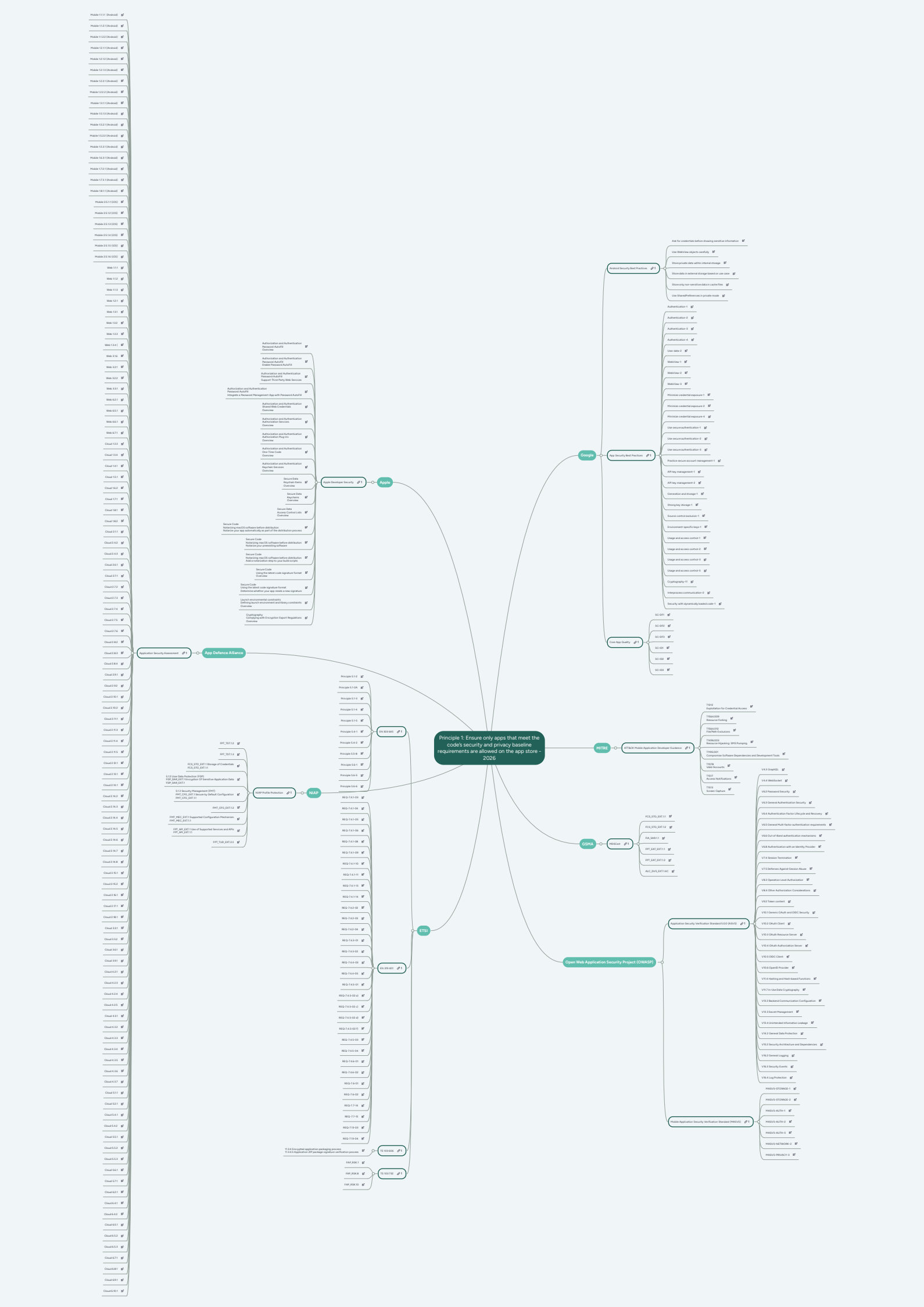

Mobile 1.1.1 (Android)
Mobile 1.1.2.1 (Android)
Mobile 1.1.2.2 (Android)
Mobile 1.2.1.1 (Android)
Mobile 1.2.1.2 (Android)
Mobile 1.2.1.3 (Android)
Mobile 1.2.2.1 (Android)
Mobile 1.2.2.2 (Android)
Mobile 1.3.1.1 (Android)
Mobile 1.5.1.3 (Android)
Mobile 1.5.2.1 (Android)
Mobile 1.5.2.2 (Android)
Mobile 1.5.4.1 (Android)
Mobile 1.6.3.1 (Android)
Mobile 1.7.2.1 (Android)
Mobile 1.7.1.1 (Android)
Mobile 1.8.1.1 (Android)
Mobile 2.5.11 (iOS)
Mobile 2.5.12 (iOS)
Mobile 2.5.13 (iOS)
Mobile 2.5.14 (iOS)
Mobile 2.5.15 (iOS)
Mobile 2.5.16 (iOS)
Web 1.1.1
Web 1.1.2
Web 1.1.3
Web 1.2.1
Web 1.3.1
Web 1.3.2
Web 1.3.3
Web 1.3.4.1
Web 3.1.6
Web 3.2.1
Web 3.2.2
Web 3.3.1
Web 4.2.1
Web 4.5.1
Web 4.6.1
Web 4.7.1
Cloud 1.3.3
Cloud 1.3.4
Cloud 1.4.1
Cloud 1.5.1
Cloud 1.6.2
Cloud 1.7.1
Cloud 1.8.1
Cloud 1.8.2
Cloud 2.1.1
Cloud 2.4.2
Cloud 2.4.3
Cloud 2.6.1
Cloud 2.7.1
Cloud 2.7.2
Cloud 2.7.3
Cloud 2.7.4
Cloud 2.7.5
Cloud 2.7.6
Cloud 2.8.2
Cloud 2.8.3
Cloud 2.8.4
Cloud 2.9.1
Cloud 2.9.2
Cloud 2.10.1
Cloud 2.10.2
Cloud 2.11.1
Cloud 2.11.3
Cloud 2.11.4
Cloud 2.11.5
Cloud 2.12.1
Cloud 2.13.1
Cloud 2.14.1
Cloud 2.16.2
Cloud 2.16.3
Cloud 2.16.4
Cloud 2.16.5
Cloud 2.16.6
Cloud 2.16.7
Cloud 2.16.8
Cloud 2.15.1
Cloud 2.15.2
Cloud 3.16.1
Cloud 2.17.1
Cloud 2.18.1
Cloud 3.1.1
Cloud 3.5.2
Cloud 3.6.1
Cloud 3.9.1
Cloud 4.2.1
Cloud 4.2.2
Cloud 4.2.4
Cloud 4.2.5
Cloud 4.3.1
Cloud 4.3.2
Cloud 4.3.3
Cloud 4.3.4
Cloud 4.3.5
Cloud 4.3.6
Cloud 4.3.7
Cloud 5.1.1
Cloud 5.2.1
Cloud 5.4.1
Cloud 5.4.2
Cloud 5.5.1
Cloud 5.5.2
Cloud 5.5.3
Cloud 5.6.1
Cloud 5.7.1
Cloud 6.2.1
Cloud 6.4.1
Cloud 6.4.2
Cloud 6.5.1
Cloud 6.5.2
Cloud 6.5.3
Cloud 6.7.1
Cloud 6.8.1
Cloud 6.9.1
Cloud 6.10.1

Principle 1: Ensure only apps that meet the code’s security and privacy baseline requirements are allowed on the app store - 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. Ask for credentials before showing sensitive information

> When requesting credentials from users so that they can access sensitive information or premium content in your app, ask for either a PIN/password/pattern or a biometric credential, such as face recognition or fingerprint recognition.

1.1.2. Use WebView objects carefully

> WebView objects in your app shouldn't let users navigate to sites that are outside of your control. Whenever possible, use an allowlist to restrict the content loaded by your app's WebView objects.
>
> In addition, never enable JavaScript interface support unless you completely control and trust the content in your app's WebView objects.

1.1.3. Store private data within internal storage

> Store all private user data within the device's internal storage, which is sandboxed per app. Your app doesn't need to request permission to view these files, and other apps can't access the files. As an added security measure, when the user uninstalls an app, the device deletes all files that the app saved within internal storage.

1.1.4. Store data in external storage based on use case

> Use external storage for large, non-sensitive files that are specific to your app as well as files that your app shares with other apps. The specific APIs that you use depend on whether your app is designed to access app-specific files or access shared files.
>
> If a file doesn't contain private or sensitive information but provides value to the user only in your app, store the file in an app-specific directory on external storage.
>
> If your app needs to access or store a file that provides value to other apps, use one of the following APIs, depending on your use case:
>
> Media files: To store and access images, audio files, and videos that are shared between apps, use the Media Store API.
> Other files: To store and access other types of shared files, including downloaded files, use the Storage Access Framework.

1.1.5. Store only non-sensitive data in cache files

> To provide faster access to non-sensitive app data, store it in the device's cache. For caches larger than 1 MB, use getExternalCacheDir(). For caches 1 MB or smaller, use getCacheDir(). Both methods provide you with the File object that contains your app's cached data.

1.1.6. Use SharedPreferences in private mode

When using getSharedPreferences() to create or access your app's SharedPreferences objects, use MODE_PRIVATE. That way, only your app can access the information within the shared preferences file.

If you want to share data across apps, don't use SharedPreferences objects. Instead, follow the steps to share data securely across apps.

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. Authentication-1

Authentication is a prerequisite for many key security operations. To control access to protected assets like user data, app functionality, and other resources, you'll need to add authentication to your Android app.

1.2.2. Authentication-2

You can improve your user's authentication experience by integrating your app with Credential Manager. Credential Manager is an Android Jetpack library that unifies API support for most major authentication methods, including passkeys, passwords, and federated sign-in solutions such as Sign-in with Google.

1.2.3. Authentication-3

To further enhance security for your app, consider adding biometric authentication methods such as fingerprint scans or facial recognition. Good candidates for adding biometric authentication might include apps for financial, healthcare, or identity management.

1.2.4. Authentication-4

Android's autofill framework can ease the sign-up and sign-in process, reducing error rates and user friction. Autofill integrates with password managers, allowing users to select complex, randomized passwords that can be stored and retrieved easily and securely.

1.2.5. User data-2

Authenticate your user whenever access to private data is required, and use modern authentication methods such as passkeys and Credential Manager. If your app needs to access personal information, keep in mind that some jurisdictions might require you to provide a privacy policy explaining your use and storage of that data. Follow the security best practice of minimizing access to user data to simplify compliance.

1.2.6. WebView-1

Because WebView consumes web content that can include HTML and JavaScript, improper use can introduce common web security issues such as cross-site scripting (JavaScript injection). Android includes a number of mechanisms to reduce the scope of these potential issues by limiting the capability of WebView to the minimum functionality required by your application.

1.2.7. WebView-2

If your application doesn't directly use JavaScript within a WebView, don't call setJavaScriptEnabled. Some sample code uses this method; if you repurpose sample code that uses it in a production application, remove that method call if it's not required. By default, WebView doesn't execute JavaScript, so cross-site scripting is not possible.

1.2.8. WebView-3

Use addJavaScriptInterface() with particular care, because it lets JavaScript invoke operations that are normally reserved for Android applications. If you use it, expose addJavaScriptInterface() only to web pages from which all input is trustworthy. If untrusted input is allowed, untrusted JavaScript might be able to invoke Android methods within your app. In general, we recommend exposing addJavaScriptInterface() only to JavaScript that is contained within your application APK.

1.2.9. Minimize credential exposure-1

Avoid unnecessary credential requests. To make phishing attacks more conspicuous and less likely to be successful, minimize the frequency of asking for user credentials. Instead, use an authorization token and refresh it. Request only the minimum amount of credential information necessary for authentication and authorization.

1.2.10. Minimize credential exposure-2

Store credentials securely. Use Credential Manager to enable passwordless authentication using passkeys or to implement federated sign-in using schemes such as Sign in with Google. If you must use traditional password authentication, don't store user IDs and passwords on the device. Instead, perform initial authentication using the username and password supplied by the user, and then use a short-lived, service-specific authorization token.

1.2.11. Minimize credential exposure-4

Limit access tokens. Use short-lived tokens operations and API calls.

1.2.12. Use secure authentication-1

Implement passkeys. Enable passkeys as a more secure and user-friendly upgrade to passwords.

1.2.13. Use secure authentication-2

Add biometrics. Offer the ability to use biometric authentication such as fingerprint or facial recognition for added security.

1.2.14. Use secure authentication-3

Use federated identity providers. Credential Manager supports federated authentication providers such as Sign in with Google.

1.2.15. Practice secure account management-1

Connect to services that are accessible to multiple applications using AccountManager. Use the AccountManager class to invoke a cloud-based service, and don't store passwords on the device.

1.2.16. API key management-1

API keys are a critical component of many Android apps, enabling them to access external services and perform essential functions such as connecting to mapping services, authentication, and weather services. However, exposing these sensitive keys can have severe consequences, including data breaches, unauthorized access, and financial losses. To prevent such scenarios, developers should implement secure strategies for handling API keys throughout the development process.

1.2.17. API key management-2

To protect services from misuse, API keys must be carefully protected. To secure a connection between the app and a service that uses an API key, you need to secure the access to the API. When your app is compiled, and your app's source code includes API keys, it's possible for an attacker to decompile the app and find these resources.

1.2.18. Generation and storage-1

Developers should treat API key storage as a critical component of data protection and user privacy using a defense-in-depth approach.

1.2.19. Strong key storage-1

For optimal key management security, use the Android Keystore, and encrypt stored keys using a robust tool such as Tink Java.

1.2.20. Source control exclusion-1

Never commit API keys to your source code repository. Adding API keys to source code risks exposure of keys to public repositories, shared code examples, and accidentally-shared files. Instead, use Gradle plugins such as the secrets-gradle-plugin to work with API keys in your project.

1.2.21. Environment-specific keys-1

If possible, use separate API keys development, testing, and production environments. Use environment-specific keys to isolate each environment, reducing the risk of exposing production data and allowing you to disable compromised keys without affecting your production environment.

1.2.22. Usage and access control-1

Generate unique keys for each app: Use separate API keys for each app to help identify and isolate compromised access.

1.2.23. Usage and access control-2

Implement IP restrictions: If possible, limit API key usage to specific IP addresses or ranges.

1.2.24. Usage and access control-3

Limit mobile app key usage: Limit API key usage to specific mobile apps by bundling them with the key or by using app certificates.

1.2.25. Usage and access control-5

Your service should provide features for restricting keys to a particular package or platform. For example, the Google Maps API limits key access by package name and signing key.

1.2.26. Cryptography-11

If you need to store a key for repeated use, use a mechanism, such as KeyStore, that provides long term storage and retrieval of cryptographic keys.

1.2.27. Interprocess communication-2

Many of the security elements are shared across IPC mechanisms. If your IPC mechanism isn't intended for use by other applications, set the android:exported attribute to false in the component's manifest element, such as for the element. This is useful for applications that consist of multiple processes within the same UID or if you decide late in development that you don't actually want to expose functionality as IPC, but you don't want to rewrite the code.

1.2.28. Security with dynamically loaded code-1

We strongly discourage loading code from outside of your application APK. Doing so significantly increases the likelihood of application compromise due to code injection or code tampering. It also adds complexity around version management and application testing—and it can make it impossible to verify the behavior of an application, so it might be prohibited in some environments.

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. SC-DF1

All sensitive data is stored in the app's internal storage.

1.3.2. SC-DF2

No personal or sensitive user data is logged to the system log or an app-specific log.

1.3.3. SC-DF3

The app does not use any non-resettable hardware IDs, such as the IMEI, for identification purposes.

1.3.4. SC-ID1

The app provides hints to autofill account credentials and other sensitive information, such as credit card info, physical address, and phone number.

1.3.5. SC-ID2

Integrate One Tap for Android for a seamless sign in experience

1.3.6. SC-ID3

The app supports biometric authentication to protect financial transactions or sensitive information, such as important user documents.

2. MITRE

2.1. ATT&CK Mobile Application Developer Guidance

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

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

2.1.1. T1212 Exploitation for Credential Access

> Application developers should consider taking measures to validate authentication requests by enabling one-time passwords, providing timestamps or sequence numbers for messages sent, using digital signatures, and/or using random session keys.

2.1.2. T1564.009 Resource Forking

> Configure applications to use the application bundle structure which leverages the /Resources folder location.

2.1.3. T1564.012 File/Path Exclusions

> Application developers should consider limiting the requirements for custom or otherwise difficult to manage file/folder exclusions. Where possible, install applications to trusted system folder paths that are already protected by restricted file and directory permissions.

2.1.4. T1496.003 Resource Hijacking: SMS Pumping

> Consider implementing CAPTCHA protection on forms that send messages via SMS.

2.1.5. T1195.001 Compromise Software Dependencies and Development Tools

> Application developers should be cautious when selecting third-party libraries to integrate into their application. Additionally, where possible, developers should lock software dependencies to specific versions rather than pulling the latest version on build.[6] GitHub Actions may be pinned to a specific commit hash rather than a tag or branch.

2.1.6. T1078 Valid Accounts

> Ensure that applications do not store sensitive data or credentials insecurely. (e.g. plaintext credentials in code, published credentials in repositories, or credentials in public cloud storage).

2.1.7. T1517 Access Notifications

> Application developers could be encouraged to avoid placing sensitive data in notification text.

2.1.8. T1513 Screen Capture

> Application developers can apply the FLAG_SECURE property to sensitive screens within their apps to make it more difficult for the screen contents to be captured.

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. FCS_STG_EXT.1.1

> The Target of Evaluation Security Functionality (TSF) shall provide [selection: hardware-based, hardware-isolated] secure cryptographic key storage.

3.1.2. FCS_STG_EXT.1.2

The TSF shall utilize the provided secure cryptographic key storage for protecting the key hierarchy.

3.1.3. FIA_SAR.1.1

The TSF shall provide a biometric verification mechanism with the Spoof Acceptance Rate [selection:
• BELOW 7%
• BETWEEN 7-20%
• ABOVE 20%].

3.1.4. FPT_EAT_EXT.1.1

The TSF shall only allow access to AT modem commands from [selection: the user interface, user-approved external connection, user-approved remote connection, [assignment: other interfaces]].

3.1.5. FPT_EAT_EXT.1.2

The TSF shall prompt for approval of AT modem commands sent to the device from outside the user interface [selection: at the time of the connection that will send the command, for each command].

NOTE: Each of the interfaces provides a method for user-approval. The direct user interface is implicitly approved as the device is unlocked and directly in use to enter the commands. How the user-approval is provided for any other (anything that is not the user interface) connection shall be described. If no commands can be sent, the open assignment can be used to specify “no interfaces.”

3.1.6. ALC_DVS_EXT.1.4C

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 data and keys provisioned to the device for a Root of Trust for the device.

4. Open Web Application Security Project (OWASP)

4.1. 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

4.1.1. V4.3 GraphQL

4.3.1
Verify that a query allowlist, depth limiting, amount limiting, or query cost analysis is used to prevent GraphQL or data layer expression Denial of Service (DoS) as a result of expensive, nested queries._x000D_

4.3.2
Verify that GraphQL introspection queries are disabled in the production environment unless the GraphQL API is meant to be used by other parties._x000D_

4.1.2. V4.4 WebSocket

4.4.2
Verify that, during the initial HTTP WebSocket handshake, the Origin header field is checked against a list of origins allowed for the application._x000D_

4.1.3. V6.2 Password Security

6.2.1
Verify that user set passwords are at least 8 characters in length although a minimum of 15 characters is strongly recommended._x000D_

6.2.10
Verify that a user’s password stays valid until it is discovered to be compromised or the user rotates it. The application must not require periodic credential rotation.

6.2.11
Verify that the documented list of context specific words is used to prevent easy to guess passwords being created._x000D_

6.2.12
Verify that passwords submitted during account registration or password changes are checked against a set of breached passwords.

6.2.2
Verify that users can change their password.

6.2.3
Verify that password change functionality requires the user’s current and new password.

6.2.4
Verify that passwords submitted during account registration or password change are checked against an available set of, at least, the top 3000 passwords which match the application’s password policy, e.g. minimum length.

6.2.5
Verify that passwords of any composition can be used, without rules limiting the type of characters permitted. There must be no requirement for a minimum number of upper or lower case characters, numbers, or special characters._x000D_

6.2.6
Verify that password input fields use type=password to mask the entry. Applications may allow the user to temporarily view the entire masked password, or the last typed character of the password._x000D_

6.2.7
Verify that “paste”functionality, browser password helpers, and external password managers are permitted.

6.2.8
Verify that the application verifies the user’s password exactly as received from the user, without any modifications such as truncation or case transformation.

6.2.9
Verify that passwords of at least 64 characters are permitted.

4.1.4. V6.3 General Authentication Security

6.3.1
Verify that controls to prevent attacks such as credential stuffing and password brute force are implemented according to the application's security documentation.

6.3.2
Verify that default user accounts (e.g., "root", "admin", or "sa") are not present in the application or are disabled._x000D_

6.3.3
Verify that either a multi-factor authentication mechanism or a combination of single-factor authentication mechanisms, must be used in order to access the application. For L3, one of the factors must be a hardware-based authentication mechanism which provides compromise and impersonation resistance against phishing attacks while verifying the intent to authenticate by requiring a user-initiated action (such as a button press on a FIDO hardware key or a mobile phone). Relaxing any of the considerations in this requirement requires a fully documented rationale and a comprehensive set of mitigating controls.

6.3.4
Verify that, if the application includes multiple authentication pathways, there are no undocumented pathways and that security controls and authentication strength are enforced consistently.

6.3.5
Verify that users are notified of suspicious authentication attempts (successful or unsuccessful). This may include authentication attempts from an unusual location or client, partially successful authentication (only one of multiple factors), an authentication attempt after a long period of inactivity or a successful authentication after several unsuccessful attempts.

6.3.6
Verify that email is not used as either a single-factor or multi-factor authentication mechanism.

6.3.7
Verify that users are notified after updates to authentication details, such as credential resets or modification of the username or email address.

6.3.8
Verify that valid users cannot be deduced from failed authentication challenges, such as by basing on error messages, HTTP response codes, or different response times. Registration and forgot password functionality must also have this protection._x000D_

4.1.5. V6.4 Authentication Factor Lifecycle and Recovery

6.4.1
Verify that system generated initial passwords or activation codes are securely randomly generated, follow the existing password policy, and expire after a short period of time or after they are initially used. These initial secrets must not be permitted to become the long term password.

6.4.2
Verify that password hints or knowledge-based authentication (so-called “secret questions”) are not present._x000D_

6.4.3
Verify that a secure process for resetting a forgotten password is implemented, that does not bypass any enabled multi-factor authentication mechanisms.

6.4.4
Verify that if a multi-factor authentication factor is lost, evidence of identity proofing is performed at the same level as during enrollment._x000D_

6.4.5
Verify that renewal instructions for authentication mechanisms which expire are sent with enough time to be carried out before the old authentication mechanism expires, configuring automated reminders if necessary._x000D_

6.4.6
Verify that administrative users can initiate the password reset process for the user, but that this does not allow them to change or choose the user’s password. This prevents a situation where they know the user’s password.

4.1.6. V6.5 General Multi-factor authentication requirements

6.5.1
Verify that lookup secrets, out-of-band authentication requests or codes, and time-based one-time passwords (TOTPs) are only successfully usable once._x000D_

6.5.2
Verify that, when being stored in the application's backend, lookup secrets with less than 112 bits of entropy (19 random alphanumeric characters or 34 random digits) are hashed with an approved password storage hashing algorithm that incorporates a 32-bit random salt. A standard hash function can be used if the secret has 112 bits of entropy or more.

6.5.4
Verify that lookup secrets and out-of-band authentication codes have a minimum of 20 bits of entropy (typically 4 random alphanumeric characters or 6 random digits is sufficient).

6.5.5
Verify that out-of-band authentication requests, codes, or tokens, as well as time-based one-time passwords (TOTPs) have a defined lifetime. Out of band requests must have a maximum lifetime of 10 minutes and for TOTP a maximum lifetime of 30 seconds._x000D_

6.5.6
Verify that any authentication factor (including physical devices) can be revoked in case of theft or other loss.

6.5.7
Verify that biometric authentication mechanisms are only used as secondary factors together with either something you have or something you know.

6.5.8
Verify that time-based one-time passwords (TOTPs) are checked based on a time source from a trusted service and not from an untrusted or client provided time._x000D_

4.1.7. V6.6 Out-of-Band authentication mechanisms

6.6.1
Verify that authentication mechanisms using the Public Switched Telephone Network (PSTN) to deliver One-time Passwords (OTPs) via phone or SMS are offered only when the phone number has previously been validated, alternate stronger methods (such as Time based One-time Passwords) are also offered, and the service provides information on their security risks to users. For L3 applications, phone and SMS must not be available as options.

6.6.2
Verify that out-of-band authentication requests, codes, or tokens are bound to the original authentication request for which they were generated and are not usable for a previous or subsequent one.

6.6.3
Verify that a code based out-of-band authentication mechanism is protected against brute force attacks by using rate limiting. Consider also using a code with at least 64 bits of entropy.

6.6.4
Verify that, where push notifications are used for multi-factor authentication, rate limiting is used to prevent push bombing attacks. Number matching may also mitigate this risk.

4.1.8. V6.8 Authentication with an Identity Provider

6.8.1
Verify that, if the application supports multiple identity providers (IdPs), the user's identity cannot be spoofed via another supported identity provider (eg. by using the same user identifier). The standard mitigation would be for the application to register and identify the user using a combination of the IdP ID (serving as a namespace) and the user's ID in the IdP.

6.8.2
Verify that the presence and integrity of digital signatures on authentication assertions (for example on JWTs or SAML assertions) are always validated, rejecting any assertions that are unsigned or have invalid signatures._x000D_

6.8.3
Verify that SAML assertions are uniquely processed and used only once within the validity period to prevent replay attacks._x000D_

6.8.4
Verify that, if an application uses a separate Identity Provider (IdP) and expects specific authentication strength, methods, or recentness for specific functions, the application verifies this using the information returned by the IdP. For example, if OIDC is used, this might be achieved by validating ID Token claims such as 'acr', 'amr', and 'auth_time'(if present). If the IdP does not provide this information, the application must have a documented fallback approach that assumes that the minimum strength authentication mechanism was used (for example, single-factor authentication using username and password)._x000D_

4.1.9. V7.4 Session Termination

7.4.4
Verify that all pages that require authentication have easy and visible access to logout functionality.

4.1.10. V7.5 Defenses Against Session Abuse

7.5.1
Verify that the application requires full re-authentication before allowing modifications to sensitive account attributes which may affect authentication such as email address, phone number, MFA configuration, or other information used in account recovery._x000D_

7.5.3
Verify that the application requires further authentication with at least one factor or secondary verification before performing highly sensitive transactions or operations._x000D_

4.1.11. V8.3 Operation Level Authorization

8.3.2
Verify that changes to values on which authorization decisions are made are applied immediately. Where changes cannot be applied immediately, (such as when relying on data in self-contained tokens), there must be mitigating controls to alert when a consumer performs an action when they are no longer authorized to do so and revert the change. Note that this alternative would not mitigate information leakage._x000D_

8.3.3
Verify that access to an object is based on the originating subject's (e.g. consumer's) permissions, not on the permissions of any intermediary or service acting on their behalf. For example, if a consumer calls a web service using a self-contained token for authentication, and the service then requests data from a different service, the second service will use the consumer's token, rather than a machine-to-machine token from the first service, to make permission decisions.

4.1.12. V8.4 Other Authorization Considerations

8.4.2
Verify that access to administrative interfaces incorporates multiple layers of security, including continuous consumer identity verification, device security posture assessment, and contextual risk analysis, ensuring that network location or trusted endpoints are not the sole factors for authorization even though they may reduce the likelihood of unauthorized access.

4.1.13. V9.2 Token content

9.2.1
Verify that, if a validity time span is present in the token data, the token and its content are accepted only if the verification time is within this validity time span. For example, for JWTs, the claims 'nbf'and 'exp'must be verified._x000D_

9.2.2
Verify that the service receiving a token validates the token to be the correct type and is meant for the intended purpose before accepting the token's contents. For example, only access tokens can be accepted for authorization decisions and only ID Tokens can be used for proving user authentication.

9.2.3
Verify that the service only accepts tokens which are intended for use with that service (audience). For JWTs, this can be achieved by validating the 'aud' claim against an allowlist defined in the service._x000D_

9.2.4
Verify that, if a token issuer uses the same private key for issuing tokens to different audiences, the issued tokens contain an audience restriction that uniquely identifies the intended audiences. This will prevent a token from being reused with an unintended audience. If the audience identifier is dynamically provisioned, the token issuer must validate these audiences in order to make sure that they do not result in audience impersonation.

4.1.14. V10.1 Generic OAuth and OIDC Security

10.1.1
Verify that tokens are only sent to components that strictly need them. For example, when using a backend-for-frontend pattern for browser-based JavaScript applications, access and refresh tokens shall only be accessible for the backend.

10.1.2
Verify that the client only accepts values from the authorization server (such as the authorization code or ID Token) if these values result from an authorization flow that was initiated by the same user agent session and transaction. This requires that client-generated secrets, such as the proof key for code exchange (PKCE) ‘code_verifier’, ‘state’or OIDC ‘nonce’, are not guessable, are specific to the transaction, and are securely bound to both the client and the user agent session in which the transaction was started._x000D_

4.1.15. V10.2 OAuth Client

10.2.2
Verify that, if the OAuth client can interact with more than one authorization server, it has a defense against mix-up attacks. For example, it could require that the authorization server return the ‘iss’parameter value and validate it in the authorization response and the token response.

10.2.3
Verify that the OAuth client only requests the required scopes (or other authorization parameters) in requests to the authorization server.

4.1.16. V10.3 OAuth Resource Server

10.3.1
Verify that the resource server only accepts access tokens that are intended for use with that service (audience). The audience may be included in a structured access token (such as the ‘aud’claim in JWT), or it can be checked using the token introspection endpoint.

10.3.2
Verify that the resource server enforces authorization decisions based on claims from the access token that define delegated authorization. If claims such as ‘sub’, ‘scope’, and ‘authorization_details’are present, they must be part of the decision._x000D_

10.3.3
Verify that if an access control decision requires identifying a unique user from an access token (JWT or related token introspection response), the resource server identifies the user from claims that cannot be reassigned to other users. Typically, it means using a combination of ‘iss’and ‘sub’claims.

10.3.4
Verify that, if the resource server requires specific authentication strength, methods, or recentness, it verifies that the presented access token satisfies these constraints. For example, if present, using the OIDC ‘acr’, ‘amr’and ‘auth_time’claims respectively.

10.3.5
Verify that the resource server prevents the use of stolen access tokens or replay of access tokens (from unauthorized parties) by requiring sender-constrained access tokens, either Mutual TLS for OAuth 2 or OAuth 2 Demonstration of Proof of Possession (DPoP).

4.1.17. V10.4 OAuth Authorization Server

10.4.1
Verify that the authorization server validates redirect URIs based on a client-specific allowlist of pre-registered URIs using exact string comparison.

10.4.10
Verify that confidential client is authenticated for client-to-authorized server backchannel requests such as token requests, pushed authorization requests (PAR), and token revocation requests.

10.4.11
Verify that the authorization server configuration only assigns the required scopes to the OAuth client._x000D_

10.4.12
Verify that for a given client, the authorization server only allows the ‘response_mode’value that this client needs to use. For example, by having the authorization server validate this value against the expected values or by using pushed authorization request (PAR) or JWT-secured Authorization Request (JAR).

10.4.13
Verify that grant type ‘code’ is always used together with pushed authorization requests (PAR).

10.4.15
Verify that, for a server-side client (which is not executed on the end-user device), the authorization server ensures that the ‘authorization_details’ parameter value is from the client backend and that the user has not tampered with it. For example, by requiring the usage of pushed authorization request (PAR) or JWT-secured Authorization Request (JAR).

10.4.2
Verify that, if the authorization server returns the authorization code in the authorization response, it can be used only once for a token request. For the second valid request with an authorization code that has already been used to issue an access token, the authorization server must reject a token request and revoke any issued tokens related to the authorization code.

10.4.3
Verify that the authorization code is short-lived. The maximum lifetime can be up to 10 minutes for L1 and L2 applications and up to 1 minute for L3 applications._x000D_

10.4.4
Verify that for a given client, the authorization server only allows the usage of grants that this client needs to use. Note that the grants ‘token’(Implicit flow) and ‘password’(Resource Owner Password Credentials flow) must no longer be used._x000D_

10.4.5
Verify that the authorization server mitigates refresh token replay attacks for public clients, preferably using sender-constrained refresh tokens, i.e., Demonstrating Proof of Possession (DPoP) or Certificate-Bound Access Tokens using mutual TLS (mTLS). For L1 and L2 applications, refresh token rotation may be used. If refresh token rotation is used, the authorization server must invalidate the refresh token after usage, and revoke all refresh tokens for that authorization if an already used and invalidated refresh token is provided.

10.4.6
Verify that, if the code grant is used, the authorization server mitigates authorization code interception attacks by requiring proof key for code exchange (PKCE). For authorization requests, the authorization server must require a valid ‘code_challenge’value and must not accept a

‘code_challenge_method’value of ‘plain’. For a token request, it must require validation of the ‘code_verifier’parameter.

10.4.7
Verify that if the authorization server supports unauthenticated dynamic client registration, it mitigates the risk of malicious client applications. It must validate client metadata such as any registered URIs, ensure the user’s consent, and warn the user before processing an authorization request with an untrusted client application.

10.4.8
Verify that refresh tokens have an absolute expiration, including if sliding refresh token expiration is applied.

4.1.18. V10.5 OIDC Client

10.5.1
Verify that the client (as the relying party) mitigates ID Token replay attacks. For example, by ensuring that the ‘nonce’claim in the ID Token matches the ‘nonce’value sent in the authentication request to the OpenID Provider (in OAuth2 refereed to as the authorization request sent to the authorization server).

10.5.2
Verify that the client uniquely identifies the user from ID Token claims, usually the ‘sub’claim, which cannot be reassigned to other users (for the scope of an identity provider).

10.5.3
Verify that the client rejects attempts by a malicious authorization server to impersonate another authorization server through authorization server metadata. The client must reject authorization server metadata if the issuer URL in the authorization server metadata does not exactly match the pre-configured issuer URL expected by the client.

10.5.4
Verify that the client validates that the ID Token is intended to be used for that client (audience) by checking that the ‘aud’claim from the token is equal to the ‘client_id’value for the client.

4.1.19. V10.6 OpenID Provider

10.6.1
Verify that the OpenID Provider only allows values ‘code’, ‘ciba’, ‘id_token’, or ‘id_token code’for response mode. Note that ‘code’is preferred over ‘id_token code’(the OIDC Hybrid flow), and ‘token’(any Implicit flow) must not be used.

4.1.20. V11.4 Hashing and Hash-based Functions

11.4.2
Verify that passwords are stored using an approved, computationally intensive, key derivation function (also known as a “password hashing function”), with parameter settings configured based on current guidance. The settings should balance security and performance to make brute-force attacks sufficiently challenging for the required level of security._x000D_

4.1.21. V11.7 In-Use Data Cryptography

11.7.2
Verify that data minimization ensures the minimal amount of data is exposed during processing, and ensure that data is encrypted immediately after use or as soon as feasible.

4.1.22. V13.2 Backend Communication Configuration

13.2.2
Verify that communications between backend application components, including local or operating system services, APIs, middleware, and data layers, are performed with accounts assigned the least necessary privileges.

13.2.3
Verify that if a credential has to be used for service authentication, the credential being used by the consumer is not a default credential (e.g., root/root or admin/admin)._x000D_

13.2.4
Verify that an allowlist is used to define the external resources or systems with which the application is permitted to communicate (e.g., for outbound requests, data loads, or file access). This allowlist can be implemented at the application layer, web server, firewall, or a combination of different layers._x000D_

13.2.5
Verify that the web or application server is configured with an allowlist of resources or systems to which the server can send requests or load data or files from._x000D_

4.1.23. V13.3 Secret Management

13.3.2
Verify that access to secret assets adheres to the principle of least privilege.

4.1.24. V13.4 Unintended Information Leakage

13.4.1
Verify that the application is deployed either without any source control metadata, including the .git or .svn folders, or in a way that these folders are inaccessible both externally and to the application itself.

13.4.2
Verify that debug modes are disabled for all components in production environments to prevent exposure of debugging features and information leakage.

13.4.3
Verify that web servers do not expose directory listings to clients unless explicitly intended._x000D_

13.4.4
Verify that using the HTTP TRACE method is not supported in production environments, to avoid potential information leakage.

4.1.25. V14.2 General Data Protection

14.2.4
Verify that controls around sensitive data related to encryption, integrity verification, retention, how the data is to be logged, access controls around sensitive data in logs, privacy and privacy-enhancing technologies, are implemented as defined in the documentation for the specific data’s protection level.

4.1.26. V15.2 Security Architecture and Dependencies

15.2.3
Verify that the production environment only includes functionality that is required for the application to function, and does not expose extraneous functionality such as test code, sample snippets, and development functionality._x000D_

4.1.27. V16.2 General Logging

16.2.5
Verify that when logging sensitive data, the application enforces logging based on the data's protection level. For example, it may not be allowed to log certain data, such as credentials or payment details. Other data, such as session tokens, may only be logged by being hashed or masked, either in full or partially.

4.1.28. V16.3 Security Events

16.3.1
Verify that all authentication operations are logged, including successful and unsuccessful attempts. Additional metadata, such as the type of authentication or factors used, should also be collected._x000D_

4.1.29. V16.4 Log Protection

16.4.2
Verify that logs are protected from unauthorized access and cannot be modified.

4.2. 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

4.2.1. MASVS-STORAGE-1

The app securely stores sensitive data.

Apps handle sensitive data coming from many sources such as the user, the backend, system services or other apps on the device and usually need to store it locally. The storage locations may be private to the app (e.g. its internal storage) or be public and therefore accessible by the user or other installed apps (e.g. public folders such as Downloads). This control ensures that any sensitive data that is intentionally stored by the app is properly protected independently of the target location._x000D_

4.2.2. MASVS-STORAGE-2

The app prevents leakage of sensitive data.

There are cases when sensitive data is unintentionally stored or exposed to publicly accessible locations; typically as a side-effect of using certain APIs, system capabilities such as backups or logs. This control covers this kind of unintentional leaks where the developer actually has a way to prevent it.

4.2.3. MASVS-AUTH-1

The app uses secure authentication and authorization protocols and follows the relevant best practices.

Most apps connecting to a remote endpoint require user authentication and also enforce some kind of autho_x0002_rization. While the enforcement of these mechanisms must be on the remote endpoint, the apps also have to ensure that it follows all the relevant best practices to ensure a secure use of the involved protocols._x000D_

4.2.4. MASVS-AUTH-2

The app performs local authentication securely according to the platform best practices.

Many apps allow users to authenticate via biometrics or a local PIN code. These authentication mechanisms need to be correctly implemented. Additionally, some apps might not have a remote endpoint, and rely fully on local app authentication.

4.2.5. MASVS-AUTH-3

The app secures sensitive operations with additional authentication.

Some additional form of authentication is often desirable for sensitive actions inside the app. This can be done in different ways (biometric, pin, MFA code generator, email, deep links, etc) and they all need to b implemented securely.

4.2.6. MASVS-NETWORK-2

The app performs identity pinning for all remote endpoints under the developer's control.

Instead of trusting all the default root CAs of the framework or device, this control will make sure that only very specific CAs are trusted. This practice is typically called certificate pinning or public key pinning.

4.2.7. MASVS-PRIVACY-3

The app is transparent about data collection and usage.

Users have the right to know how their data is being used. This control ensures that apps provide clear information about data collection, storage, and sharing practices, including any behavior a user wouldn't reasonably expect, such as background data collection. Apps should also adhere to platform guidelines on data declarations._x000D_

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 5.1-2

Where pre-installed unique per device passwords are used to authenticate users against the device or for machine-to-machine authentication, these shall be generated with a mechanism that reduces the risk of automated attacks against a class or type of consumer IoT device.

EXAMPLE 2: Pre-installed passwords are sufficiently randomized.

As a counter-example, passwords with incremental counters (such as "password1", "password2" and so on) are easily guessable. Further, using a password that is related in an obvious way to public information (sent over the air or within a network), such as MAC address or Wi-Fi® SSID, can allow for password retrieval using automated means.

5.1.2. Principle 5.1-2A

Passwords should not be used for machine-to-machine authentication.

Using human-memorable passwords for machine-to-machine authentication is generally not appropriate, but pre-shared keys can be appropriate when their use follows best practice.

5.1.3. Principle 5.1-3

Authentication mechanisms used to authenticate users against the consumer IoT device or used for machine-to-machine authentication shall use best practice cryptography, appropriate to the properties of the technology, operating environment, risk and usage.

5.1.4. Principle 5.1-4

Where a user can authenticate against a consumer IoT device, the consumer IoT device shall provide to the user or an administrator a simple mechanism to change the authentication value used.

EXAMPLE 3: For biometric authentication values the device manufacturer allows this change in authentication value through retraining against a new biometric.

EXAMPLE 4: A parent in a household creates an account on the consumer IoT device for their child and selects and manages the PIN or password that the child uses. The parent is an administrator on the consumer IoT device and can restrict the child from changing the PIN or password.

EXAMPLE 5: To make it simple for the user to change a password, the manufacturer designs the password change process in a way that it requires a minimal number of steps. The manufacturer explains the process in a user manual and in a video tutorial.

An authentication mechanism used for authenticating users, whether it be a fingerprint, password or other token, needs to have its value changeable. This is easier when this mechanism is part of the normal usage flow of the consumer IoT device.

5.1.5. Principle 5.1-5

The consumer IoT device shall have a mechanism available which makes successful brute-force attacks on authentication mechanisms via network interfaces impracticable unless the device has a resource constraint determined by the use case that prevents the implementation.

EXAMPLE 6: A consumer IoT device has a limitation on the number of authentication attempts within a certain time interval. It also uses increasing time intervals between attempts.

EXAMPLE 7: The client application is able to lock an account or to delay additional authentication attempts after a limited number of failed authentication attempts.

This provision addresses attacks that perform "credential stuffing" or exhaust an entire key-space. It is important that these types of attacks are detected by the consumer IoT device and defended against, whilst guarding against a related threat of "resource exhaustion" and denial of service attacks.

5.1.6. Principle 5.4-1

Sensitive security parameters in persistent storage shall be stored securely by the consumer IoT device.

NOTE 1: Stored critical security parameters need adequate measures to prevent both modification and disclosure while stored public security parameters only need adequate measures to prevent modification.

Secure storage mechanisms can be used to secure sensitive security parameters. Appropriate mechanisms include those
provided by a Trusted Execution Environment (TEE), encrypted storage associated with the hardware, Secure Elements (SE) or Dedicated Security Components (DSC), and processing capabilities of software running on a UICC, according to ETSI TR 121 905 [i.29], ETSI TS 102 221 [i.25], embedded UICC according to GSMA SGP.22 Technical Specification v2.2.1 [i.26].

NOTE 2: This provision applies to persistent storage, but manufacturers can also implement similar approaches for sensitive security parameters in memory.

EXAMPLE 1: The root keys involved in authorization and access to licensed radio frequencies (e.g. LTE-m cellular access) are stored in a UICC.

EXAMPLE 2: A remote controlled door-lock using a Trusted Execution Environment (TEE) to store and access the sensitive security parameters.

EXAMPLE 3: A wireless thermostat stores the credentials for the wireless network in a tamper protected microcontroller rather than in external flash storage.

5.1.7. Principle 5.4-2

Where a hard-coded unique per device identity is used in a consumer IoT device for security purposes, it shall be implemented in such a way that it resists tampering by means such as physical, electrical or software.

EXAMPLE 4: A master key used for network access that is unique to the consumer IoT device is stored in UICC which is compliant to relevant ETSI standards (for example ETSI TS 102 221 [i.25]).

5.1.8. Principle 5.5-8

The manufacturer shall follow secure management processes for critical security parameters that relate to the consumer IoT device and for the entire lifecycle of the critical security parameters.

EXAMPLE 4: Open, peer-reviewed standards for critical security parameters (commonly referred to as "key management").

5.1.9. Principle 5.6-1

All unused network interfaces and all unused logical interfaces that are accessible through a network interface shall be disabled.

Disabling unused network and logical interfaces can significantly reduce the attack surface of a consumer IoT device.These interfaces may have been used only during the development phase of the device and are no longer required for normal operation, or may be specific to certain environments or purposes. By default, these interfaces are to be disabled before the device is made available to the public. The disabling can be realized for example by appropriate access control mechanisms that are for example preventing unauthorized access to or misuse of the unused interfaces. However, users may have the ability to manually enable any disabled interface that is required for their specific usage. In such a case it is to be ensured that the enabling capability cannot be misused easily, e.g. by appropriate access control and authentication mechanisms.

EXAMPLE 1: An administrative UI that is supposed to be accessed from the LAN is not accessible from the WAN by default.

EXAMPLE 2: A Direct Firmware Update service exposed over Bluetooth® Low Energy is used for development but not expected to be used in production. It is disabled in the final product.

5.1.10. Principle 5.6-5

The manufacturer should only enable software services that are used or required for the intended use or operation of the consumer IoT device.

EXAMPLE 7: The manufacturer does not provision the consumer IoT device with any background processes, kernel extensions, commands, programs or tools that are not required for the intended use.

5.1.11. Principle 5.6-6

Code should be minimized to the functionality necessary for the consumer IoT device to operate.

EXAMPLE 8: "Dead" or unused code is removed and not considered to be benign.

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-7.4.1-03

The TSP shall provide setting up specific accounts to be used for administrative purposes like installation, configuration, management or maintenance.

5.2.2. REQ-7.4.1-04

Privileged accounts shall be used only if the privileges are necessary for the specific activity.

5.2.3. REQ-7.4.1-05

Strong identification, authentication and authorisation procedures shall be used for privileged accounts.

5.2.4. REQ-7.4.1-06

Where appropriate, the TSP shall ensure that users and devices are authenticated by multi-factor or continuous authentication mechanisms, such as secure voice, video and text, before accessing the TSP's network and ITS information systems, depending on the classification of the systems to be accessed.

5.2.5. REQ-7.4.1-08

The TSP shall:
a) assign and revoke access rights based on the principles of need-to-know, least privilege and separation of duties;
b) ensure that access rights are modified accordingly upon termination or change of employment;
c) ensure that access to network and information systems is authorised by the relevant persons;
d) ensure that access rights appropriately address third-party access, such as visitors, suppliers and service
providers, in particular by limiting access rights in scope and in duration;
e) maintain a register of access rights granted; and
f) apply logging to the management of access rights.

5.2.6. REQ-7.4.1-09

Access to information and application system functions shall be restricted in accordance with the access control policy.

5.2.7. REQ-7.4.1-10

The TSP's system shall provide sufficient computer security controls for the separation of trusted roles identified in TSP's practices, including the separation of security administration and operation functions. Particularly, use of system utility programs shall be restricted and controlled.

5.2.8. REQ-7.4.1-11

TSP's personnel shall be identified and authenticated before using critical applications related to the service.

5.2.9. REQ-7.4.1-13

Sensitive data shall be protected against being revealed through re-used storage objects (e.g. deleted files) or storage media (see clause 7.3.2) being accessible to unauthorized users.

5.2.10. REQ-7.4.1-14

The TSP shall review and, where appropriate, update the access control policies at planned intervals and when significant incidents or significant changes to operations or risks occur.

5.2.11. REQ-7.4.2-02

The policies for privileged accounts and system administration accounts shall:
a) establish strong identification, authentication such as multi-factor authentication, and authorisation procedures for privileged accounts and system administration accounts;
b) set up specific accounts to be used for system administration operations exclusively, such as installation, configuration, management or maintenance;
c) individualise and restrict system administration privileges to the highest extent possible;
d) provide that system administration accounts are only used to connect to system administration systems.

5.2.12. REQ-7.4.2-03

The TSP shall review access rights of privileged and system administration accounts at planned intervals and be modified based on organisational changes.

5.2.13. REQ-7.4.2-04

The TSP shall document the results of the review, including the necessary changes of access rights.

5.2.14. REQ-7.4.3-01

The TSP shall restrict and control the use of system administration systems in accordance with the access control policy.

5.2.15. REQ-7.4.3-02

For system administration systems, the TSP shall:
a) only use system administration systems for system administration purposes, and not for any other operations;
b) separate logically such systems from application software not used for system administrative purposes; and
c) protect access to system administration systems through authentication and encryption.

5.2.16. REQ-7.4.4-03

The TSP shall only permit identities assigned to multiple persons, such as shared identities, where they are necessary for business or operational reasons and are subject to an explicit approval process and documentation.

5.2.17. REQ-7.4.4-05

The TSP shall regularly review the identities for network and information systems and their users and, if no longer needed, deactivate them without delay.

5.2.18. REQ-7.4.5-01

The TSP shall implement secure authentication procedures and technologies based on access restrictions and the policy on access control.

5.2.19. REQ-7.4.5-02 a)

Ensure the strength of authentication is appropriate to the classification of the asset to be accessed.

5.2.20. REQ-7.4.5-02 c)

Require the change of authentication credentials initially, at predefined intervals and upon suspicion that the credentials were compromised.

5.2.21. REQ-7.4.5-02 d)

Require the reset of authentication credentials and the blocking of users after a predefined number of unsuccessful log-in attempts.

5.2.22. REQ-7.4.5-02 f)

Require separate credentials to access privileged access or administrative accounts.

5.2.23. REQ-7.4.5-03

The TSP shall to the extent feasible use state-of-the-art authentication methods, in accordance with the associated assessed risk and the classification of the asset to be accessed, and unique authentication information.

5.2.24. REQ-7.4.5-04

The TSP shall review the authentication procedures and technologies at planned intervals.

5.2.25. REQ-7.4.6-01

The TSP shall ensure that users are authenticated by multiple authentication factors or continuous authentication mechanisms for accessing the TSP's network and information systems, where appropriate, in accordance with the classification of the asset to be accessed.

5.2.26. REQ-7.4.6-02

The TSP shall ensure that the strength of authentication is appropriate for the classification of the asset to be accessed.

5.2.27. REQ-7.6-01

The TSP shall control physical access to components of the TSP's system whose security is critical to the provision of its trust services, prevent or reduce the consequences of events originating from physical and environmental threats, and minimize risks related to physical security.

5.2.28. REQ-7.6-02

Physical access to components of the TSP's system whose security is critical to the provision of its trust services shall be limited to authorized individuals.

5.2.29. REQ-7.7-14

Before developing a network and information system, including software, the TSP shall lay down rules for the secure development of network and information systems and apply them when developing network and information systems in-house, or when outsourcing the development of network and information systems. The rules shall cover all development phases, including specification, design, development, implementation and testing.

5.2.30. REQ-7.7-15

For secure development, the TSP shall:
a) carry out an analysis of security requirements at the specification and design phases of any development or acquisition project undertaken by the TSP or on behalf of those entities;
b) apply principles for engineering secure systems and secure coding principles to any information system development activities such as promoting cybersecurity-by-design, zero-trust architectures;
c) lay down security requirements regarding development environments;
d) establish and implement security testing processes in the development life cycle;
e) appropriately select, protect and manage security test data;
f) sanitise and anonymise testing data according to the risk assessment.

5.2.31. REQ-7.13-03

Trust services provided and end user products used in the provision of those services shall be made accessible for persons with disabilities, where feasible.

5.2.32. REQ-7.13-04

Applicable standards on accessibility such as ETSI EN 301 549 [i.6] should be taken into account.

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. 11.3.4 Encrypted application packaging process 11.3.4.5 Application ZIP package signature verification process

After decrypting the encrypted application package as defined in clause 11.3.4.4, terminals shall verify the resulting CMS SignedData according to the following process.

Terminals shall use the Operator Signing Root CA to verify the certificates included in the certificates block of the CMS SignedData structure as detailed in section 5.1 of IETF RFC 5652 [12].

Terminals shall extract the application ZIP file from the encapContentInfo block of the CMS SignedData. Terminals shall fail and reject the verification if any of the following conditions occur:
• The certificate chain fails certificate path validation as defined in clause 6 of IETF RFC 5280 [11] (this includes a check that none of the certificates have expired). The required check that certificates have not been revoked shall be performed by obtaining the appropriate CRLs using the cRLDistributionPoints extension (see table 23).
• The Operator Name, as signalled via the Organization ('O=') attribute of the subject field, or the organisation_id, as signalled via the CommonName ('CN=') attribute of the subject field do not match those defined in the bilateral agreement for the operator whose organisation_id is found during the discovery process in clause 6.1.5.
• The value of the message-digest field contained in the CMS SignedData structure does not match with the terminal generating a message-digest of the extracted application ZIP file when applying the hashing function communicated via the SignatureAlgorithm field.

If verification fails, the terminal shall follow the process outlined in clause 6.1.9.

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. FAP_RSK.1

Applications are required to store data in prescribed locations and key storage systems.
FAP_RSK.1.1 The applications on the TOE shall only store data within their own storage context.
FAP_RSK.1.2 The applications on the TOE shall only store keys using the TSF-provided key storage.

5.4.2. FAP_RSK.8

Application data requires protection through all interfaces by which the data may be exposed from the device.
FAP_RSK.8.1 The applications on the TOE shall not support unauthorized direct access to data from external sources.

5.4.3. FAP_RSK.10

Debugging modes can expose application data, and so production applications are required to not be configured to be debuggable.
FAP_RSK.10.1 The applications on the TOE shall not have any debug functionality enabled.

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. FPT_TST.1.2

The TSF shall provide authorized users with the capability to verify the integrity of [[DRBG seed/output data]].

6.1.2. FPT_TST.1.3

The TSF shall provide authorized users with the capability to verify the integrity of [[TSF DRBG specified in FCS_RBG.1]].

6.1.3. FCS_STO_EXT.1 Storage of Credentials FCS_STO_EXT.1.1

The application shall [selection:

not store any credentials
invoke the functionality provided by the platform to securely store [assignment: list of credentials]
securely store [assignment: list of credentials] with platform provided [selection:
[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
] and cryptographic key size of 256-bits.
PBKDF2 function that uses [selection:
HMAC-SHA256
HMAC-SHA384
HMAC-SHA512
] with [assignment: positive integer of 1,000 or greater] iterations and output cryptographic key size of [assignment: positive integer of 256 of greater] bits that meet the following [NIST SP 800-132].
]
implement functionality to securely store [assignment: list of credentials] according to [selection: FCS_COP.1/SKC, FCS_PBKDF_EXT.1]

] to non-volatile memory.

6.1.4. 5.1.2 User Data Protection (FDP) FDP_DAR_EXT.1 Encryption Of Sensitive Application Data FDP_DAR_EXT.1

The application shall [selection:

leverage platform-provided functionality to encrypt sensitive data
implement functionality to encrypt sensitive data as defined in the PP-Module for File Encryption
protect sensitive data in accordance with FCS_STO_EXT.1
not store any sensitive data

] in non-volatile memory.

6.1.5. 5.1.3 Security Management (FMT) FMT_CFG_EXT.1 Secure by Default Configuration FMT_CFG_EXT.1.1

The application shall [selection: not use credentials, use platform-provided credentials, provide only enough functionality to set new credentials when configured with default credentials or no credentials for application provided credentials].

6.1.6. FMT_CFG_EXT.1.2

The application shall be configured by default with file permissions which protect the application binaries and data files from modification by normal unprivileged users.

6.1.7. FMT_MEC_EXT.1 Supported Configuration Mechanism FMT_MEC_EXT.1.1

The application shall [selection: invoke the mechanisms recommended by the platform vendor for storing and setting configuration options, implement functionality to encrypt and store configuration options as defined by FDP_PRT_EXT.1 in the PP-Module for File Encryption].

6.1.8. FPT_API_EXT.1 Use of Supported Services and APIs FPT_API_EXT.1.1

The application shall use only documented platform APIs.

6.1.9. FPT_TUD_EXT.2.3

> The application installation package shall be digitally signed such that [selection:
> • its platform can cryptographically verify them prior to installation.
> • the application can verify them using [selection: Leighton-Micali Signature. , eXtended Merkle Signature Scheme.]
>]

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 1.1.1.1 (Android)

> The app shall securely store sensitive data in external storage.

7.1.2. Mobile 1.1.2.1 (Android)

> The Keyboard Cache shall be disabled for sensitive data inputs.

7.1.3. Mobile 1.1.2.2 (Android)

> No sensitive data shall be stored in system logs.

7.1.4. Mobile 1.2.1.1 (Android)

> No insecure random number generators shall be utilized for any security sensitive context

7.1.5. Mobile 1.2.1.2 (Android)

> No insecure operations shall be used for symmetric cryptography.

7.1.6. Mobile 1.2.1.3 (Android)

> Strong cryptography shall be implemented according to industry best practices.

7.1.7. Mobile 1.2.2.1 (Android)

> Cryptographic keys shall only be used for their defined purpose.

7.1.8. Mobile 1.2.2.2 (Android)

> Cryptographic key management shall be implemented properly.

7.1.9. Mobile 1.3.1.1 (Android)

> If using OAuth 2.0 to authenticate, Proof Key for Code Exchange (PKCE) shall be implemented to protect the code grant.

7.1.10. Mobile 1.5.1.3 (Android)

> Any sensitive functionality exposed via IPC shall be intentional and at the minimum required level.

7.1.11. Mobile 1.5.2.1 (Android)

WebViews shall securely execute JavaScript.

7.1.12. Mobile 1.5.2.2 (Android)

WebView shall be configured to allow the minimum set of protocol handlers required while disabling potentially dangerous handlers.

7.1.13. Mobile 1.5.3.1 (Android)

The app shall by default mask data in the User Interface when it is known to be sensitive.

7.1.14. Mobile 1.6.3.1 (Android)

Compiler security features shall be enabled.

7.1.15. Mobile 1.7.2.1 (Android)

The app shall disable all debugging symbols in the production version.

7.1.16. Mobile 1.7.3.1 (Android)

The app shall not be debuggable if installed from outside of commercial app stores.

7.1.17. Mobile 1.8.1.1 (Android)

The app shall minimize access to sensitive data and resources provided by the platform.

7.1.18. Mobile 2.5.1.1 (iOS)

The app shall not expose sensitive data via IPC mechanisms.

7.1.19. Mobile 2.5.1.2 (iOS)

The app shall not expose sensitive data via App Extensions.

7.1.20. Mobile 2.5.1.3 (iOS)

The app shall not expose sensitive functionality via Custom URL Schemes.

7.1.21. Mobile 2.5.1.4 (iOS)

The app shall not expose sensitive data via UIActivity Sharing.

7.1.22. Mobile 2.5.1.5 (iOS)

The app shall not use the general pasteboard for sharing sensitive information.

7.1.23. Mobile 2.5.1.6 (iOS)

The app shall not expose sensitive functionality via Universal Links.

7.1.24. Web 1.1.1

Authentication is resistant to brute force attacks

7.1.25. Web 1.1.2

System generated initial passwords or activation codes shall be securely randomly generated and expire after a short period.

7.1.26. Web 1.1.3

Passwords shall be stored in a form that is resistant to offline attacks.

7.1.27. Web 1.2.1

Default credentials shall not present on publicly exposed interfaces.

7.1.28. Web 1.3.1

Out of band verifier shall expire in a reasonable timeframe.

7.1.29. Web 1.3.2

Out of band verifier shall only be used once.

7.1.30. Web 1.3.3

Out of band verifier shall be securely random.

7.1.31. Web 1.3.4 (

Out of band verifier shall be resistant to brute force attacks.

7.1.32. Web 3.1.6

Directory browsing shall be disabled unless deliberately desired.

7.1.33. Web 3.2.1

Application shall implement only secure and recommended OAuth 2.0 flows, such as the Authorization Code Flow or the Authorization Code Flow with PKCE, while avoiding the use of deprecated flows like the Implicit Flow or the Resource Owner Password Credentials Flow.

7.1.34. Web 3.2.2

Ensure that the application securely validates the redirect_uri and state parameters during the OAuth 2.0 authorization process to prevent open redirect and CSRF vulnerabilities.

7.1.35. Web 3.3.1

Application administrative interfaces shall use appropriate multi-factor authentication to prevent unauthorized use.

7.1.36. Web 6.2.1

Disable debug modes in production environments.

7.1.37. Web 6.5.1

The application shall not log credentials or payment details. Session tokens shall only be stored in logs in an irreversible, hashed form.

7.1.38. Web 6.6.1

Browser storage is securely cleared during logout.

7.1.39. Web 6.7.1

The application shall securely store access tokens, API keys, and other server-side secrets.

7.1.40. Cloud 1.3.3

Azure: Ensure FTP deployments are Disabled.

7.1.41. Cloud 1.3.4

Google: Ensure “Block Project-Wide SSH Keys” Is Enabled for VM Instances.

7.1.42. Cloud 1.4.1

Azure: Ensure Virtual Machines are utilizing Managed Disks.

7.1.43. Cloud 1.5.1

Google: Ensure That IP Forwarding Is Not Enabled on Instances.

7.1.44. Cloud 1.6.2

Google: Ensure That Instances Are Not Configured To Use the Default Service Account With Full Access to All Cloud APIs.

7.1.45. Cloud 1.7.1

Google: Ensure ‘Enable Connecting to Serial Ports’ Is Not Enabled for VM Instance.

7.1.46. Cloud 1.8.1

Azure: Ensure that Register with Azure Active Directory is enabled on App Service.

7.1.47. Cloud 1.8.2

Google: Ensure Oslogin Is Enabled for a Project.

7.1.48. Cloud 2.1.1

Azure: Ensure the Key Vault is Recoverable.

7.1.49. Cloud 2.4.2

Azure: Ensure that 'Users can add gallery apps to My Apps' is set to 'No'.

7.1.50. Cloud 2.4.3

Azure: Ensure That ‘Users Can Register Applications’ Is Set to ‘No’.

7.1.51. Cloud 2.6.1

Google: Ensure Secrets are Not Stored in Cloud Functions Environment Variables by Using Secret Manager.

7.1.52. Cloud 2.7.1

AWS: Ensure no 'root' user account access key exists.

7.1.53. Cloud 2.7.2

AWS: Do not setup access keys during initial user setup for all IAM users that have a console password.

7.1.54. Cloud 2.7.3

AWS: Ensure IAM policies that allow full ":" administrative privileges are not attached.

7.1.55. Cloud 2.7.4

Azure: Ensure That 'Guest users access restrictions' is set to 'Guest user access is restricted to properties and memberships of their own directory objects'.

7.1.56. Cloud 2.7.5

Google: Ensure That IAM Users Are Not Assigned the Service Account User or Service Account Token Creator Roles at Project Level.

7.1.57. Cloud 2.7.6

Google: Ensure That Cloud KMS Cryptokeys Are Not Anonymously or Publicly Accessible.

7.1.58. Cloud 2.8.2

AWS: Ensure IAM password policy requires minimum length of 14 or greater.

7.1.59. Cloud 2.8.3

AWS: Ensure there is only one active access key available for any single IAM user.

7.1.60. Cloud 2.8.4

AWS: Ensure access keys are rotated every 90 days or less.

7.1.61. Cloud 2.9.1

AWS: Ensure IAM password policy prevents password reuse.

7.1.62. Cloud 2.9.2

Azure: Ensure that a Custom Bad Password List is set to 'Enforce' for your Organization.

7.1.63. Cloud 2.10.1

AWS: Ensure credentials unused for 45 days or greater are disabled.

7.1.64. Cloud 2.10.2

Azure: Ensure Guest Users Are Reviewed on a Regular Basis.

7.1.65. Cloud 2.11.1

AWS: Eliminate use of the 'root' user for administrative and daily tasks.

7.1.66. Cloud 2.11.3

Azure: Ensure That 'Restrict access to Azure AD administration portal' is Set to 'Yes'.

7.1.67. Cloud 2.11.4

Azure: Ensure That No Custom Subscription Administrator Roles Exist.

7.1.68. Cloud 2.11.5

Google: Ensure That Service Account Has No Admin Privileges.

7.1.69. Cloud 2.12.1

Google: Ensure that Corporate Login Credentials are Used.

7.1.70. Cloud 2.13.1

Azure: Ensure that 'Number of days before users are asked to re-confirm their authentication information' is set to '90'.

7.1.71. Cloud 2.14.1

Azure: Ensure That 'Number of methods required to reset' is set to '2'.

7.1.72. Cloud 2.14.2

Azure: Ensure that 'Require Multi-Factor Authentication to register or join devices with Azure AD' is set to 'Yes'.

7.1.73. Cloud 2.14.3

Azure: Ensure that 'Multi-Factor Auth Status' is 'Enabled' for all Privileged Users.

7.1.74. Cloud 2.14.4

Azure: Ensure that 'Allow users to remember multi-factor authentication on devices they trust' is Disabled.

7.1.75. Cloud 2.14.5

Azure: Ensure that A Multi-factor Authentication Policy Exists for All Users.

7.1.76. Cloud 2.14.6

Azure: Ensure Multi-factor Authentication is Required for Risky Sign-ins.

7.1.77. Cloud 2.14.7

Google: Ensure that Multi-Factor Authentication is 'Enabled' for All Non-Service Accounts.

7.1.78. Cloud 2.14.8

Azure: Ensure that 'Multi-Factor Auth Status' is 'Enabled' for all Non-Privileged Users.

7.1.79. Cloud 2.15.1

Azure: Ensure that A Multi-factor Authentication Policy Exists for Administrative Groups.

7.1.80. Cloud 2.15.2

Azure: Ensure Multi-factor Authentication is Required for Azure Management.

7.1.81. Cloud 2.16.1

AWS: Ensure MFA is enabled for the 'root' user account.

7.1.82. Cloud 2.17.1

Azure: Ensure that 'Notify users on password resets?' is set to 'Yes'.

7.1.83. Cloud 2.18.1

AWS: Ensure IAM Users Receive Permissions Only Through Groups.

7.1.84. Cloud 3.3.1

Azure: Ensure That 'All users with the following roles' is set to 'Owner'.

7.1.85. Cloud 3.5.2

Azure: Ensure the Storage Container Storing the Activity Logs is not Publicly Accessible.

7.1.86. Cloud 3.6.1

Azure: Ensure Any of the ASC Default Policy Settings are Not Set to 'Disabled'.

7.1.87. Cloud 3.9.1

AWS: Ensure management console sign-in without MFA is monitored.

7.1.88. Cloud 4.2.1

Google: Ensure Legacy Networks Do Not Exist for Older Projects.

7.1.89. Cloud 4.2.3

Google: Ensure That RSASHA1 Is Not Used for the Key-Signing Key in Cloud DNS DNSSEC.

7.1.90. Cloud 4.2.4

Google: Ensure That RSASHA1 Is Not Used for the Zone-Signing Key in Cloud DNS DNSSEC.

7.1.91. Cloud 4.2.5

AWS: Ensure that EC2 Metadata Service only allows IMDSv2.

7.1.92. Cloud 4.3.1

Azure: Ensure that RDP access from the Internet is evaluated and restricted.

7.1.93. Cloud 4.3.2

Azure: Ensure that SSH access from the Internet is evaluated and restricted.

7.1.94. Cloud 4.3.3

Google: Ensure That SSH Access Is Restricted From the Internet.

7.1.95. Cloud 4.3.4

Google: Ensure That RDP Access Is Restricted From the Internet.

7.1.96. Cloud 4.3.5

AWS: Ensure no Network ACLs allow ingress from 0.0.0.0/0 to remote server administration ports.

7.1.97. Cloud 4.3.6

AWS: Ensure no security groups allow ingress from 0.0.0.0/0 to remote server administration ports.

7.1.98. Cloud 4.3.7

AWS: Ensure no security groups allow ingress from ::/0 to remote server administration ports.

7.1.99. Cloud 5.1.1

Azure: Ensure Soft Delete is Enabled for Azure Containers and Blob Storage.

7.1.100. Cloud 5.2.1

Azure: Ensure Default Network Access Rule for Storage Accounts is Set to Deny.

7.1.101. Cloud 5.4.1

AWS: Ensure EBS Volume Encryption is Enabled in all Regions.

7.1.102. Cloud 5.4.2

AWS: Ensure that encryption is enabled for EFS file systems.

7.1.103. Cloud 5.5.1

AWS: Ensure that S3 Buckets are configured with 'Block public access (bucket settings)'.

7.1.104. Cloud 5.5.2

Azure: Ensure that 'Public access level' is disabled for storage accounts with blob containers.

7.1.105. Cloud 5.5.3

Google: Ensure That Cloud Storage Bucket Is Not Anonymously or Publicly Accessible.

7.1.106. Cloud 5.6.1

Azure: Ensure that 'Enable key rotation reminders' is enabled for each Storage Account.

7.1.107. Cloud 5.7.1

Azure: Ensure that Storage Account Access Keys are Periodically Regenerated.

7.1.108. Cloud 6.2.1

Google: Ensure 'external scripts enabled' database flag for Cloud SQL SQL Server instance is set to 'off'.

7.1.109. Cloud 6.4.1

AWS: Ensure that encryption-at-rest is enabled for RDS Instances.

7.1.110. Cloud 6.4.2

Azure: Ensure that 'Data encryption' is set to 'On' on a SQL Database.

7.1.111. Cloud 6.5.1

AWS: Ensure that public access is not given to RDS Instance.

7.1.112. Cloud 6.5.2

Azure: Ensure no Azure SQL Databases allow ingress from 0.0.0.0/0 (ANY IP).

7.1.113. Cloud 6.5.3

Google: Ensure That Cloud SQL Database Instances Do Not Implicitly Whitelist All Public IP Addresses.

7.1.114. Cloud 6.7.1

Azure: Ensure 'Allow access to Azure services' for PostgreSQL Database Server is disabled.

7.1.115. Cloud 6.8.1

Google: Ensure Instance IP assignment is set to private.

7.1.116. Cloud 6.9.1

Google: Ensure That a MySQL Database Instance Does Not Allow Anyone To Connect With Administrative Privileges.

7.1.117. Cloud 6.10.1

Google: Ensure 'remote access' database flag for Cloud SQL SQL Server instance is set to 'off'.

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 Password AutoFill Overview

Password AutoFill simplifies login and account creation tasks for iOS apps and webpages. With just a few taps, your users can create and save new passwords or log in to an existing account. Users don’t even need to know their password; the system handles everything. This convenience increases the likelihood that users will complete your app’s onboarding process and start using your app more quickly. Additionally, by encouraging users to select unique, strong passwords, you increase the security of your app.

By default, Password AutoFill saves the user’s login credentials on their current iOS device. iOS can sync these credentials securely across the user’s devices using iCloud Keychain. Password AutoFill recommends credentials only for your app’s associated domain, and the user must authenticate using Face ID or Touch ID before accessing these credentials. For more information on privacy and security, see Approach to Privacy and iOS Security Guide.

Password AutoFill also provides credentials from third-party password managers that implement a credential provider extension. For more information on the credential provider extension, see the Authentication Services framework.

8.1.2. Authorization and Authentication Password AutoFill Enable Password AutoFill

Password AutoFill uses heuristics to determine when the user logs in or creates new passwords, and automatically provides the password QuickType bar. These heuristics give users some Password AutoFill support in most apps, even if those apps haven’t been updated to support AutoFill. However, to provide the best user experience and ensure your app fully supports Password AutoFill, perform the following steps:

Set up your app’s associated domains. To learn how to set up your app’s associated domains, see Supporting associated domains.

Set the correct AutoFill type on relevant text fields. For an iOS app, see Enabling Password AutoFill on a text input view. For a web app, see Enabling Password AutoFill on an HTML input element.

8.1.3. Authorization and Authentication Password AutoFill Support Third-Party Web Services

Password AutoFill streamlines logging into web services at your domain; however, if you need to log into a third-party service, use ASWebAuthenticationSession instead, which supports Password AutoFill when your user hasn’t already logged in.

8.1.4. Authorization and Authentication Password AutoFill Integrate a Password Management App with Password AutoFill

If you’re developing a password management app, create AutoFill Credential Provider Extensions to surface credentials from your app in Password AutoFill and pull your app’s password data into the Password AutoFill workflow. When your app integrates with Password AutoFill, users don’t have to copy and paste credentials. Instead, they can use password data stored in your app easily because the data will be offered to the user to fill in compatible user name and password fields. To integrate a password app with Password AutoFill, use in the Authentication Services framework.

8.1.5. Authorization and Authentication Shared Web Credentials Overview

The Security.SecSharedCredentials API provides functions for storing and requesting shared password-based credentials. Users often save their username and password in their iCloud keychain when logging into websites in Safari. Later, they may run a native app from the same developer to access the same account. With shared web credentials, the app can access the credentials stored for the website instead of requiring the user to reenter a username and password. Users can also create new accounts, update passwords, or delete their accounts from within the app. These changes are then saved and used by Safari.

8.1.6. Authorization and Authentication Authorization Services Overview

The Security.Authorization API is a programming interface to the Security Server and its policy database. This API facilitates access control to restricted areas of the operating system and allows you to restrict a user's access to particular features in your macOS app. Use authorization services in:

Software that restricts access to its own tools

Applications that call system tools

Software installers that install privileged tools or require access to restricted areas of the operating system

As shown in the image below, the Security Server is a daemon running in the operating system that provides a trusted implementation of various security protocols, including authorization computation. In turn, the Security Server relies on the Security Agent to interface with users when authentication is needed. Thus an app can verify credentials (usernames and passwords) without ever accessing them directly. This authorization process also allows the means of authentication to change in the future (such as adding Touch ID) without your having to modify your app.

8.1.7. Authorization and Authentication Authorization Plug-ins Overview

Use plug-ins to extend macOS authorization services to perform authorizations in a new way or to implement a new policy that is too complex to be implemented entirely with the authorization policy database.

You must import this API explicitly:

import Security.AuthorizationPlugin

When your plug-in needs to interact with the user, subclass the SFAuthorizationPluginView class to maintain the look and feel of the system authentication dialogs.

8.1.8. Authorization and Authentication One-Time Code Overview

iOS detects when an SMS message contains a one-time code and offers that code for AutoFill in one-time code fields in apps and websites. Improve the usability of one-time codes in your apps and websites by setting the content type of fields where people enter one-time codes. Improve security by adopting domain-bound codes to make it harder for people to enter one-time codes into phishing websites.

8.1.9. Authorization and Authentication Keychain Services Overview

Computer users often have small secrets that they need to store securely. For example, most people manage numerous online accounts. Remembering a complex, unique password for each is impossible, but writing them down is both insecure and tedious. Users typically respond to this situation by recycling simple passwords across many accounts, which is also insecure.

The keychain services API helps you solve this problem by giving your app a mechanism to store small bits of user data in an encrypted database called a keychain. When you securely remember the password for them, you free the user to choose a complicated one.

The keychain is not limited to passwords, as shown in Figure 1. You can store other secrets that the user explicitly cares about, such as credit card information or even short notes. You can also store items that the user needs but may not be aware of. For example, the cryptographic keys and certificates that you manage with Certificate, Key, and Trust Services enable the user to engage in secure communications and to establish trust with other users and devices. You use the keychain to store these items as well.

8.1.10. Secure Data Keychain Items Overview

When you want to store a secret such as a password or cryptographic key, you package it as a keychain item. Along with the data itself, you provide a set of publicly visible attributes both to control the item's accessibility and to make it searchable. As shown in Figure 1, keychain services handles data encryption and storage (including data attributes) in a keychain, which is an encrypted database stored on disk. Later, authorized processes use keychain services to find the item and decrypt its data.

8.1.11. Secure Data Keychains Overview

In iOS, apps have access to a single keychain (which logically encompasses the iCloud keychain). This keychain is automatically unlocked when the user unlocks the device and then locked when the device is locked. An app can access only its own keychain items, or those shared with a group to which the app belongs. It can't manage the keychain container itself.

In macOS, however, the system supports an arbitrary number of keychains. You typically rely on the user to manage these with the Keychain Access app and work implicitly with the default keychain, much as you would in iOS. Nevertheless, the keychain services API does provide functions that you can use to manipulate keychains directly. For example, you can create and manage a keychain that is private to your app. On the other hand, robust access control mechanisms typically make this unnecessary for anything other than an app trying to replicate the keychain access utility.

8.1.12. Secure Data Access Control Lists Overview

In macOS, for items not stored on the iCloud keychain, each protected keychain item—like a password or private key—has an associated access instance that contains an access control list (ACL). The entries in this list in turn each contain an array of operations and an array of apps trusted to carry out those operations with the item. The collection of ACL entries govern the accessibility of the corresponding keychain item.

When an app attempts to access a keychain item for a particular purpose—like using a private key to sign a document—the system looks for an entry in the item's ACL containing the operation. If there's no entry that lists the operation, then the system denies access and it's up to the calling app to try something else or to notify the user.

If there is an entry that lists the operation, the system checks whether the calling app is among the entry's trusted apps. If so, the system grants access. Otherwise, the system prompts the user for confirmation. The user may choose to Deny, Allow, or Always Allow the access. In the latter case, the system adds the app to the list of trusted apps for that entry, enabling the app to gain access in the future without prompting the user again.

8.1.13. Secure Code Notarizing macOS software before distribution Notarize your app automatically as part of the distribution process

Before distributing your app directly to customers, your Account Holder must sign the app with your Developer ID. Xcode's Organizer window includes a workflow for generating a distributable version of your app. This workflow includes an option to notarize your macOS app automatically. To notarize your app using this workflow, do the following:
1. Open your Xcode project.
2. Create an archive of your app.
2. Choose Window > Organizer to open the Organizer window.
4. In the Archives tab, select the archive you created.
5. Click Distribute App to view the distribution options.
6. Choose Developer ID for your method of distribution.
7. Click Next.
8. Choose Upload to send your archive to the Apple notary service.
9. Click Next.

When you click Next, Xcode uploads your archive to the notary service. When the upload is complete, the notary service begins the scanning process, which usually takes less than an hour. While the notary service scans your software, you can continue to prepare your archive for distribution. For example, you can export the archive and perform any final testing that you require prior to making your software available to customers.

When the notarization process finishes, Xcode downloads the ticket and staples it to your archive. At that point, export your archive again to receive a distributable version of your software that includes the notary ticket. For more information about how to use the Xcode UI to upload your software, see Upload a macOS app to be notarized.

If Xcode doesn't let you upload for notarization, be sure that you are building a macOS archive, as described in Notarize macOS apps with external dependencies. For targets other than macOS apps, use the notarytool command line utility to notarize, as described in Customizing the notarization workflow.

8.1.14. Secure Code Notarizing macOS software before distribution Notarize your preexisting software

Notarizing your preexisting software lets Gatekeeper warn users when they try to run it. It also helps the notary service distinguish your legitimate software from variants that have been tampered with. You can notarize an existing disk image, installer package, or ZIP archive containing your app.

To notarize your preexisting software, do the following:
1. Make your active Xcode installation one that supports notarization by using the xcode-select command-line tool. For information about how to use this tool, see its man page, as described in Reading UNIX Manual Pages.
2. Upload your software to the Apple notary service, as described in Upload your app to the notarization service.
3. Staple the returned ticket to your existing software, as described in Staple the ticket to your distribution.

For tips on how to resolve issues that can occur during notarization, see Resolving common notarization issues.

8.1.15. Secure Code Notarizing macOS software before distribution Add a notarization step to your build scripts

If you use an automated build system, you can integrate the notarization process into your existing build scripts. The notarytool and stapler command-line tools (included with Xcode) allow you to upload your software to the Apple notary service, and to staple the resulting ticket to your executable. Alternatively, you can interact directly with the notary service using the Notary API.

For information about how to incorporate notarization into your custom build scripts, see Customizing the notarization workflow.

8.1.16. Secure Code Using the latest code signature format Overview

Before you distribute an app, you apply a code signature to it. The signature certifies that you are the app's creator and enables the system to detect unwanted modifications — whether accidental or malicious — that happen after you sign your app. As a security measure, iOS refuses to launch an app that has a missing or invalid signature.

Starting in iOS 15, iPadOS 15, tvOS 15, and watchOS 8, the system checks for a new, more secure signature format that uses Distinguished Encoding Rules, or DER, to embed entitlements into your app's signature. Apps signed with a previous signature format will not launch.

8.1.17. Secure Code Using the latest code signature format Determine whether your app needs a new signature

This change to DER embeddded entitlements won't affect most apps. For apps that you distribute through the App Store or TestFlight, App Store Connect first verifies your signature, and then re-signs the app using an Apple identity, before making the app available for download. Apps available through these channels already have the new signature format.

For apps that you distribute other ways, such as Ad Hoc or through the Apple Developer Enterprise Program, Xcode and the codesign utility have created signatures that use the new format for several years. If you signed your app on a Mac running macOS 10.14 through macOS 11, the app already has the new signature format, but your signature may not include the required DER entitlements. macOS 11 and later will sign app bundles with the new signature format that include the DER entitlements by default.

8.1.18. Launch environmental constraints Defining launch environment and library constraints Overview

You define launch environment and library constraints in constraint dictionaries that you either save in launchd property list files, or in separate property list files that you use in code signing. The constraint dictionaries you create contain facts and operations. Facts are assertions that properties of the executable the operating system is launching, or the library your process is loading, match conditions you specify. Operations allow for rich combinations of facts.

The top level of a constraint dictionary is implicitly an $and operation that includes all of the facts and operations in the dictionary. When one process tries to launch another process — by calling execve(_:_:_:) or posix_spawn(_:_:_:_:_:_:) — the operating system checks that the executable file satisfies its own self constraint. It also checks that the parent process's executable satisfies the executable's parent constraint, and that the responsible process's executable satsifies the executable's responsible process constraint. If any of these launch constraints aren't satisfied, the operating system doesn't run the program.

Your process can load a dynamic library if all of the facts and operations at the top level of the library constraint dictionary are true for the file that contains the library. If any part of the library constraint isn't true, your process doesn't load the library.

launchd tests constraints that you specify in launchd property list files when it needs to start your launch daemon or agent. If the executable specified in the launchd property list doesn't satisfy the constraint in the property list, then launchd doesn't start the process.

8.1.19. Cryptography Complying with Encryption Export Regulations Overview

When you submit your app to TestFlight or the App Store, you upload your app to a server in the United States. If you distribute your app outside the U.S. or Canada, your app is subject to U.S. export laws, regardless of where your legal entity is based. If your app uses, accesses, contains, implements, or incorporates encryption, this is considered an export of encryption software, which means your app is subject to U.S. export compliance requirements, as well as the import compliance requirements of the countries where you distribute your app.

Every time you submit a new version of your app, App Store Connect asks you questions to guide you through a compliance review. You can bypass these questions and streamline the submission process by providing the required information in your app's Information Property List file.