used when dynamically building JavaScript content (including JSON), to avoid changing the message or document structure (to avoid JavaScript and JSON injection).

1.2.4
Verify that data selection or database queries (e.g., SQL, HQL, NoSQL, Cypher) use parameterized queries, ORMs, entity frameworks, or are otherwise protected from SQL Injection and other database injection attacks. This is also relevant when writing stored procedures.

1.2.5
Verify that the application protects against OS command injection and that operating system calls use parameterized OS queries or use contextual command line output encoding.

1.2.6
Verify that the application protects against LDAP injection vulnerabilities, or that specific security controls to prevent LDAP injection have been implemented.

1.2.7
Verify that the application is protected against XPath injection attacks by using query parameterization or precompiled queries.

1.2.8
Verify that LaTeX processors are configured securely (such as not using the "– shell-escape"flag) and an allowlist of commands is used to prevent LaTeX injection attacks.

1.2.9
Verify that the application escapes special characters in regular expressions (typically using a backslash) to prevent them from being misinterpreted as metacharacters.

2.1.3. V1.3 Sanitization

1.3.1
Verify that all untrusted HTML input from WYSIWYG editors or similar is sanitized using a well-known and secure HTML sanitization library or framework feature.

1.3.10
Verify that format strings which might resolve in an unexpected or malicious way when used are sanitized before being processed.

1.3.11
Verify that the application sanitizes user input before passing to mail systems to protect against SMTP or IMAP injection.

1.3.12
Verify that regular expressions are free from elements causing exponential backtracking, and ensure untrusted input is sanitized to mitigate ReDoS or Runaway Regex attacks.

1.3.2
Verify that the application avoids the use of eval() or other dynamic codeexecution features such as Spring Expression Language (SpEL). Where there is no alternative, any user input being included must be sanitized before being executed.

1.3.3
Verify that data being passed to a potentially dangerous context is sanitized beforehand to enforce safety measures, such as only allowing characters which are safe for this context and trimming input which is too long.

1.3.4
Verify that user-supplied Scalable Vector Graphics (SVG) scriptable content is validated or sanitized to contain only tags and attributes (such as draw graphics) that are safe for the application, e.g., do not contain scripts and foreignObject.

1.3.5
Verify that the application sanitizes or disables user-supplied scriptable or expression template language content, such as Markdown, CSS or XSL stylesheets, BBCode, or similar.

1.3.6
Verify that the application protects against Server-side Request Forgery (SSRF) attacks, by validating untrusted data against an allowlist of protocols, domains, paths and ports and sanitizing potentially dangerous characters before using the data to call another service.

1.3.7
Verify that the application protects against template injection attacks by not allowing templates to be built based on untrusted input. Where there is no alternative, any untrusted input being included dynamically during template creation must be sanitized or strictly validated.

1.3.8
Verify that the application appropriately sanitizes untrusted input before use in Java Naming and Directory Interface (JNDI) queries and that JNDI is configured securely to prevent JNDI injection attacks.

1.3.9
Verify that the application sanitizes content before it is sent to memcache to prevent injection attacks.

2.1.4. V1.4 Memory, String, and Unmanaged Code

1.4.1
Verify that the application uses memory-safe string, safer memory copy and pointer arithmetic to detect or prevent stack, buffer, or heap overflows.

1.4.2
Verify that sign, range, and input validation techniques are used to prevent integer overflows.

1.4.3
Verify that dynamically allocated memory and resources are released, and that references or pointers to freed memory are removed or set to null to prevent dangling pointers and use-after-free vulnerabilities.

2.1.5. V1.5 Safe Deserialization

1.5.1
Verify that the application configures XML parsers to use a restrictive configuration and that unsafe features such as resolving external entities are disabled to prevent XML eXternal Entity (XXE) attacks.

1.5.2
Verify that deserialization of untrusted data enforces safe input handling, such as using an allowlist of object types or restricting client-defined object types, to prevent deserialization attacks. Deserialization mechanisms that are explicitly defined as insecure must not be used with untrusted input.

1.5.3
Verify that different parsers used in the application for the same data type (e.g., JSON parsers, XML parsers, URL parsers), perform parsing in a consistent way and use the same character encoding mechanism to avoid issues such as JSON Interoperability vulnerabilities or different URI or file parsing behavior being exploited in Remote File Inclusion (RFI) or Server-side Request Forgery (SSRF) attacks.

2.1.6. V2.1 Validation and Business Logic Documentation

2.1.1
Verify that the application's documentation defines input validation rules for how to check the validity of data items against an expected structure. This could be common data formats such as credit card numbers, email addresses, telephone numbers, or it could be an internal data format._x000D_

2.1.2
Verify that the application's documentation defines how to validate the logical and contextual consistency of combined data items, such as checking that suburb and ZIP code match.

2.1.3
Verify that expectations for business logic limits and validations are documented, including both per-user and globally across the application.

2.1.7. V2.2 Input Validation

2.2.1
Verify that input is validated to enforce business or functional expectations for that input. This should either use positive validation against an allow list of values, patterns, and ranges, or be based on comparing the input to an expected structure and logical limits according to predefined rules. For L1, this can focus on input which is used to make specific business or security decisions. For L2 and up, this should apply to all input.

2.2.2
Verify that the application is designed to enforce input validation at a trusted service layer. While client-side validation improves usability and should be encouraged, it must not be relied upon as a security control

2.2.3
Verify that the application ensures that combinations of related data items are reasonable according to the pre-defined rules

2.1.8. V2.3 Business Logic Security

2.3.1
Verify that the application will only process business logic flows for the same user in the expected sequential step order and without skipping steps.

2.3.2
Verify that business logic limits are implemented per the application’s documentation to avoid business logic flaws being exploited._x000D_

2.3.3
Verify that transactions are being used at the business logic level such that either a business logic operation succeeds in its entirety or it is rolled back to the previous correct state._x000D_

2.3.4
Verify that business logic level locking mechanisms are used to ensure that limited quantity resources (such as theater seats or delivery slots) cannot be double-booked by manipulating the application’s logic.

2.3.5
Verify that high-value business logic flows require multi-user approval to prevent unauthorized or accidental actions. This could include but is not limited to large monetary transfers, contract approvals, access to classified information, or safety overrides in manufacturing.

2.1.9. V2.4 Anti-automation

2.4.1
Verify that anti-automation controls are in place to protect against excessive calls to application functions that could lead to data exfiltration, garbage-data creation, quota exhaustion, rate-limit breaches, denial-of-service, or overuse of costly resources.

2.4.2
Verify that business logic flows require realistic human timing, preventing excessively rapid transaction submissions._x000D_

2.1.10. V3.2 Unintended Content Interpretation

3.2.1
Verify that security controls are in place to prevent browsers from rendering content or functionality in HTTP responses in an incorrect context (e.g., when an API, a user-uploaded file or other resource is requested directly). Possible controls could include: not serving the content unless HTTP request header fields (such as Sec-Fetch-*) indicate it is the correct context, using the sandbox directive of the Content-Security-Policy header field or using the attachment disposition type in the Content-Disposition header field.

3.2.2
Verify that content intended to be displayed as text, rather than rendered as HTML, is handled using safe rendering functions (such as createTextNode or textContent) to prevent unintended execution of content such as HTML or JavaScript.

3.2.3
Verify that the application avoids DOM clobbering when using client-side JavaScript by employing explicit variable declarations, performing strict type checking, avoiding storing global variables on the document object, and implementing namespace isolation.

2.1.11. V3.3 Cookie Setup

3.3.1
Verify that cookies have the ‘Secure’attribute set, and if the ’__Host-’prefix is not used for the cookie name, the ’__Secure-’prefix must be used for the cookie name._x000D_

3.3.2
Verify that each cookie’s ‘SameSite’attribute value is set according to the purpose of the cookie, to limit exposure to user interface redress attacks and browser-based request forgery attacks, commonly known as cross-site request forgery (CSRF).

3.3.3
Verify that cookies have the ’__Host-’prefix for the cookie name unless they are explicitly designed to be shared with other hosts.

3.3.4
Verify that if the value of a cookie is not meant to be accessible to client-side scripts (such as a session token), the cookie must have the ‘HttpOnly’ attribute set and the same value (e. g. session token) must only be transferred to the client via the ‘Set-Cookie’header field.

3.3.5
Verify that when the application writes a cookie, the cookie name and value length combined are not over 4096 bytes. Overly large cookies will not be stored by the browser and therefore not sent with requests, preventing the user from using application functionality which relies on that cookie.

2.1.12. V4.2 HTTP Message Structure Validation

4.2.5
Verify that, if the application (backend or frontend) builds and sends requests, it uses validation, sanitization, or other mechanisms to avoid creating URIs (such as for API calls) or HTTP request header fields (such as Authorization or Cookie), which are too long to be accepted by the receiving component. This could cause a denial of service, such as when sending an overly long request (e.g., a long cookie header field), which results in the server always responding with an error status.

2.1.13. V4.4 WebSocket

4.4.3
Verify that, if the application's standard session management cannot be used, dedicated tokens are being used for this, which comply with the relevant Session Management security requirements.

2.1.14. V5.2 File Upload and Content

5.2.1
Verify that the application will only accept files of a size which it can process without causing a loss of performance or a denial of service attack.

5.2.2
Verify that when the application accepts a file, either on its own or within an archive such as a zip file, it checks if the file extension matches an expected file extension and validates that the contents correspond to the type represented by the extension. This includes, but is not limited to, checking the initial 'magic bytes', performing image re-writing, and using specialized libraries for file content validation. For L1, this can focus just on files which are used to make specific business or security decisions. For L2 and up, this must apply to all files being accepted._x000D_

5.2.3
Verify that the application checks compressed files (e.g., zip, gz, docx, odt) against maximum allowed uncompressed size and against maximum number of files before uncompressing the file.

5.2.4
Verify that a file size quota and maximum number of files per user are enforced to ensure that a single user cannot fill up the storage with too many files, or excessively large files.

5.2.5
Verify that the application does not allow uploading compressed files containing symlinks unless this is specifically required (in which case it will be necessary to enforce an allowlist of the files that can be symlinked to).

5.2.6
Verify that the application rejects uploaded images with a pixel size larger than the maximum allowed, to prevent pixel flood attacks.

2.1.15. V5.3 File Storage

5.3.1
Verify that files uploaded or generated by untrusted input and stored in a public folder, are not executed as server-side program code when accessed directly with an HTTP request.

5.3.2
Verify that when the application creates file paths for file operations, instead of user-submitted filenames, it uses internally generated or trusted data, or if user-submitted filenames or file metadata must be used, strict validation and sanitization must be applied. This is to protect against path traversal, local or remote file inclusion (LFI, RFI), and server-side request forgery (SSRF) attacks._x000D_

5.3.3
Verify that server-side file processing, such as file decompression, ignores user-provided path information to prevent vulnerabilities such as zip slip._x000D_

2.1.16. V5.4 File Download

5.4.1
Verify that the application validates or ignores user-submitted filenames, including in a JSON, JSONP, or URL parameter and specifies a filename in the Content-Disposition header field in the response.

5.4.2
Verify that file names served (e.g., in HTTP response header fields or email attachments) are encoded or sanitized (e.g., following RFC 6266) to preserve document structure and prevent injection attacks.

5.4.3
Verify that files obtained from untrusted sources are scanned by antivirus scanners to prevent serving of known malicious content._x000D_

2.1.17. V6.5 General Multi-factor authentication requirements

6.5.3
Verify that lookup secrets, out-of-band authentication code, and time-based one-time password seeds, are generated using a Cryptographically Secure Pseudorandom Number Generator (CSPRNG) to avoid predictable values.

2.1.18. V6.7 Cryptographic authentication mechanism

6.7.1
Verify that the certificates used to verify cryptographic authentication assertions are stored in a way protects them from modification.

6.7.2
Verify that the challenge nonce is at least 64 bits in length, and statistically unique or unique over the lifetime of the cryptographic device._x000D_

2.1.19. V7.1 Session Management Documentation

7.1.1
Verify that the user’s session inactivity timeout and absolute maximum session lifetime are documented, are appropriate in combination with other controls, and that the documentation includes justification for any deviations from NIST SP 800-63B re-authentication requirements.

7.1.2
Verify that the documentation defines how many concurrent (parallel) sessions are allowed for one account as well as the intended behaviors and actions to be taken when the maximum number of active sessions is reached.

2.1.20. V7.2 Fundamental Session Management Security

7.2.1
Verify that the application performs all session token verification using a trusted, backend service.

7.2.2
Verify that the application uses either self-contained or reference tokens that are dynamically generated for session management, i.e. not using static API secrets and keys.

7.2.3
Verify that if reference tokens are used to represent user sessions, they are unique and generated using a cryptographically secure pseudo-random number generator (CSPRNG) and possess at least 128 bits of entropy._x000D_

7.2.4
Verify that the application generates a new session token on user authentication, including re-authentication, and terminates the current session token.

2.1.21. V7.3 Session Timeout

7.3.1
Verify that there is an inactivity timeout such that re-authentication is enforced according to risk analysis and documented security decisions.

7.3.2
Verify that there is an absolute maximum session lifetime such that re-authentication is enforced according to risk analysis and documented security decisions.

2.1.22. V7.4 Session Termination

7.4.1
Verify that when session termination is triggered (such as logout or expiration), the application disallows any further use of the session. For reference tokens or stateful sessions, this means invalidating the session data at the application backend. Applications using self-contained tokens will need a solution such as maintaining a list of terminated tokens, disallowing tokens produced before a per-user date and time or rotating a per-user signing key.

7.4.2
Verify that the application terminates all active sessions when a user account is disabled or deleted (such as an employee leaving the company).

7.4.3
Verify that the application gives the option to terminate all other active sessions after a successful change or removal of any authentication factor (including password change via reset or recovery and, if present, an MFA settings update).

7.4.5
Verify that application administrators are able to terminate active sessions for an individual user or for all users._x000D_

2.1.23. V7.5 Defenses Against Session Abuse

7.5.2
Verify that users are able to view and (having authenticated again with at least one factor) terminate any or all currently active sessions.

2.1.24. V7.6 Federated Re-authentication

7.6.1
Verify that session lifetime and termination between Relying Parties (RPs) and Identity Providers (IdPs) behave as documented, requiring re-authentication as necessary such as when the maximum time between IdP authentication events is reached.

7.6.2
Verify that creation of a session requires either the user's consent or an explicit action, preventing the creation of new application sessions without user interaction.

2.1.25. V8.2 General Authorization Design

8.2.1
Verify that the application ensures that function-level access is restricted to consumers with explicit permissions.

8.2.2
Verify that the application ensures that data-specific access is restricted to consumers with explicit permissions to specific data items to mitigate insecure direct object reference (IDOR) and broken object level authorization (BOLA).

8.2.3
Verify that the application ensures that field-level access is restricted to consumers with explicit permissions to specific fields to mitigate broken object property level authorization (BOPLA)._x000D_

8.2.4
Verify that adaptive security controls based on a consumer's environmental and contextual attributes (such as time of day, location, IP address, or device) are implemented for authentication and authorization decisions, as defined in the application's documentation. These controls must be applied when the consumer tries to start a new session and also during an existing session.

2.1.26. V8.3 Operation Level Authorization

8.3.1
Verify that the application enforces authorization rules at a trusted service layer and doesn't rely on controls that an untrusted consumer could manipulate, such as client-side JavaScript._x000D_

2.1.27. V8.4 Other Authorization Considerations

8.4.1
Verify that multi-tenant applications use cross-tenant controls to ensure consumer operations will never affect tenants with which they do not have permissions to interact.

2.1.28. V9.1 Token source and integrity

9.1.1
Verify that self-contained tokens are validated using their digital signature or MAC to protect against tampering before accepting the token’s contents.

9.1.2
Verify that only algorithms on an allowlist can be used to create and verify self-contained tokens, for a given context. The allowlist must include the permitted algorithms, ideally only either symmetric or asymmetric algorithms, and must not include the ‘None’algorithm. If both symmetric and asymmetric must be supported, additional controls will be needed to prevent key confusion._x000D_

9.1.3
Verify that key material that is used to validate self-contained tokens is from trusted pre-configured sources for the token issuer, preventing attackers from specifying untrusted sources and keys. For JWTs and other JWS structures, headers such as ‘jku’, ‘x5u’, and ‘jwk’must be validated against an allowlist of trusted sources.

2.1.29. V10.2 OAuth Client

10.2.1
Verify that, if the code flow is used, the OAuth client has protection against browser-based request forgery attacks, commonly known as cross-site request forgery (CSRF), which trigger token requests, either by using proof key for code exchange (PKCE) functionality or checking the ‘state’parameter that was sent in the authorization request.

2.1.30. V10.4 OAuth Authorization Server

10.4.14
Verify that the authorization server issues only sender-constrained (Proof-of-Possession) access tokens, either with certificate-bound access tokens using mutual TLS (mTLS) or DPoP-bound access tokens (Demonstration of Proof of Possession).

10.4.16
Verify that the client is confidential and the authorization server requires the use of strong client authentication methods (based on public-key cryptography and resistant to replay attacks), such as mutual TLS (‘tls_client_auth’, ‘self_signed_tls_client_auth’) or private key JWT (‘private_key_jwt’).

10.4.9
Verify that refresh tokens and reference access tokens can be revoked by an authorized user using the authorization server user interface, to mitigate the risk of malicious clients or stolen tokens.

2.1.31. V10.5 OIDC Client

10.5.5
Verify that, when using OIDC back-channel logout, the relying party mitigates denial of service through forced logout and cross-JWT confusion in the logout flow. The client must verify that the logout token is correctly typed with a value of ‘logout+jwt’, contains the ‘event’claim with the correct member name, and does not contain a ‘nonce’claim. Note that it is also recommended to have a short expiration (e.g., 2 minutes)._x000D_

2.1.32. V10.6 OpenID Provider

10.6.2
Verify that the OpenID Provider mitigates denial of service through forced logout. By obtaining explicit confirmation from the end-user or, if present, validating parameters in the logout request (initiated by the relying party), such as the ‘id_token_hint’.

2.1.33. V10.7 Consent Management

10.7.1
Verify that the authorization server ensures that the user consents to each authorization request. If the identity of the client cannot be assured, the authorization server must always explicitly prompt the user for consent.

10.7.2
Verify that when the authorization server prompts for user consent, it presents sufficient and clear information about what is being consented to. When applicable, this should include the nature of the requested authorizations (typically based on scope, resource server, Rich Authorization Requests (RAR) authorization details), the identity of the authorized application, and the lifetime of these authorizations.

10.7.3
Verify that the user can review, modify, and revoke consents which the user has granted through the authorization server.

2.1.34. V11.1 Cryptographic Inventory and Documentation

11.1.1
Verify that there is a documented policy for management of cryptographic keys and a cryptographic key lifecycle that follows a key management standard such as NIST SP 800-57. This should include ensuring that keys are not overshared (for example, with more than two entities for shared secrets and more than one entity for private keys).

11.1.2
Verify that a cryptographic inventory is performed, maintained, regularly updated, and includes all cryptographic keys, algorithms, and certificates used by the application. It must also document where keys can and cannot be used in the system, and the types of data that can and cannot be protected using the keys._x000D_

11.1.3
Verify that cryptographic discovery mechanisms are employed to identify all instances of cryptography in the system, including encryption, hashing, and signing operations.

11.1.4
Verify that a cryptographic inventory is maintained. This must include a documented plan that outlines the migration path to new cryptographic standards, such as post-quantum cryptography, in order to react to future threats.

2.1.35. V11.2 Secure Cryptography Implementation

11.2.1
Verify that industry-validated implementations (including libraries and hardware-accelerated implementations) are used for cryptographic operations.

11.2.2
Verify that the application is designed with crypto agility such that random number, authenticated encryption, MAC, or hashing algorithms, key lengths, rounds, ciphers and modes can be reconfigured, upgraded, or swapped at any time, to protect against cryptographic breaks. Similarly, it must also be possible to replace keys and passwords and re-encrypt data. This will allow for seamless upgrades to post-quantum cryptography (PQC), once high-assurance implementations of approved PQC schemes or standards are widely available.

11.2.3
Verify that all cryptographic primitives utilize a minimum of 128-bits of security based on the algorithm, key size, and configuration. For example, a 256-bit ECC key provides roughly 128 bits of security where RSA requires a 3072-bit key to achieve 128 bits of security._x000D_

11.2.4
Verify that all cryptographic operations are constant-time, with no ‘short-circuit’operations in comparisons, calculations, or returns, to avoid leaking information.

11.2.5
Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable vulnerabilities, such as Padding Oracle attacks.

2.1.36. V11.3 Encryption Algorithms

11.3.1
Verify that insecure block modes (e.g., ECB) and weak padding schemes (e.g., PKCS#1 v1.5) are not used._x000D_

11.3.2
Verify that only approved ciphers and modes such as AES with GCM are used.

11.3.3
Verify that encrypted data is protected against unauthorized modification preferably by using an approved authenticated encryption method or by combining an approved encryption method with an approved MAC algorithm.

11.3.4
Verify that nonces, initialization vectors, and other single-use numbers are not used for more than one encryption key and data-element pair. The method of generation must be appropriate for the algorithm being used._x000D_

11.3.5
Verify that any combination of an encryption algorithm and a MAC algorithm is operating in encrypt-then-MAC mode.

2.1.37. V11.4 Hashing and Hash-based Functions

11.4.1
Verify that only approved hash functions are used for general cryptographic use cases, including digital signatures, HMAC, KDF, and random bit generation. Disallowed hash functions, such as MD5, must not be used for any cryptographic purpose.

11.4.3
Verify that hash functions used in digital signatures, as part of data authentication or data integrity are collision resistant and have appropriate bit-lengths. If collision resistance is required, the output length must be at least 256 bits. If only resistance to second pre-image attacks is required, the output length must be at least 128 bits.

11.4.4
Verify that the application uses approved key derivation functions with key stretching parameters when deriving secret keys from passwords. The parameters in use must balance security and performance to prevent brute-force attacks from compromising the resulting cryptographic key._x000D_

2.1.38. V11.5 Random Values

11.5.1
Verify that all random numbers and strings which are intended to be non-guessable must be generated using a cryptographically secure pseudo-random number generator (CSPRNG) and have at least 128 bits of entropy. Note that UUIDs do not respect this condition._x000D_

11.5.2
Verify that the random number generation mechanism in use is designed to work securely, even under heavy demand._x000D_

2.1.39. V11.6 Public Key Cryptography

11.6.1
Verify that only approved cryptographic algorithms and modes of operation are used for key generation and seeding, and digital signature generation and verification. Key generation algorithms must not generate insecure keys vulnerable to known attacks, for example, RSA keys which are vulnerable to Fermat factorization._x000D_

11.6.2
Verify that approved cryptographic algorithms are used for key exchange (such as Diffie-Hellman) with a focus on ensuring that key exchange mechanisms use secure parameters. This will prevent attacks on the key establishment process which could lead to adversary-in-the-middle attacksor cryptographic breaks._x000D_

2.1.40. V11.7 In-Use Data Cryptography

11.7.1
Verify that full memory encryption is in use that protects sensitive data while it is in use, preventing access by unauthorized users or processes._x000D_

2.1.41. V12.1 General TLS Security Guidance

12.1.1
Verify that only the latest recommended versions of the TLS protocol are enabled, such as TLS 1.2 and TLS 1.3. The latest version of the TLS protocol must be the preferred option.

12.1.2
Verify that only recommended cipher suites are enabled, with the strongest cipher suites set as preferred. L3 applications must only support cipher suites which provide forward secrecy.

12.1.3
Verify that the application validates that mTLS client certificates are trusted before using the certificate identity for authentication or authorization.

12.1.4
Verify that proper certification revocation, such as Online Certificate Status Protocol (OCSP) Stapling, is enabled and configured.

2.1.42. V12.2 HTTPS Communication with External Facing Services

12.2.2
Verify that external facing services use publicly trusted TLS certificates.

2.1.43. V12.3 General Service to Service Communication Security

12.3.2
Verify that TLS clients validate certificates received before communicating with a TLS server.

12.3.4
Verify that TLS connections between internal services use trusted certificates. Where internally generated or self-signed certificates are used, the consuming service must be configured to only trust specific internal CAs and specific self-signed certificates.

2.1.44. V13.1 Configuration Documentation

13.1.3
Verify that the application documentation defines resource-management strategies for every external system or service it uses (e.g., databases, file handles, threads, HTTP connections). This should include resource-release procedures, timeout settings, failure handling, and where retry logic is implemented, specifying retry limits, delays, and back-off algorithms. For synchronous HTTP request–response operations it should mandate short timeouts and either disable retries or strictly limit retries to prevent cascading delays and resource exhaustion.

13.1.4
Verify that the application’s documentation defines the secrets that are critical for the security of the application and a schedule for rotating them, based on the organization’s threat model and business requirements._x000D_

2.1.45. V13.3 Secret Management

13.3.1
Verify that a secrets management solution, such as a key vault, is used to securely create, store, control access to, and destroy backend secrets. These could include passwords, key material, integrations with databases and third-party systems, keys and seeds for time-based tokens, other internal secrets, and API keys. Secrets must not be included in application source code or included in build artifacts. For an L3 application, this must involve a hardware-backed solution such as an HSM.

13.3.3
Verify that all cryptographic operations are performed using an isolated security module (such as a vault or hardware security module) to securely manage and protect key material from exposure outside of the security module.

13.3.4
Verify that secrets are configured to expire and be rotated based on the application's documentation.

2.1.46. V13.4 Unintended Information Leakage

13.4.5
Verify that documentation (such as for internal APIs) and monitoring endpoints are not exposed unless explicitly intended.

13.4.6
Verify that the application does not expose detailed version information of backend components._x000D_

13.4.7
Verify that the web tier is configured to only serve files with specific file extensions to prevent unintentional information, configuration, and source code leakage.

2.1.47. V14.2 General Data Protection

14.2.1
Verify that sensitive data is only sent to the server in the HTTP message body or header fields, and that the URL and query string do not contain sensitive information, such as an API key or session token._x000D_

14.2.2
Verify that the application prevents sensitive data from being cached in server components, such as load balancers and application caches, or ensures that the data is securely purged after use._x000D_

14.2.3
Verify that defined sensitive data is not sent to untrusted parties (e.g., user trackers) to prevent unwanted collection of data outside of the application's control._x000D_

14.2.5
Verify that caching mechanisms are configured to only cache responses which have the expected content type for that resource and do not contain sensitive, dynamic content. The web server should return a 404 or 302 response when a non-existent file is accessed rather than returning a different, valid file. This should prevent Web Cache Deception attacks.

14.2.6
Verify that the application only returns the minimum required sensitive data for the application's functionality. For example, only returning some of the digits of a credit card number and not the full number. If the complete data is required, it should be masked in the user interface unless the user specifically views it.

14.2.7
Verify that sensitive information is subject to data retention classification, ensuring that outdated or unnecessary data is deleted automatically, on a defined schedule, or as the situation requires.

14.2.8
Verify that sensitive information is removed from the metadata of user-submitted files unless storage is consented to by the user._x000D_

2.1.48. V14.3 Client-side Data Protection

14.3.1
Verify that authenticated data is cleared from client storage, such as the browser DOM, after the client or session is terminated. The 'Clear-Site-Data' HTTP response header field may be able to help with this but the client-side should also be able to clear up if the server connection is not available when the session is terminated.

14.3.2
Verify that the application sets sufficient anti-caching HTTP response header fields (i.e., Cache-Control: no-store) so that sensitive data is not cached in browsers._x000D_

14.3.3
Verify that data stored in browser storage (such as localStorage, sessionStorage, IndexedDB, or cookies) does not contain sensitive data, with the exception of session tokens._x000D_

2.1.49. V15.1 Secure Coding and Architecture Documentation

15.1.3
Verify that the application documentation identifies functionality which is time-consuming or resource-demanding. This must include how to prevent a loss of availability due to overusing this functionality and how to avoid a situation where building a response takes longer than the consumer’s timeout. Potential defenses may include asynchronous processing, using queues, and limiting parallel processes per user and per application._x000D_

2.1.50. V15.2 Security Architecture and Dependencies

15.2.2
Verify that the application has implemented defenses against loss of availability due to functionality which is time-consuming or resource-demanding, based on the documented security decisions and strategies for this.

15.2.5
Verify that the application implements additional protections around parts of the application which are documented as containing “dangerous functionality”or using third-party libraries considered to be “risky components”. This could include techniques such as sandboxing, encapsulation, containerization or network level isolation to delay and deter attackers who compromise one part of an application from pivoting elsewhere in the application._x000D_

2.1.51. V15.3 Defensive Coding

15.3.1
Verify that the application only returns the required subset of fields from a data object. For example, it should not return an entire data object, as some individual fields should not be accessible to users.

15.3.2
Verify that where the application backend makes calls to external URLs, it is configured to not follow redirects unless it is intended functionality.

15.3.3
Verify that the application has countermeasures to protect against mass assignment attacks by limiting allowed fields per controller and action, e.g., it is not possible to insert or update a field value when it was not intended to be part of that action._x000D_

15.3.4
Verify that all proxying and middleware components transfer the user’s original IP address correctly using trusted data fields that cannot be manipulated by the end user, and the application and web server use this correct value for logging and security decisions such as rate limiting, taking into account that even the original IP address may not be reliable due to dynamic IPs, VPNs, or corporate firewalls._x000D_

15.3.5
Verify that the application explicitly ensures that variables are of the correct type and performs strict equality and comparator operations. This is to avoid type juggling or type confusion vulnerabilities caused by the application code making an assumption about a variable type.

15.3.6
Verify that JavaScript code is written in a way that prevents prototype pollution, for example, by using Set() or Map() instead of object literals.

15.3.7
Verify that the application has defenses against HTTP parameter pollution attacks, particularly if the application framework makes no distinction about the source of request parameters (query string, body parameters, cookies, or header fields).

2.1.52. V15.4 Safe Concurrency

15.4.1
Verify that shared objects in multi-threaded code (such as caches, files, or in-memory objects accessed by multiple threads) are accessed safely by using thread-safe types and synchronization mechanisms like locks or semaphores to avoid race conditions and data corruption.

15.4.2
Verify that checks on a resource's state, such as its existence or permissions, and the actions that depend on them are performed as a single atomic operation to prevent time-of-check to time-of-use (TOCTOU) race conditions. For example, checking if a file exists before opening it, or verifying a user's access before granting it._x000D_

15.4.3
Verify that locks are used consistently to avoid threads getting stuck, whether by waiting on each other or retrying endlessly, and that locking logic stays within the code responsible for managing the resource to ensure locks cannot be inadvertently or maliciously modified by external classes or code.

15.4.4
Verify that resource allocation policies prevent thread starvation by ensuring fair access to resources, such as by leveraging thread pools, allowing lower-priority threads to proceed within a reasonable timeframe._x000D_

2.1.53. V16.2 General Logging

16.2.3
Verify that the application only stores or broadcasts logs to the files and services that are documented in the log inventory.

2.1.54. V16.3 Security Events

16.3.2
Verify that failed authorization attempts are logged. For L3, this must include logging all authorization decisions, including logging when sensitive data is accessed (without logging the sensitive data itself)._x000D_

16.3.3
Verify that the application logs the security events that are defined in the documentation and also logs attempts to bypass the security controls, such as input validation, business logic, and anti-automation._x000D_

16.3.4
Verify that the application logs unexpected errors and security control failures such as backend TLS failures.

2.1.55. V16.4 Log Protection

16.4.1
Verify that all logging components appropriately encode data to prevent log injection.

2.1.56. V16.5 Error Handling

15.5.1
Verify that a generic message is returned to the consumer when an unexpected or security-sensitive error occurs, ensuring no exposure of sensitive internal system data such as stack traces, queries, secret keys, and tokens.

15.5.2
Verify that the application continues to operate securely when external resource access fails, for example, by using patterns such as circuit breakers or graceful degradation.

15.5.3
Verify that the application fails gracefully and securely, including when an exception occurs, preventing fail-open conditions such as processing a transaction despite errors resulting from validation logic.

15.5.4
Verify that a “last resort”error handler is defined which will catch all unhandled exceptions. This is both to avoid losing error details that must go to log files and to ensure that an error does not take down the entire application process, leading to a loss of availability.

2.1.57. V17.2 Media

17.2.1
Verify that the key for the Datagram Transport Layer Security (DTLS) certificate is managed and protected based on the documented policy for management of cryptographic keys.

17.2.4
Verify that the media server is able to continue processing incoming media traffic when encountering malformed Secure Real-time Transport Protocol (SRTP) packets._x000D_

17.2.5
Verify that the media server is able to continue processing incoming media traffic during a flood of Secure Real-time Transport Protocol (SRTP) packets from legitimate users._x000D_

17.2.6
Verify that the media server is not susceptible to the “ClientHello”Race Condition vulnerability in Datagram Transport Layer Security (DTLS) by checking if the media server is publicly known to be vulnerable or by performing the race condition test.

17.2.7
Verify that any audio or video recording mechanisms associated with the media server are able to continue processing incoming media traffic during a flood of Secure Real-time Transport Protocol (SRTP) packets from legitimate users.

17.2.8
Verify that the Datagram Transport Layer Security (DTLS) certificate is checked against the Session Description Protocol (SDP) fingerprint attribute, terminating the media stream if the check fails, to ensure the authenticity of the media stream.

2.1.58. V17.3 Signaling

17.3.1
Verify that the signaling server is able to continue processing legitimate incoming signaling messages during a flood attack. This should be achieved by implementing rate limiting at the signaling level.

17.3.2
Verify that the signaling server is able to continue processing legitimate signaling messages when encountering malformed signaling message that could cause a denial of service condition. This could include implementing input validation, safely handling integer overflows, preventing buffer overflows, and employing other robust error-handling techniques._x000D_

2.2. Mobile Application Security Verification Standard (MASVS)

Link: https://github.com/OWASP/owasp-masvs/releases/tag/v2.1.0

Link: https://github.com/OWASP/owasp-masvs/releases/tag/v2.1.0

2.2.1. MASVS-CRYPTO-1

The app employs current strong cryptography and uses it according to industry best practices.

Cryptography plays an especially important role in securing the user's data - even more so in a mobile environment, where attackers having physical access to the user's device is a likely scenario. This control covers general cryptography best practices, which are typically defined in external standards.

2.2.2. MASVS-CRYPTO-2

The app performs key management according to industry best practices.

Even the strongest cryptography would be compromised by poor key management. This control covers the management of cryptographic keys throughout their lifecycle, including key generation, storage and protection.

2.2.3. MASVS-PLATFORM-2

The app uses WebViews securely.

WebViews are typically used by apps that have a need for increased control over the UI. This control ensures that WebViews are configured securely to prevent sensitive data leakage as well as sensitive functionality exposure (e.g. via JavaScript bridges to native code).

2.2.4. MASVS-PLATFORM-3

The app uses the user interface securely.

Sensitive data has to be displayed in the UI in many situations (e.g. passwords, credit card details, OTP codes in notifications). This control ensures that this data doesn't end up being unintentionally leaked due to platform mechanisms such as auto-generated screenshots or accidentally disclosed via e.g. shoulder surfing or sharing the device with another person.

2.2.5. MASVS-CODE-4

The app validates and sanitizes all untrusted inputs.

Apps have many data entry points including the UI, IPC, the network, the file system, etc. This incoming data might have been inadvertently modified by untrusted actors and may lead to bypass of critical security checks as well as classical injection attacks such as SQL injection, XSS or insecure deserialization. This control ensures that this data is treated as untrusted input and is properly verified and sanitized before it's used.

2.2.6. MASVS-PRIVACY-1

The app minimizes access to sensitive data and resources.

Apps should only request access to the data they absolutely need for their functionality and always with informed consent from the user. This control ensures that apps practice data minimization and restricts access control, reducing the potential impact of data breaches or leaks.

Furthermore, apps should share data with third parties only when necessary, and this should include enforcing that third-party SDKs operate based on user consent, not by default or without it. Apps should prevent third-party SDKs from ignoring consent signals or from collecting data before consent is confirmed.

Additionally, apps should be aware of the 'supply chain' of SDKs they incorporate, ensuring that no data is unnecessarily passed down their chain of dependencies. This end-to-end responsibility for data aligns with recent SBOM regulatory requirements, making apps more accountable for their data practices.

2.2.7. MASVS-PRIVACY-2

The app prevents identification of the user.

Protecting user identity is crucial. This control emphasizes the use of unlinkability techniques like data abstraction, anonymization and pseudonymization to prevent user identification and tracking.

Another key aspect addressed by this control is to establish technical barriers when employing complex 'fingerprint-like' signals (e.g. device IDs, IP addresses, behavioral patterns) for specific purposes. For instance, a fingerprint used for fraud detection should be isolated and not repurposed for audience measurement in an analytics SDK. This ensures that each data stream serves its intended function without risking user privacy.

2.2.8. MASVS-PRIVACY-4

The app offers user control over their data.

Users should have control over their data. This control ensures that apps provide mechanisms for users to manage, delete, and modify their data, and change privacy settings as needed (e.g. to revoke consent). Additionally, apps should re-prompt for consent and update their transparency disclosures when they require more data than initially specified._x000D_

3. GSMA

3.1. MDSCert

Link: https://www.gsma.com/solutions-and-impact/technologies/security/wp-content/uploads/2025/02/FS.56-v1.0.pdf

Link: https://www.gsma.com/solutions-and-impact/technologies/security/wp-content/uploads/2025/02/FS.56-v1.0.pdf

3.1.1. FPR_ANO.2.1

> The TSF shall ensure that [THE SYSTEM SOFTWARE OTA CLIENT] is unable to determine the real user data bound to [the TOE (HARDWARE PLATFORM)].
>
> NOTE: To function properly, the OTA client will have a high level of system permissions, and the OTA system may require a combination of a Device ID and a valid user account to access the OTA update. The definition of real user data encompasses user data that is associated with any of the applications or services on the device. This requirement specifies that the real user data on the device is not accessible to the OTA client regardless of the system permissions assigned to the OTA client.

3.1.2. FPR_ANO.2.2

> The TSF shall provide [SYSTEM SOFTWARE UPDATES] to [THE USER] without soliciting any reference to the real user data.

3.1.3. FPT_LNW_EXT.1.1

> The TSF shall ensure that no listening network sockets to the external or loopback IP networks are associated with processes with system permissions on the device.

3.1.4. FPT_LNW_EXT.1.2

> The TSF shall ensure that [assignment: Network sockets and associated processes] exposed to the external or loopback IP networks have no system permission on the device.

3.1.5. ALC_DVS_EXT.1.1D

> The developer shall produce and provide development security documentation on the generation, protection and use of signing keys.

3.1.6. ALC_DVS_EXT.1.2D

> The developer shall produce and provide development security documentation on the acquisition or generation of device unique identifiers.

3.1.7. ALC_DVS_EXT.1.3D

> The developer shall produce and provide development security documentation on the provisioning of data or keys that may be used by a Root of Trust.

3.1.8. ALC_DVS_EXT.1.1C

> The development security documentation shall describe all the physical, procedural, personnel, and other security measures that are necessary to protect the confidentiality and integrity of the keys used to sign the publicly released system software and its updates.

3.1.9. ALC_DVS_EXT.1.2C

The development security documentation shall describe the procedures for selecting the proper signing keys used for a device and to ensure the use of the proper keys in the build process.

3.1.10. ALC_DVS_EXT.1.3C

The development security documentation shall describe all the physical, procedural, personnel, and other security measures that are necessary to protect the confidentiality and integrity of the unique, non- modifiable identifiers (such as IMEI, attestation keys or Device Unique Keys) and how they are properly acquired/created and provisioned for each device.

4. MITRE

4.1. ATT&CK Mobile Application Developer Guidance

Link: https://attack.mitre.org/versions/v18/mitigations/M1013/

Link: https://attack.mitre.org/versions/v18/mitigations/M1013/

4.1.1. Preventing SQL Injection (Secure Coding Practice)

Train developers to use parameterized queries or prepared statements instead of directly embedding user input into SQL queries.

Use Case: A web application accepts user input to search a database. By sanitizing and validating user inputs, developers can prevent attackers from injecting malicious SQL commands.

4.1.2. Cross-Site Scripting (XSS) Mitigation

Require developers to implement output encoding for all user-generated content displayed on a web page.

Use Case: An e-commerce site allows users to leave product reviews. Properly encoding and escaping user inputs prevents malicious scripts from being executed in other users' browsers.

4.1.3. Static Code Analysis in the Build Pipeline

Incorporate tools into CI/CD pipelines to automatically scan for vulnerabilities during the build process.

Use Case: A fintech company integrates static analysis tools to detect hardcoded credentials in their source code before deployment.

4.1.4. T1574.001 DLL

When possible, include hash values in manifest files to help prevent side-loading of malicious libraries.

4.1.5. T1559.003 XPC Services

Enable the Hardened Runtime capability when developing applications. Do not include the com.apple.security.get-task-allow entitlement with the value set to any variation of true.

4.1.6. T1647 Plist File Modification

Ensure applications are using Apple's developer guidance which enables hardened runtime.

4.1.7. T1593.003 Code Repositories

Application developers uploading to public code repositories should be careful to avoid publishing sensitive information such as credentials and API keys.

4.1.8. T1550.001 Application Access Token

Consider implementing token binding strategies, such as Azure AD token protection or OAuth Proof of Possession, that cryptographically bind a token to a secret. This may prevent the token from being used without knowledge of the secret or possession of the device the token is tied to.

4.1.9. T1626 Abuse Elevation Control Mechanism

Applications very rarely require administrator permission. Developers should be cautioned against using this higher degree of access to avoid being flagged as a potentially malicious application.

4.1.10. T1635.001 URI Hijacking

Developers should use Android App Links[11] and iOS Universal Links[12] to provide a secure binding between URIs and applications, preventing malicious applications from intercepting redirections. Additionally, for OAuth use cases, PKCE[13] should be used to prevent use of stolen authorization codes.

5. ETSI

5.1. EN 303 645

Link: https://www.etsi.org/deliver/etsi_en/303600_303699/303645/03.01.03_60/en_303645v030103p.pdf

Link: https://www.etsi.org/deliver/etsi_en/303600_303699/303645/03.01.03_60/en_303645v030103p.pdf

5.1.1. Principle 5.1-1

Where passwords are used to authenticate users against the device or for machine-to-machine authentication, and in any state other than the factory default, all consumer IoT device passwords shall be unique per device or defined by the user.

NOTE 1: There are many mechanisms used for performing authentication, and passwords are not the only mechanism for authenticating a user to a device. However if they are used, following best practice on passwords is encouraged according to NIST Special Publication 800-63B [i.3].

NOTE 2: Standard pairing codes are not considered as passwords used for machine-to-machine authentication.

Many consumer IoT devices are sold with universal default usernames and passwords (such as "admin, admin") for user interfaces through to network protocols. Continued usage of universal default values has been the source of many security issues in IoT [i.17] and the practice needs to be discontinued. The above provision can be achieved by the use of pre-installed passwords that are unique per device and/or by requiring the user to choose a password that follows best practice as part of initialization, or by some other method that does not use passwords.

EXAMPLE 1: During initialization a consumer IoT device generates certificates that are used to authenticate a user to the consumer IoT device via an associated service like a mobile application.

To increase security, multi-factor authentication, such as use of a password plus OTP procedure, can be used to better protect the consumer IoT device or an associated service. Consumer IoT device security can further be strengthened by having unique and immutable identities.

5.1.2. Principle 5.4-4

Any critical security parameters used for integrity and authenticity checks of software updates and for protection of communication with associated services in consumer IoT device software shall be unique per device and shall be produced with a mechanism that reduces the risk of automated attacks against classes of consumer IoT devices.

EXAMPLE 5: A different symmetric key is deployed on every consumer IoT device of the same product class for generating and verifying message authentication codes for software updates.

EXAMPLE 6: The consumer IoT device uses the manufacturer's public key to verify a software update. This is not a critical security parameter and does not need to be unique per device.

Principleing a consumer IoT device with unique critical security parameters helps to protect the integrity and authenticity of software updates as well as the communication of the consumer IoT device with associated services. If global critical security parameters are used, their disclosure can enable wide-scale attacks on other IoT devices such as to enable the creation of botnets.

5.1.3. Principle 5.5-1

The consumer IoT device shall use best practice cryptography to communicate securely.

Appropriateness of security controls and the use of best practice cryptography is dependent on many factors including the properties of technology, operating environment, risk and usage context. As security is ever-evolving it is difficult to give prescriptive advice about cryptography or other security measures without the risk of such advice quickly becoming obsolete.

5.1.4. Principle 5.5-2

The consumer IoT device should use reviewed or evaluated implementations to deliver network and security functionalities, particularly in the field of cryptography.

Reviews and evaluations can involve an independent internal or external entity.

EXAMPLE 1: Distributed software libraries within the development and test community, certified software modules, and hardware equipment crypto-service providers (such as the Secure Element and Trust Execution Environment) are all reviewed or evaluated.

5.1.5. Principle 5.5-3

Cryptographic algorithms and primitives should be replaceable.

NOTE 1: This is a building block for "cryptoagility".

NOTE 2: Changing cryptographic algorithms and primitives in configuration is a security relevant change.

For devices that cannot be updated, it is important that the intended lifetime of the device does not exceed the recommended usage lifetime of cryptographic algorithms used by the device (including key sizes).

The concept of cryptoagility may cover different aspects such as hardware resources of the consumer IoT device including cryptographic hardware accelerators, background infrastructures such as Public Key Infrastructures and development or design processes that anticipate the lifetime of cryptographic algorithms and primitives.

5.1.6. Principle 5.5-4

Access to consumer IoT device functionality via a network interface in the initialized state should only be possible after authentication on that interface.

NOTE 3: Functionality can vary significantly on the use case and can encompass a range of things, including access to personal data and device actuators.

There are consumer IoT devices that provide public, open data for example in the Web of Things [i.18]. These consumer IoT devices are accessible without authentication to provide open access to all.

There are consumer IoT devices that include the ability to make data or services publicly available within a restricted network. These devices will often make assumptions about the network connectivity and/or the sensitivity of the data which is being made publicly available as part of their expected use.

EXAMPLE 2: Smart Television's, expecting to be in a user's home network, can make recorded media publicly
available without user authentication but access restricted to the home network only via services such as DLNA.

The consumer IoT device can be compromised via vulnerabilities in network services. A suitable authentication mechanism can protect against unauthorized access and can contribute to defence-in-depth in the consumer IoT device.

5.1.7. Principle 5.5-5

Consumer IoT device functionality that allows security-relevant changes in configuration via a network interface shall only be accessible after authentication. The exception is for network service protocols that are relied upon by the consumer IoT device and where the manufacturer cannot guarantee what configuration will be required for the consumer IoT device to operate.

NOTE 4: Protocols that are an exception include ARP, DHCP, DNS, ICMP, and NTP.

EXAMPLE 3: Security-relevant changes include permission management, configuration of network keys and password changes.

5.1.8. Principle 5.6-7

Software should run with least necessary privileges, taking account of both security and functionality.

EXAMPLE 9: Minimal daemons/processes run with "root" privileges. In particular the processes that use network interfaces require unprivileged users rather than requiring a "root" user.

EXAMPLE 10: Applications running on a consumer IoT device that includes a multi-user operating system (e.g. Linux®) use different users for each component or service.

NOTE 2: Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Software attacks on consumer IoT devices that aim to corrupt memory can be mitigated through mechanisms such as stack canaries, Address Space Layout Randomization (ASLR). The manufacturer can use platform security features where they are available to help further reduce the risk. Reducing privileges that they run at and minimizing code also helps to mitigate this risk.

5.1.9. Principle 5.6-8

The consumer IoT device should include a hardware-level access control mechanism for memory.

Software exploits often use the lack of access control in memory to execute malicious code. Access control mechanisms limit whether data in memory on the consumer IoT device can be executed. Suitable mechanisms include technologies such as MMUs or MPUs, executable space protection (e.g. NX bits), memory tagging, and trusted execution environments.

5.1.10. Principle 5.7-1

The consumer IoT device should verify its software using secure boot mechanisms.

A hardware root of trust is one way to provide strong attestation as part of a secure boot mechanism. A hardware root of trust is a component of a system from which all other components derive their "trust" - i.e. the source of cryptographic trust within that system. To fulfil its function, the hardware root of trust is reliable and resistant to both physical and logical tampering, as there is no mechanism to determine that the component has failed or been altered. By utilizing a hardware root of trust, a consumer IoT device can have confidence in results of cryptographic functions, such as those utilized for secure boot. A hardware root of trust can be either backed by mechanisms used for secure storage of credentials or other alternatives providing baseline levels of security assurance proportionate to the required level of security for a given consumer IoT device.

5.1.11. Principle 5.7-2

If an unauthorized change is detected to the software, the consumer IoT device should alert the user and/or administrator to the issue and should not connect to wider networks than those necessary to perform the alerting function.

The ability to recover remotely from unauthorized changes can rely on a known good state, such as locally storing a known good version to enable safe recovery and updating of the consumer IoT device. This will avoid denial of service and costly recalls or maintenance visits, whilst managing the risk of potential takeover of the consumer IoT device by an attacker subverting update or other network communications mechanisms.

If a consumer IoT device detects an unauthorized change to its software, it will be able to inform the right stakeholder. In some cases, consumer IoT devices can have the ability to be in administration mode.

EXAMPLE: A thermostat in a room can have a user mode; this mode prevents changing of other settings. If an unauthorized change to software is detected, an alert to the administrator is appropriate, as the administrator has the ability to act on the alert (whereas a user does not).

5.1.12. Principle 5.8-1

The confidentiality of personal data transiting between the consumer IoT device and associated services should be protected with best practice cryptography, appropriate to the properties of the technology, operating environment, risk and usage.

5.1.13. Principle 5.8-2

The confidentiality of sensitive personal data communicated between the consumer IoT device and associated services shall be protected with best practice cryptography, appropriate to the properties of the technology, operating environment, risk and usage.

NOTE: In the context of this provision, "sensitive personal data" is data whose disclosure has a high potential to cause harm to the individual. What is to be treated as "sensitive personal data" varies across products and use cases, but examples are: video stream of a home security camera, payment information, content of communication data and timestamped location data. Carrying out security and data protection impact assessments can help the manufacturer make appropriate choices.

5.1.14. Principle 5.8-3

All external sensing capabilities of the consumer IoT device shall be documented in an accessible way that is clear and transparent for the user.

EXAMPLE: An external sensing capability can be an optic or acoustic sensor.

Clause 6 of the present document contains provisions specific to protecting personal data.

5.1.15. Principle 5.10-1

If telemetry data is collected from consumer IoT products, such as usage and measurement data, it should be examined for security anomalies.

EXAMPLE 1: Security anomalies can be represented by a deviation from normal behaviour of the consumer IoT device, as expressed by the monitored indicators, for example an abnormal increase of failed login attempts.

EXAMPLE 2: Telemetry from multiple devices allows a manufacturer to notice that updates are failing due to invalid software update authenticity checks.

Examining telemetry, including log data, is useful for security evaluation and allows for unusual circumstances to be identified early and dealt with, minimizing security risk and allowing quick mitigation of problems.

Clause 6 of the present document contains provisions specific to protecting personal data when telemetry data is collected.

5.1.16. Principle 5.11-1

Users shall be provided with functionality such that all their user data can be erased from the consumer IoT device in a simple manner.

NOTE 2: User data in this context means all individual data which is stored on the IoT device including personal data, user configuration and cryptographic material such as user passwords or keys.

NOTE 3: User data in this context refers to the data that is created by the user or generated by the device as a result of user activity (e.g. event logs). It does not include data that would be present prior to the user's first usage of the device.

5.1.17. Principle 5.11-2

The consumer should be provided with functionality on the consumer IoT device such that personal data can be deleted from associated services in a simple manner.

5.1.18. Principle 5.11-3

Users should be given clear instructions on how to delete and where possible to erase their personal data from the device and associated services.

5.1.19. Principle 5.11-4

Users should be provided with clear confirmation that personal data has been deleted and where possible erased from devices and associated services.

5.1.20. Principle 5.13-1A

Data input at application layer to the device via user interfaces shall be validated by the device regarding unexpected data input to prevent system manipulations and failures.

EXAMPLE 1: Manipulation of the integrated database via SQL injection over a user interface is mitigated via discarding invalid data input.

Systems can be subverted by incorrectly formatted data or code transferred across different types of interface. Automated tools such as fuzzers can be used by attackers or testers to exploit potential gaps and weaknesses that emerge as a result of not validating data.

EXAMPLE 2: The consumer IoT device receives data that is not of the expected type, for example executable code rather than user inputted text. The software on the consumer IoT device has been written so that the input is parameterized or "escaped", preventing this code from being run.

5.1.21. Principle 5.13-1B

Data input at application layer to the device via network interfaces shall be validated by the device regarding unexpected data input to prevent system manipulations and failures.

NOTE: This includes network accessible APIs without prior authentication and network accessible APIs for authentication mechanisms.

EXAMPLE 3: Out of range data is received by a temperature sensor. Rather than trying to process this input, the consumer IoT device identifies that the data are outside of the possible bounds, discards the data and captures the event as telemetry data.

5.1.22. Principle 6-1

The manufacturer shall provide consumers with clear and transparent information about what personal data is processed and for what purposes, by whom, and for how long, for each consumer IoT device and associated service. This also applies to third parties that can be involved, including advertisers.

EXAMPLE 1: This information could be provided by the manufacturer in a privacy policy.

EXAMPLE 2: A smart health tracker app stores medical information (sleep profiles, pulse readings, blood pressure), and activity information (step counts, running speed and location), from paired smart fitness devices belonging to the user. This information is provided in a centralized service in order for users to track their training activities and change in fitness over time. This data is held by the manufacturer as the provider of the service, it is not made available to any third parties except as regulated by law. The data is retained until either the user deletes it, or the user's account is deactivated (after 90 days of inactivity or by user action).

To support the information to the user, the manufacturer has a process in place to inform the user with a notification if personal data was compromised as described by the vulnerability management process.

5.1.23. Principle 6-2

Where personal data will be processed on the basis of consumers' consent, the consumer IoT device shall provide a means to acquire this consent in a valid way.

Obtaining consent "in a valid way" normally involves giving consumers a free, obvious and explicit opt-in choice of whether their personal data can be used for a specified purpose (see also the ETSI TR 103 621 [i.31] for some examples).

5.1.24. Principle 6-3A

Where personal data will be processed on the basis of consumers' consent, the consumer IoT device shall provide a means to withdraw this consent at any time.

5.1.25. Principle 6-3B

Where personal data will be processed on the basis of consumers' consent, the consumer IoT device shall provide a means of storing information about this consent.

EXAMPLE 3: The device stores timestamps indicating when consent was given or withdrawn, and the data processing purposes covered by the consent.

5.1.26. Principle 6-4

If telemetry data is collected from consumer IoT products, the processing of personal data should be limited to that which is necessary for the intended functionality identified in provision 6-5.

Some telemetry cannot be easily collected without a risk of personal data collection (e.g. crash dumps). Where telemetry data could contain personal data, the use of techniques such as data anonymization can reduce the risk of personal data compromise if the processing does not require the personal data.

5.1.27. Principle 6-5

If telemetry data is collected from consumer IoT products, consumers shall be provided with information on what telemetry data is collected, how it is being used, by whom, and for what purposes.

5.1.28. Principle 6-6

Data stored and processed on a consumer IoT device, or made available to an associated service by the consumer IoT device, for purposes identified in provision 6-1 shall be limited to that which is necessary for the purpose for which it is being collected or processed, and deleted once no longer necessary for any of the purposes identified.

EXAMPLE 4: A smart Television stores user viewing history and user program ratings in order to suggest programs of interest to the user and save them for later viewing. The viewing history is stored on the device for up to 1 year, after which it deemed no longer relevant for this purpose and deleted. User ratings are retained indefinitely on the device but can be deleted by the user. The programs automatically saved are replaced after 28 days by default, with the user able to configure their own retention policy, including indefinite retention. The viewing history is also made available to the operator by the device for the purpose of improving suggestions only if the user has consented to this collection.

5.1.29. Principle 6-7

When the purpose of data collection from consumer IoT devices, or processing on the consumer IoT device, is solely to compute an aggregate result, the data collected should be the minimum required to compute the aggregate, the aggregation should happen as early as possible, and the retention of both collected data and the resulting aggregate should be minimized.

EXAMPLE 5: Federated learning and analytics enable multiple devices to collaboratively train machine learning models or compute data queries, under the coordination of a central server. Each device's raw data is stored locally and not exchanged or transferred; instead, focused updates intended for immediate aggregation are uploaded to achieve the learning objective.

5.1.30. Principle 6-8

Data anonymization technologies should be used to protect privacy during data collection, processing and storage.

EXAMPLE 6: IoT devices may locally add protective noise to data before sending it to centralized aggregators or processing coordinators.

5.2. EN 319 401

Link: https://www.etsi.org/deliver/etsi_en/319400_319499/319401/03.02.00_20/en_319401v030200a.pdf

Link: https://www.etsi.org/deliver/etsi_en/319400_319499/319401/03.02.00_20/en_319401v030200a.pdf

5.2.1. REQ-7.4.1-02

The Trusted Service Provider (TSP) shall administer user access of operators, administrators and other privileged accounts and system auditors applying the principle of "least privileges" when configuring access privileges

5.2.2. REQ-7.4.1-07

The TSP shall provide, modify, remove and document access rights to network and information systems in accordance with the access control policy. In addition:
a) The TSP shall review access rights to privileged and administrator accounts at planned intervals, and access rights shall be modified based on organisational changes.
b) The result of the review, including the necessary changes of access rights, shall be documented.

5.2.3. REQ-7.4.2-01

The TSP shall maintain policies for management of privileged accounts and system administration accounts as part of the access control policy.

5.2.4. REQ-7.4.5-02 b)

Control the allocation to users and management of secret authentication information by a process that ensures the confidentiality of the information, including advising personnel on appropriate handling of authentication information.

5.2.5. REQ-7.4.5-02 e)

Terminate inactive sessions after a predefined period of inactivity.

5.2.6. REQ-7.5-01

Appropriate security controls shall be in place for the management of any cryptographic keys, cryptographic algorithms, and cryptographic devices throughout their lifecycle.

5.2.7. REQ-7.5-02

The TSP shall establish, implement and apply a policy and procedures related to cryptography, with a view to ensuring adequate and effective use of cryptography to protect the confidentiality, authenticity and integrity of data in line with the TSP's asset classification and the results of the risk assessment.

5.2.8. REQ-7.5-03

The policy and procedures for cryptography shall establish:

a) in accordance with the TSP's classification of assets, the type, strength and quality of the cryptographic measures required to protect the TSP's assets, including data at rest and data in transit;

b) based on point a), the protocols or families of protocols to be adopted, as well as cryptographic algorithms, cipher strength, cryptographic solutions and usage practices to be approved and required for use by the TSP, following, where appropriate, a cryptographic agility approach; and

c) the TSP's approach to key management, including, where appropriate, methods for the following:
i. generating different keys for cryptographic systems and applications;
ii. issuing and obtaining public key certificates;
iii. distributing keys to intended entities, including how to activate keys when received;
iv. storing keys, including how authorised users obtain access to keys;
v. changing or updating keys, including rules on when and how to change keys;
vi. dealing with compromised keys;
vii. revoking keys including how to withdraw or deactivate keys;
viii. recovering lost or corrupted keys;
ix. backing up or archiving keys;
x. destroying keys;
xi. logging and auditing of key management-related activities; and
xii. setting activation and deactivation dates for keys ensuring that the keys can only be used for the specified period of time according to the organization's rules on key management.

5.2.9. REQ-7.5-04

The TSP shall review and, where appropriate, update their cryptography policy and procedures at planned intervals, taking into account the state of the art in cryptography.

5.2.10. REQ-7.5-05

For the purpose of the provision of its trust services, the TSP shall select and use suitable cryptographic techniques in accordance with ETSI TS 119 312 [i.28].

5.3. TS 103 606

Link: https://www.etsi.org/deliver/etsi_ts/103600_103699/103606/01.02.01_60/ts_103606v010201p.pdf

Link: https://www.etsi.org/deliver/etsi_ts/103600_103699/103606/01.02.01_60/ts_103606v010201p.pdf

5.3.1. 11.2.1 Mutual TLS Authentication 11.2.1.2 Client certificate 11.2.1.2.4 Client certificate profile

The terminal shall contain a valid client certificate and associated private key to enable the terminal to authenticate to a server over TLS. The client certificate shall be a standard X.509 v3 certificate conforming to IETF RFC 5280 [11] with the additional constraints profiled in table 20 and table 21.
• The public key shall be an RSA key of length at least 2 048 bits.
• The client certificate subject should not include any personal email addresses.
• The client certificate shall be signed by either the Client Root CA or a Client Intermediate CA.

5.3.2. 11.3 Operator application authentication 11.3.1 Encrypted application package overview

This clause defines how to create the encrypted application package using the application ZIP file as defined in clause 7.4 with authentication information. The file is signed, encrypted and packaged into a Cryptographic Message Syntax (CMS) message (IETF RFC 5652 [12]) for delivery to a terminal as defined in clause 11.3.4.

5.3.3. 11.3 Operator application authentication 11.3.2 Operator Signing Certificate

In order to sign an application package, an Operator signing key is used and published in an Operator Signing Certificate. Each Operator Signing Certificate is issued by a self-signed Operator Signing Root CA as described in figure 3. Operators may use an intermediate CA to provide better separation between different signing certificates. The example in figure 3 shows an operator's hierarchy with two country-specific Intermediate CA and three signing certificates.

The Operator Signing Root CA certificate shall have the same profile as a Client Root CA certificate, which is defined in clause 11.2.1.2.3. An Operator Signing Root CA certificate and any intermediate certificate installed in the terminal as Operator Signing trust anchor shall have a validity period of at least 25 years from the point of signing the bilateral agreement.

Operators shall release the Operator Signing Root CA certificate and intervening Operator Signing Intermediate CA's certificates to manufacturers under the bilateral agreement. The details around the release of the operator certificate used for operator application authentication as well as further content of the bilateral agreements are out of scope of the present document. Manufacturers shall include this CA in the list of supported operators in those terminals to which the bilateral agreement applies (see clause 6.6.1).

NOTE: Manufacturers may consider a mechanism for adding additional Operator Signing Root CA certificates to their products if they wish to be able to support operator applications that become available after the manufacture of a terminal.

The Operator Signing Certificate shall be a standard X.509 v3 certificate conforming to IETF RFC 5280 [11] with the following additional constraints:
• The public key shall be an RSA key of length at least 2 048 bits.
• The Operator Signing Certificate subject field should not include any personal email addresses.
• The Operator Signing Certificate shall be signed by either the Operator Signing Root CA or an Operator Signing Intermediate CA.
• The Basic Fields of the Operator Signing Certificate shall further adhere to the constraints in table 22.
• The Extension Fields of the Operator Signing Certificate shall further adhere to the constraints listed in table 23.
The present document only permits key encryption using RSAES-PKCS-v1.5/PKCS#1 v1.5 identified as 'rsaEncryption' in IETF RFC 3447 [18].

5.3.4. 11.3 Operator application authentication 11.3.3 Terminal Packaging Certificate

Manufacturers may create a hierarchy as described in figure 4 including the use of Intermediate CAs to provide better separation between different packaging certificates.

Manufacturers shall create a Terminal Packaging Certificate that is either self-signed or signed by a self-signed Manufacturer Packaging Root CA certificate or signed by a Terminal Packaging Intermediate CA. Manufacturers shall use the profiles detailed in clause 11.2.1.2.3 to generate a Manufacturer Packaging Root CA. The Manufacturer Packaging Root CA certificate is not required to have a minimum validity period.

Manufacturers shall release the relevant Terminal Packaging Certificates to operators under the bilateral agreement. The release process is out of scope for the present document. Operators shall use all of the provided Terminal Packaging Certificates in the process of creating the encrypted application package as defined in clause 11.3.4.3.

The Terminal Packaging Certificate shall be a standard X.509 v3 certificate conforming to IETF RFC 5280 [11] with the following additional constraints:
• The public key shall be an RSA key of length at least 2 048 bits.
• The Terminal Packaging Certificate subject field should not include any personal email addresses.
• The Terminal Packaging Certificate shall not expire during the lifetime of the bilateral agreement.
• The Basic Fields of the Terminal Packaging Certificate shall further adhere to the constraints in table 24.
• The Extension Fields of the Terminal Packaging Certificate shall further adhere to the constraints listed in table 25.
The present document only permits key encryption using RSAES-PKCS-v1.5/PKCS#1 v1.5 identified as 'rsaEncryption' in IETF RFC 3447 [18].

5.3.5. 11.3.4 Encrypted application packaging process 11.3.4.2 Operator application signing process

This clause describes the process of packaging an application ZIP file into a CMS SignedData structure. This structure is then encrypted into a CMS EnvelopedData structure according to clause 11.3.4.3.

A CMS SignedData shall be constructed according to section 5 of IETF RFC 5652 [12], with the following additional requirements:
• The signer-specific message-digest algorithm for generating the messageDigest shall be either SHA256 or SHA384 (as signalled through in the Operator Signing Certificate).
• Although IETF RFC 5652 [12] specifies that certificates are optional, the Operator Signing Certificate and any Operator Intermediate CAs shall be included in the certificates block within the CMS SignedData. The Operator Signing Certificate shall be profiled as defined in clause 11.3.2.
• The application ZIP file shall be included in the CMS SignedData under the EncapsulatedContentInfo field, with the eContentType field set to id-data.

5.3.6. 11.3.4 Encrypted application packaging process 11.3.4.3 Process for encrypting an application package

To create the final encrypted application package, the operator shall use the CMS SignedData defined in clause 11.3.4.2 to create an CMS EnvelopedData defined in section 6 of IETF RFC 5652 [12], with the following additional requirements:
• The contentEncryptionAlgorithm used to encrypt the data shall be set to either aes-128-cbc or aes-256-cbc.
• In the creation of the CMS EnvelopedData, Operators shall use the Terminal Packaging Certificates of all potential recipients of the encrypted application package. The encrypted content-encryption key and other recipient-specific information shall be used as defined in section 6.2 of IETF RFC 5652 [12] using a ktri field of type KeyTransRecipientInfo as described in section 6.2.1 of IETF RFC 5652 [12].
• Operators shall ensure that the content-encryption key is randomly generated for each generation of the encrypted applicationPackage as detailed in section 6.2 of the IETF RFC 5652 [12].

5.3.7. 11.3.4 Encrypted application packaging process 11.3.4.4 Process for decrypting an application package

After downloading the encrypted application package as defined in clause 6.1.7 , terminals shall decrypt the encrypted application package to access the application zip file according to the following process.

The terminal shall use the private key of the Terminal Packaging Certificate to decrypt the EncryptedKey of the relevant RecipientInfo field of the received encrypted application package as defined in section 6.2 of IET RFC 5652 [12]. The terminal shall use the decrypted key to decrypt the encryptedContent field, resulting in the CMS SignedData structure.

The terminal shall support the following cipher algorithms:
• aes-128-cbc
• aes-256-cbc

Terminals encountering other cipher algorithms shall fail the decryption process and follow the process outlined in clause 6.1.9.

5.3.8. 12 Privacy

In the case of privileged operator applications, use of an operator application is entirely a user choice. Terminals are also required to allow the user to uninstall a privileged operator application (see clause 6.7).

Operator applications will need to include appropriate terms and conditions as part of the package and obtain user agreement to these when they are first launched after being installed.
• For a privileged operator application, if the user does not accept the terms and conditions then the user can uninstall the operator application or the operator application can uninstall itself (see clause 6.7).
• For an operator-specific operator application, if the user does not accept the terms and conditions then they have to return the terminal to where they got it from.

Operator applications are securely delivered to the terminal and have a capability for secure communications allowing any necessary data exchanges with operators to be kept confidential.

5.4. TS 103 732

Link: https://www.etsi.org/deliver/etsi_ts/103700_103799/10373204/01.01.01_60/ts_10373204v010101p.pdf

Link: https://www.etsi.org/deliver/etsi_ts/103700_103799/10373204/01.01.01_60/ts_10373204v010101p.pdf

5.4.1. O.LIMITED_PERMISSIONS

The Target Of Evaluation (TOE) shall provide system permissions only to the preinstalled applications under TOE manufacturer control.

5.4.2. FAP_PRM.1

System permissions to TOE functionality are restricted only to all the preloaded applications and solely to the preinstalled application created by the TOE developer.
FAP_PRM.1.1 The TSF shall restrict the ability of applications to request [assignment: permissions defined as system permission by the TOE developer] to only preloaded applications and preinstalled applications created by the TOE developer.

5.4.3. FAP_RSK.2

Cryptographic functions require proper key generation algorithms.
FAP_RSK.2.1 The applications on the TOE shall use appropriate cryptographic primitives to support the algorithms in use.
FAP_RSK.2.2 The applications on the TOE shall not use deprecated cryptographic modes or algorithms.

5.4.4. FAP_RSK.3

Encryption of stored data is required to be done using properly generated cryptographic keys.
FAP_RSK.3.1 The applications on the TOE shall not have hardcoded symmetric keys as the method of internal data protection.

5.4.5. FAP_RSK.6

To ensure a secure connection to the proper network endpoints, certificate checks shall be performed on the server certificate.
FAP_RSK.6.1 "The applications on the TOE shall validate the X.509 server certificate according to the following rules:
• The certificate path shall terminate with a certificate in the TOE trust anchor.
• The TOE shall use the main OS-provided method to verify the certificate."

5.4.6. FAP_RSK.7

Application input, whether from APIs or the user, requires sanitization checks to prevent malformed data from being accepted.
FAP_RSK.7.1 The applications on the TOE shall sanitize all input from external sources via any available interface.

6. NIAP

6.1. NIAP Profile Protection

Link: https://www.niap-ccevs.org/static_html/protection-profile/516/PP_APP_V2.0.htm

Link: https://www.niap-ccevs.org/static_html/protection-profile/516/PP_APP_V2.0.htm

6.1.1. 5.1.1 Cryptographic Support (FCS) FCS_CKM_EXT.1 Cryptographic Key Generation Services FCS_CKM_EXT.1.1

The application shall [selection:

Generate no asymmetric cryptographic keys

Invoke platform-provided functionality for asymmetric key generation

Implement asymmetric key generation
].

6.1.2. FCS_CKM.1.1/AK

The application shall [selection:

invoke platform-provided functionality
implement functionality

] to generate asymmetric cryptographic keys in accordance with a specified cryptographic key generation algorithm

6.1.3. FCS_CKM.1.1/SK

The application shall [selection: invoke platform-provided functionality, implement functionality] to generate symmetric cryptographic keys using a Random Bit Generator as specified in FCS_RBG_EXT.1 and specified cryptographic key sizes 256-bit.

6.1.4. FCS_CKM.2.1

The application shall [selection: invoke platform-provided functionality, implement functionality] to perform cryptographic key establishment in accordance with a specified cryptographic key establishment method.

6.1.5. FCS_RBG_EXT.1 Random Bit Generation Services FCS_RBG_EXT.1.1

The application shall:

Use no Deterministic Random Bit Generator (DRBG) functionality

Invoke platform-provided DRBG functionality

Implement DRBG functionality

6.1.6. FCS_RBG.1.1

The TOE Security Functionality (TSF) shall perform deterministic random bit generation services using [selection:
Hash_DRBG (any)
HMAC_DRBG (any)
CTR_DRBG (AES)
] in accordance with [NIST SP 800-90A] after initialization with a seed.

6.1.7. FCS_RBG.1.2

The TSF shall use a [selection: TSF noise source [assignment: name of noise source] , multiple TSF noise sources [assignment: names of noise sources] , TSF interface for seeding] for initialized seeding.

6.1.8. FCS_RBG.1.3

The TSF shall update the RBG state by [selection: reseeding, uninstantiating and reinstantiating] using a [selection: TSF noise source [assignment: name of noise source] , TSF interface for seeding] in the following situations: [selection:
on demand
on the condition: [assignment: condition]
after [assignment: time]
] in accordance with [assignment: list of standards].

6.1.9. FCS_RBG.2.1

The TSF shall be able to accept a minimum input of [assignment: minimum input length, in bits, greater than zero] from a TSF interface for the purpose of seeding.

6.1.10. FCS_RBG.3.1

The TSF shall be able to seed the RBG using a TSF software-based noise source with a minimum of [assignment: number of bits] bits of min-entropy.

6.1.11. FCS_RBG.4.1

The TSF shall be able to seed the RBG using [assignment: number] TSF software-based noise source(s).

6.1.12. FCS_RBG.5.1

The TSF shall [assignment: combining operation] [selection: output from TSF noise source(s), input from TSF interface(s) for seeding] to create the entropy input into the derivation function as defined in [assignment: list of standards] , resulting in a minimum of [assignment: number of bits] bits of min-entropy.

6.1.13. FPT_FLS.1.1

The TSF shall preserve a secure state when the following types of failures occur: [DRBG self-test failure].

6.1.14. FPT_TST.1.1

The TSF shall run a suite of the following self-tests [selection: during initial start-up, periodically during normal operation, at the request of the authorized user, at the conditions [assignment: conditions under which self-test should occur]] to demonstrate the correct operation of [TSF DRBG specified in FCS_RBG.1].

6.1.15. FCS_PBKDF_EXT.1.1

The application shall condition passwords/passphrases with [assignment: Password-based Key Derivation Functions] in accordance with a specified cryptographic algorithm as specified in FCS_COP.1 /KeyedHash , with [selection: [assignment: positive integer between 1,000 and 9,999], [assignment: positive integer between 1,0000 and 199,999], [assignment: positive integer greater than 200,000]] iterations, and output size of [assignment: positive integer of 256 or greater] bits that meet the following [NIST SP 800-132].

6.1.16. FCS_PBKDF_EXT.1.2

The TSF shall generate salts in accordance with FCS_SNI_EXT.1 and with entropy corresponding to the security strength selected for PBKDF in FCS_PBKDF_EXT.1.

6.1.17. FDP_DEC_EXT.1 Access to Platform Resources FDP_DEC_EXT.1.1

The application shall restrict its access to only [selection:
no hardware resources
network connectivity
camera
microphone
location services
NFC
USB
Bluetooth
[assignment: list of additional hardware resources]
].

6.1.18. FDP_DEC_EXT.1.2

The application shall restrict its access to only [selection:
no sensitive information repositories
address book
calendar
call lists
system logs
[assignment: list of additional sensitive information repositories]
].

6.1.19. FDP_NET_EXT.1 Network Communications FDP_NET_EXT.1.1

The application shall restrict network communication to [selection:
no network communication
user-initiated communication for [assignment: list of functions for which the user can initiate network communication]
respond to [assignment: list of remotely initiated communication]
[assignment: list of application-initiated network communication]
].

6.1.20. FMT_SMF.1 Specification of Management Functions FMT_SMF.1.1

The TSF shall be capable of performing the following management functions [selection:
no management functions
enable/disable the transmission of any information describing the system's hardware, software, or configuration
enable/disable the transmission of any Personally Identifiable Information (PII)
enable/disable transmission of any application state (e.g. crashdump) information
enable/disable network backup functionality to [assignment: list of enterprise or commercial cloud backup systems]
[assignment: list of other management functions to be provided by the TSF]
].

6.1.21. 5.1.4 Privacy (FPR) FPR_ANO_EXT.1 User Consent for Transmission of Personally Identifiable Information FPR_ANO_EXT.1.1

The application shall [selection, choose one of:
    not use PII
    not transmit PII over a network
    require user approval before executing [assignment: list of functions that transmit PII over a network]
].

6.1.22. 5.1.5 Protection of the TSF (FPT) FPT_AEX_EXT.1 Anti-Exploitation Capabilities FPT_AEX_EXT.1.1

The application shall not request to map memory at an explicit address except for [assignment: list of explicit exceptions].

6.1.23. FPT_AEX_EXT.1.2

The application shall [selection, choose one of:
    not allocate any memory region with both write and execute permissions
    allocate memory regions with write and execute permissions for only [assignment: list of functions performing just-in-time compilation]
].

6.1.24. FPT_AEX_EXT.1.3

The application shall be compatible with security features provided by the platform vendor.

6.1.25. FPT_AEX_EXT.1.4

The application shall not write user-modifiable files to directories that contain executable files unless explicitly directed by the user to do so.

6.1.26. FPT_AEX_EXT.1.5

The application shall be built with stack-based buffer overflow protection enabled.

6.1.27. FCS_COP.1.1/Hash

The application shall perform [cryptographic hashing services] in accordance with a specified cryptographic algorithm [selection:
    SHA-256
    SHA-384
    SHA-512
] and message digest sizes [selection:
    256
    384
    512
] bits that meet the following: [FIPS Pub 180-4, "Secure Hash Standard"].

6.1.28. FCS_COP.1.1/KeyedHash

The application shall perform [keyed-hash message authentication] in accordance with a specified cryptographic algorithm [selection:
HMAC-SHA-256
HMAC-SHA-384
HMAC-SHA-512
] with key sizes [assignment: key size (in bits) used in HMAC] and message digest sizes [selection: 256, 384, 512] bits that meet the following: [FIPS Pub 198-1, "The Keyed-Hash Message Authentication Code," and FIPS Pub 180-4, "Secure Hash Standard"].

6.1.29. FCS_COP.1.1/SigGen

The application shall perform [cryptographic signature services (generation)] in accordance with a specified cryptographic algorithm:
• CNSA 2.0 Compliant Algorithm:
○ Module-Lattice-Based Digital Signature Standard using the parameter set ML-DSA-87 that meets the following [FIPS 204, Module-Lattice-Based Digital Signature Standard]

• CNSA 1.0 Compliant Algorithms: [selection:
○ RSA schemes using cryptographic key sizes of [3072-bit or greater] that meet the following: [FIPS PUB 186-5, “Digital Signature Standard (DSS),” Section 5]
○ ECDSA schemes using [“NIST curves” [selection: P-384, P-521]] that meet the following: [FIPS PUB 186-5, “Digital Signature Standard (DSS),” Section 6]

6.1.30. FCS_COP.1.1/SigVer

The application shall perform [cryptographic signature services (verification)] in accordance with a specified cryptographic algorithm [selection:
• CNSA 2.0 Compliant Algorithms: [selection:
○ Leighton-Micali Signature Algorithm for verification using cryptographic key sizes of [selection: 192, 256] bits that meet the following [NIST SP 800-208, "Recommendation for Stateful Hash-Based Signature Schemes"]
○ eXtended Merkle Signature Scheme Algorithm for verification using cryptographic key sizes of [selection: 192, 256] bits that meets the following: [NIST SP 800-208, "Recommendation for Stateful Hash-Based Signature Schemes"]
○ Module-Lattice-Based Digital Signature Standard using the parameter set ML-DSA-87 that meets the following [FIPS 204, Module-Lattice-Based Digital Signature Standard]
]
• CNSA 1.0 Compliant Algorithms: [selection:
○ RSA schemes using cryptographic key sizes of [3072-bit or greater] that meet the following: [FIPS PUB 186-5, “Digital Signature Standard (DSS),” Section 5]
○ ECDSA schemes using [“NIST curves” [selection: P-384, P-521]] that meet the following: [FIPS PUB 186-5, “Digital Signature Standard (DSS),” Section 6]
]
].

6.1.31. FCS_COP.1.1/SKC

> The application shall [selection: perform, invoke the platform to perform] [encryption and decryption] in accordance with a specified cryptographic algorithm [selection:
> AES-CBC (as defined in NIST SP 800-38A) mode
> AES-GCM (as defined in NIST SP 800-38D) mode
> AES-XTS (as defined in NIST SP 800-38E) mode
> AES-CCM (as defined in NIST SP 800-38C) mode
> AES-CTR (as defined in NIST SP 800-38A) mode
>] and cryptographic key size of [256-bits].

6.1.32. FCS_SNI_EXT.1.1

> The application shall [selection: use no salts, use salts that are generated by a DRBG as specified in FCS_RBG_EXT.1]

6.1.33. FCS_SNI_EXT.1.2

> The application shall use [selection: no nonces, unique nonces with a minimum size of [64] bits.]

6.1.34. FCS_SNI_EXT.1.3

> The application shall:
> use no IVs
> create IVs in the following manner [selection:
> CBC: IVs shall be non-repeating and unpredictable;
> CCM: Nonce shall be non-repeating;
> CTR: "Initial Counter" shall be non-repeating. No counter value shall be repeated across multiple messages with the same secret key.
> XTS: No IV. Tweak values shall be non-negative integers, assigned consecutively, and starting at an arbitrary non-negative integer;
> GCM: IV shall be non-repeating. The number of invocations of GCM shall not exceed 2^32 for a given secret key. The IV constructed using one of two allowed construction methods given in Section 8.2 of NIST SP 800-38D.

6.1.35. FPT_API_EXT.2.1

> The application [selection, choose one of: shall use platform-provided libraries, does not implement functionality] for parsing [assignment: list of formats parsed that are included in the IANA MIME media types].

7. App Defence Alliance

7.1. Application Security Assessment

Link: https://github.com/appdefensealliance/ASA-WG/blob/main/README.md

Link: https://github.com/appdefensealliance/ASA-WG/blob/main/README.md

7.1.1. Mobile 1.5.1.1 (Android)

> The app shall limit content provider exposure and harden queries against injection attacks.

7.1.2. Mobile 1.5.1.2 (Android)

> The app shall use verified links and sanitize all link input data.

7.1.3. Mobile 1.5.1.4 (Android)

All Pending Intents shall be immutable or otherwise justified for mutability.

7.1.4. Mobile 1.6.3.2 (Android)

The App shall Mitigate Against Injection Flaws in Content Providers.

7.1.5. Mobile 1.6.3.3 (Android)

Arbitrary URL redirects shall not be included in the app's webviews.

7.1.6. Mobile 1.6.3.4 (Android)

Any use of implicit intents shall be appropriate for the app's functionality and any return data shall be handled securely.

7.1.7. Mobile 1.7.1.1 (Android)

The app shall be properly signed.

7.1.8. Mobile 1.8.2.1 (Android)

The app shall be transparent about data collection and usage.

7.1.9. Mobile 1.8.3.1 (Android)

Users shall have the ability to request their data to be deleted via an in-app mechanism.

7.1.10. Web 2.1.1

The application shall not reveal passwords or session tokens in URL parameters. In cases where the application provides an API, the application shall prevent (or give developers an option) to prevent exposing sensitive information like API keys or session tokens within the URL query strings.

7.1.11. Web 2.2.1

Users shall have the ability to logout of the application. Logout or session expiration shall invalidate all stateful session tokens, including refresh tokens.

7.1.12. Web 2.2.2

The application shall provide the option (or acts by default) to terminate all other active sessions, including stateful refresh tokens, after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties.

7.1.13. Web 2.2.3

Non-revocable sateless authentication tokens must expire within 24 hours of being issued.

7.1.14. Web 2.3.1

Cookie-based session tokens shall have the 'Secure' attribute set.

7.1.15. Web 2.3.2

Cookie-based session tokens shall have the 'HttpOnly' attribute set.

7.1.16. Web 2.3.3

The application shall use session tokens rather than static API secrets and keys, except with legacy implementations.

7.1.17. Web 2.3.4

Stateless session tokens shall use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.

7.1.18. Web 2.4.1

Verify the application ensures a full, valid login session or requires re-authentication or secondary verification before allowing any sensitive transactions or account modifications.

7.1.19. Web 3.1.1

The application shall enforce least privilege access control rules on a trusted service layer.

7.1.20. Web 3.1.2

All user and data attributes and policy information used by access controls shall not be able to be manipulated by end users unless specifically authorized.

7.1.21. Web 3.1.3

Access controls shall fail securely including when an exception occurs.

7.1.22. Web 3.1.4

Sensitive resources shall be protected against Insecure Direct Object Reference (IDOR) attacks targeting creation, reading, updating and deletion of records, such as creating or updating someone else's record, viewing everyone's records, or deleting all records.

7.1.23. Web 3.1.5

The application or framework shall enforce a strong anti-CSRF mechanism to protect authenticated functionality, and effective anti-automation or anti-CSRF protects unauthenticated functionality.

7.1.24. Web 4.1.3

No instances of weak cryptography which meaningfully impact the confidentiality or integrity of confidential data.

7.1.25. Web 6.3.1

The origin header shall not be used for authentication of access control decisions.

7.1.26. Web 6.4.1

The application shall not be susceptible to subdomain takeovers.

7.1.27. Cloud 1.6.1

Google: Ensure That Instances Are Not Configured To Use the Default Service Account.

7.1.28. Cloud 2.4.1

Azure: Ensure User consent for applications is set to Do not allow user consent.

7.1.29. Cloud 2.5.1

Azure: Ensure that the Expiration Date that is no more than 90 days in the future is set for all Keys in RBAC Key Vaults.

7.1.30. Cloud 2.5.2

Azure: Ensure that the Expiration Date that is no more than 90 days in the future is set for all Keys in Non-RBAC Key Vaults.

7.1.31. Cloud 2.5.3

Azure: Ensure that the Expiration Date that is no more than 90 days in the future is set for all Secrets in RBAC Key Vaults.

7.1.32. Cloud 2.5.4

Azure: Ensure that the Expiration Date that is no more than 90 days in the future is set for all Secrets in Non-RBAC Key Vaults.

7.1.33. Cloud 2.8.1

Azure: Ensure Security Defaults is enabled on Azure Active Directory.

7.1.34. Cloud 5.8.1

Azure: Ensure that Shared Access Signature Tokens Expire Within an Hour.

7.1.35. Cloud 6.11.1

Azure: Ensure that Azure Active Directory Admin is Configured for SQL Servers.

8. Apple

8.1. Apple Developer Security

Link: https://developer.apple.com/documentation/security

Link: https://developer.apple.com/documentation/security

8.1.1. Authorization and Authentication Sessions Overview

Use the Security.AuthSession API to work with session management and inquiry functions.

8.1.2. Secure Data Preventing Insecure Network Connections Ensure the Network Server Meets Minimum Requirements

A secure server establishes its identity using an X.509 digital certificate. A connecting client examines this certificate to perform default server trust evaluation, which includes checking that the certificate:

Has an intact digital signature, showing that the certificate hasn't been tampered with.

Isn't expired.

Has a name that matches the server's DNS name.

Is signed by another valid certificate, which is in turned signed by another, and so on back to a trusted anchor certificate, which must be issued by a Certificate Authority (CA). The anchor certificate must be part of the client operating system, as indicated in Lists of available trusted root certificates in iOS, or be installed on the client by the user or a system administrator.

ATS requires all of these things, and then provides extended security checks:

The server certificate must be signed with either a Rivest-Shamir-Adleman (RSA) key of at least 2048 bits, or an Elliptic-Curve Cryptography (ECC) key of at least 256 bits.

The certificate must use the Secure Hash Algorithm 2 (SHA-2) with a digest length, sometimes called a fingerprint, of at least 256 bits (that is, SHA-256 or greater).

The connection must use Transport Layer Security (TLS) protocol version 1.2 or later.

Data must be exchanged using either the AES-128 or the AES-256 symmetric cipher.

The link must support perfect forward secrecy (PFS) through Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange.

8.1.3. Secure Data Preventing Insecure Network Connections Configure Exceptions Only When Needed; Prefer Server Fixes

ATS disallows a connection if the server fails to meet one of the security checks discussed in the previous section. Your best response is to update the server. If you can't do that for some reason, you can specify exceptions in your app to disable one or more aspects of ATS.

8.1.4. Secure Code Code Signing Services Overview

Code signing is a macOS security technology that you use to certify that an app was created by you. Once an app is signed, the system can detect any change to the app—whether the change is introduced accidentally or by malicious code. You can control how your signed code loads signed plug-ins and other signed code without invalidating the signatures of the host code or of the guest (dynamically loaded) code.

You work with code objects that represent uniquely identified elements of running code in the system. In addition to UNIX processes, these elements can include scripts, applets, widgets, and so forth. You also work with static code objects that represent code in the file system. Static code includes applications, tools, frameworks, plug-ins, scripts, and so on. Generally, a code object has a specific static code object from which it originates and that holds its static signing data. The reverse, however, is not true—given a static code object, it is not possible to find, enumerate, or control any code object that originated from it.

8.1.5. Secure Code Notarizing macOS software before distribution Overview

Notarize your macOS software to give users more confidence that the Developer ID-signed software you distribute has been checked by Apple for malicious components. Notarization of macOS software is not App Review. The Apple notary service is an automated system that scans your software for malicious content, checks for code-signing issues, and returns the results to you quickly. If there are no issues, the notary service generates a ticket for you to staple to your software; the notary service also publishes that ticket online where Gatekeeper can find it.

When the user first installs or runs your macOS software, the presence of a ticket (either online or attached to the executable) tells Gatekeeper that Apple notarized the software. Gatekeeper then places descriptive information in the initial launch dialog to help the user make an informed choice about whether to launch the app.

You can notarize several different types of software deliverables, including:

- macOS apps
- Non-app bundles, such as kernel extensions
- Disk images (UDIF format)
- Flat installer packages

Notarization also protects your users if your Developer ID signing key is exposed. The notary service maintains an audit trail of the software distributed using your signing key. If you discover unauthorized versions of your software, you can work with Apple to revoke the tickets associated with those versions.

8.1.6. Secure Code Notarizing macOS software before distribution Prepare your software for notarization

Apple’s notary service requires you to adopt the following protections:

Enable code-signing for all of the executables you distribute, and ensure that executables have valid code signatures, as described in Ensure a valid code signature.

Use a “Developer ID” application, kernel extension, system extension, or installer certificate for your code-signing signature. (Don’t use a Mac Distribution, ad hoc, Apple Developer, or local development certificate.) Verify the certificate type before submitting, as described in Use a valid Developer ID certificate. For more information, see Synchronizing code signing identities with your developer account.

Enable the Hardened Runtime capability for your app and command line targets, as described in Enable hardened runtime.

Include a secure timestamp with your code-signing signature. (The Xcode distribution workflow includes a secure timestamp by default. For custom workflows, see Include a secure timestamp.)

Don’t include the com.apple.security.get-task-allow entitlement with the value set to any variation of true. If your software hosts third-party plug-ins and needs this entitlement to debug the plug-in in the context of a host executable, see Avoid the get-task-allow entitlement.

Link against the macOS 10.9 or later SDK, as described in Use the macOS 10.9 SDK or later.

Ensure your processes have properly-formatted XML, ASCII-encoded entitlements as described in Ensure properly formatted entitlements.

Apple recommends that you notarize all of the software that you’ve distributed, including older releases, and even software that doesn’t meet all of these requirements or that is unsigned. Apple’s notary service uses a variety of methods, including telemetry, to determine which of the above rules to relax for preexisting software. For more information, see Notarize your preexisting software.

8.1.7. Secure Code Notarizing macOS software before distribution Add the entitlements needed by plug-ins

When you enable the extra security enforced by the hardened runtime, as notarization requires, this impacts both your app and any plug-ins that your app hosts. Plug-ins don't declare their own entitlements. Instead, they inherit the entitlements of the host process. Therefore, a host app must include all the entitlements that prospective plug-ins require, even when the plug-ins are notarized separately.

For example, if a plug-in employs deep integration with the host executable via C function pointer overrides, or uses a JavaScript engine for custom workflows, the host executable must declare the Allow Unsigned Executable Memory Entitlement or Allow execution of JIT-compiled code entitlement, respectively. In some cases, a plug-in fails to even load if the host executable lacks the proper entitlement.

Be aware that even if your app doesn't provide a dedicated plug-in architecture, it might still load plug-ins, like drivers for professional mirrorless cameras and legacy DSLR cameras that don't conform to the driverless USB video device class (UVC) standard. If your app works with this kind of hardware, be sure to declare the Disable Library Validation Entitlement to load the corresponding plug-ins.

Also include resource access entitlements, like the Address Book or Location access entitlements, and the related purpose strings, that support your app's plug-ins. For example, if a Print Dialog Extension (PDE) that provides fax services wants to access a user's contact list, the host executable must declare the Address book entitlement and include the NSContactsUsageDescription purpose string in its Information Property List for the plug-in to operate.

For a complete list of hardened runtime entitlements, see Hardened Runtime. For information about usage strings, see Requesting access to protected resources.

8.1.8. Secure Code Preparing your app to work with pointer authentication Overview

The arm64e architecture introduces pointer authentication codes (PACs) to detect and guard against unexpected changes to pointers in memory. The addition of pointer authentication is transparent to most apps because the compiler manages the process. In rare cases — for example, if your app manipulates the stack directly, or if you pass pointers between C++ and Objective-C++ — you might have to adjust your code to work with PACs.

Pointer authentication works by offering a special CPU instruction to add a cryptographic signature — or PAC — to unused high-order bits of a pointer before storing the pointer. Another instruction removes and authenticates the signature after reading the pointer back from memory. Any change to the stored value between the write and the read invalidates the signature. The CPU interprets authentication failure as memory corruption and sets a high-order bit in the pointer, making the pointer invalid and causing the app to crash.

8.1.9. Secure Code Preparing your app to work with pointer authentication Build an arm64e binary to adopt pointer authentication

You automatically adopt pointer authentication in your app when you build and deploy a binary that targets the arm64e architecture. You can do this starting in Xcode 10.1. To build an arm64e slice, go to your target's build settings in Xcode and find the Architectures item. Click the current setting and choose Other. In the box that appears, add arm64e.

Devices using the Apple A12 or later A-series processor — like the iPhone XS, iPhone XS Max, iPhone XR, and Apple TV 4K (2nd generation) — support the arm64e architecture, as do the Apple Watch Series 4 or later, and Mac and iPads with Apple silicon. To test your adoption, you have to run your app on one of these devices. You can't test using the Simulator.

8.1.10. Secure Code Preparing your app to work with pointer authentication Recognize pointer authentication failures

When pointer authentication fails, the system invalidates the failing pointer by setting a high-order bit. Subsequent use of the pointer results in a segmentation fault. The crash report contains a message that includes the value of the pointer both after and before invalidation:
Exception Subtype: KERN_INVALID_ADDRESS at 0x0040000105394398 -> 0x0000000105394398 (possible pointer authentication failure)

Typically, the compiler adds the CPU instructions for both creating and authenticating the PAC. In the unusual case that you manage these steps yourself — for example, if you're authoring your own compiler — and if you try to use a signed pointer without first applying the authentication instruction to remove the signature, that also triggers a segmentation fault. In this case, the presence of the signature in the high-order bits invalidates the pointer:
Exception Subtype: KERN_INVALID_ADDRESS at 0x217c000105394398 -> 0x0000000105394398 (possible pointer authentication failure)

Be aware that other invalid memory accesses, where high-order bits are erroneously set, can also look like pointer authentication failures.

8.1.11. Secure Code Preparing your app to work with pointer authentication Update your code to avoid pointer authentication failures

Most code doesn't require modification to run with pointer authentication, with the possible exception of some low-level code that relies on arm64-specific behavior. For example, a crash reporting library that examines the stack contents needs to strip the PAC out of return addresses. The Apple Clang compiler provides utilities in the ptrauth.h header file — like the ptrauth_strip macro — to help with these kinds of tasks.

Pointer authentication can also expose latent bugs in existing code. In C++, it's incorrect to call a virtual method using a declaration that differs from its definition. In practice, such calls typically succeed in arm64, but trigger a pointer authentication failure in arm64e. You might encounter this bug when using OS_OBJECT types like dispatch_queue_t and xpc_connection_t. You can't pass instances of these types from C++ code to an Objective-C++ function (or vice versa) because they're defined differently in Objective-C++ to support automatic reference counting (ARC).

More generally, the PAC calculation takes into account the pointer value, one of several keys loaded into the CPU, and an optional salt value. To prevent reuse of pointers in different contexts, the PAC calculation depends on the pointer type. Keep these rules in mind when looking for possible issues in your code:
• Return addresses are signed with a key that's unique per process, using a salt derived from the stack pointer.
• Function pointers are signed with a key that's fixed across all processes, allowing sharing of library code between processes.
• Virtual method table entries are signed with a key that's shared across all apps, using a salt derived from the method signature.

8.1.12. Secure Code Protecting user data with App Sandbox Overview

Even if you adopted secure coding practices while developing your app, it may still have vulnerabilities that threaten your users' security and privacy. App Sandbox — a requirement for distributing your app on the App Store — limits the scope for an attacker to abuse platform features via your app.

When you create a new macOS app in Xcode, it receives the App Sandbox entitlement and a default set of capabilities. If you have an existing app, you can adopt App Sandbox to provide people with additional security.

8.1.13. Secure Code Hardened Runtime Overview

The Hardened Runtime, along with System Integrity Protection (SIP), protects the runtime integrity of your software by preventing certain classes of exploits, like code injection, dynamically linked library (DLL) hijacking, and process memory space tampering. To enable the Hardened Runtime for your app, navigate in Xcode to your target's Signing & Capabilities information and click the + button. In the window that appears, choose Hardened Runtime.

The Hardened Runtime doesn't affect the operation of most apps, but it does disallow certain less common capabilities, like just-in-time (JIT) compilation. If your app relies on a capability that the Hardened Runtime restricts, add an entitlement to disable an individual protection. You add an entitlement by enabling one of the runtime exceptions or access permissions listed in Xcode. Make sure to use only the entitlements that are absolutely necessary for your app's functionality.

You add entitlements only to executables. Shared libraries, frameworks, and in-process plug-ins inherit the entitlements of their host executable.

Due to their privileged position in the system, macOS refuses to load system extensions that use Hardened Runtime exception entitlements. There's one exception to this general rule: macOS allows the Allow execution of JIT-compiled code entitlement in non-DEXT system extensions.

The default value of these Boolean entitlements is false. When Xcode signs your code, it includes an entitlement only if the value is true. If you're manually signing code, follow this convention to ensure maximum compatibility. Don't include an entitlement if the value is false.

8.1.14. Secure Code Disabling and Enabling System Integrity Protection Overview

System Integrity Protection (SIP) in macOS protects the entire system by preventing the execution of unauthorized code. The system automatically authorizes apps that the user downloads from the App Store. The system also authorizes apps that a developer notarizes and distributes directly to users. The system prevents the launching of all other apps by default.

During development, it may be necessary for you to disable SIP temporarily to install and test your code. You don't need to disable SIP to run and debug apps from Xcode, but you might need to disable it to install system extensions, such as DriverKit drivers.

8.1.15. Secure Code Disabling and Enabling System Integrity Protection Disable System Integrity Protection Temporarily

To disable SIP, do the following:
1. Restart your computer in Recovery mode.
2. Launch Terminal from the Utilities menu.
3. Run the command csrutil disable.
4. Restart your computer.

8.1.16. Secure Code Disabling and Enabling System Integrity Protection Enable System Integrity Protection

To reenable SIP, do the following:
1. Restart your computer in Recovery mode.
2. Launch Terminal from the Utilities menu.
3. Run the command csrutil enable.
4. Restart your computer.

8.1.17. Cryptography Declare Your App's Use of Encryption Overview

Add the ITSAppUsesNonExemptEncryption key to your app's information property list with a Boolean value that indicates whether your app uses encryption. Set the value to NO if your app—including any third-party libraries it links against—doesn't use encryption, or if it only uses forms of encryption that are exempt from export compliance documentation requirements. Otherwise, set it to YES.

Typically, the use of encryption that's built into the operating system—for example, when your app makes HTTPS connections using URLSession—is exempt from export documentation upload requirements, whereas the use of proprietary encryption is not. To determine whether your use of encryption is considered exempt, see Determine and upload app encryption documentation.

8.1.18. Cryptography Provide Compliance Documentation Overview

If your app requires export compliance documentation, upload the required items to App Store Connect, as described in Determine and upload app encryption documentation. After successfully reviewing the documents, Apple provides you with a code. Add this string as the value for the ITSEncryptionExportComplianceCode key in your app's information property list.

8.1.19. Cryptography Certificate, Key, and Trust Services Overview

The certificate, key, and trust services API is a collection of functions and data structures that you use to conduct secure and authenticated data transactions. Specifically, you use this API to manage and use:
• Certificates and identities. A certificate is a collection of data that identifies its owner in a tamper-evident way. When you use a certificate to distribute a public key, a receiver can be confident of its origin. You can also package a certificate together with its corresponding private key in an identity object that you keep secret.
• Policies and trust services. When you receive a certificate, before you can use the embedded public key, you have to answer the question, "Can I trust this certificate?" You conduct an evaluation of trust according to a set of criteria, or a trust policy.
• Cryptographic keys. After you have a key whose origin you trust, you can begin to conduct cryptographic operations, such as encryption or data signing and verification. These operations in turn typically serve a larger purpose, such as authenticating a user, transmitting data securely, or verifying that a block of data is unaltered since being sealed with a signature.

8.1.20. Cryptography Cryptographic Message Syntax Services Overview

When you want to exchange data securely using the Multipurpose Internet Mail Extensions (MIME) protocol, you use the version of the protocol known as S/MIME defined in RFC 3851. This allows you to, among other things, ensure data integrity through digital signatures and data confidentiality through encryption. S/MIME in turn relies on the Cryptographic Message Syntax (CMS) protocol defined in RFC 3852 to carry out these operations.

Cryptographic message syntax services provides encoder objects that perform encryption using the CMS protocol's enveloped-data content type and sign using the signed-data content type. When a message is both signed and encrypted, the enveloped data content contains the signed data content. That is, the message is first signed and then the signed content is encrypted.

8.1.21. Cryptography Randomization Services Overview

Many security operations rely on randomization to avoid reproducibility. This is true for many complex cryptographic operations, such as key generation, but is also true for something as simple as generating a password string that can't be easily guessed. If the string's characters are truly random (and kept hidden), an attacker has no choice but to try every possible combination one at a time in a brute force attack. For sufficiently long strings, this becomes unfeasible.

But the strength of such a password depends on the quality of the randomization. True randomization is not possible in a deterministic system, such as one where software instructions from a bounded set are executed according to well-defined rules. But even "good" randomization (in a statistical sense) is difficult to produce under these conditions. If an attacker can infer patterns in insufficiently randomized data, your system becomes compromised. Use randomization services to generate a cryptographically secure set of random numbers.

8.1.22. Cryptography Security Transforms Overview

You use security transforms to assemble a chain of security-related operations that you apply to a stream of data in macOS.

8.1.23. Cryptography ASN.1 Overview

You use an ASN.1 coder to encode and decode both DER and BER data streams based on templates that you supply, which in turn are based upon ASN.1 specifications. You must import this API explicitly:

```
import Security.SecAsn1Coder
import Security.SecAsn1Templates
```

8.1.24. Result Codes Security Framework Result Codes Discussion

Use the SecCopyErrorMessageString(_:_:) function to obtain a human readable string corresponding to these status codes.

In addition to the codes listed here, certain Security framework services provide additional status codes that are specific to that service. In particular, see Authorization Services Result Codes, Sessions API Result Codes, Secure Transport Result Codes, Secure Download Result Codes, and Code Signing Services Result Codes.